ÉTUDES & DÉVELOPPEMENT

UML 2 ET MDE

Ingénierie des modèles avec études de cas

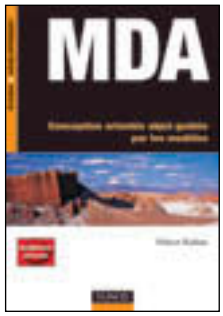
Compléments sur le web

Franck Barbier

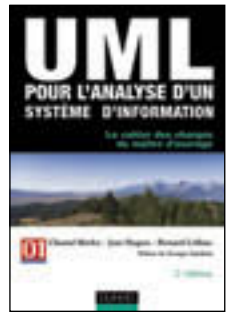
DUNOD

UML 2 ET MDE

Ingénierie des modèles avec études de cas



MDA
Conception orientée objet guidée par les modèles
Hubert Kadima
240 pages
Dunod, 2005



UML
pour l'analyse d'un système d'information
2e édition
Chantal Morley
Jean Hugues
Bernard Leblanc
256 pages
Dunod, 2002

Penser objet avec UML et JAVA
3e édition
Michel Lai
232 pages
Dunod, 2004

UML 2 ET MDE

Ingénierie des modèles avec études de cas

Franck Barbier

Professeur des universités

DUNOD

Illustration de couverture : digitalvision®

ISBN 2 10 049526 7

Table des matières

Avant-propos

À Flora

Recherche, vous avez dit « recherche » ?

La science, c'est le doute. Si vous cherchez des certitudes, ne croyez pas les trouver dans cet ouvrage. Ne pas tricher conduit à douter et la science sous-entend le doute. On accuse beaucoup les chercheurs de ne pas être des « trouveurs ». « Trouveur » est un néologisme qui confirme l'opinion répandue qu'il n'y en a pas beaucoup, ou pas du tout. « Des chercheurs on en trouve ; des trouveurs on en cherche. » (Charles de Gaulle, approximativement.) Bref, même si cet ouvrage privilégie pragmatisme, cas concrets, solutions éprouvées, une démarche d'industriel du logiciel en somme, il reste néanmoins le fait d'un chercheur qui, par expérience, sait que rien n'est jamais arrêté, indiscutable… Au contraire, toute démarche progressiste rendra à termes caducs les propos de cet ouvrage. Doutons donc, comme le préconisait le dessinateur-humoriste G. Wolinski qui, un jour qu'il lui était demandée la définition d'un c… répondit, après un temps de réflexion non négligeable dû pour beaucoup à l'incongruité de la question : « quelqu'un qui ne doute jamais ». Nous n'en sommes ici que plus confortés dans notre démarche.

Ce livre est-il fait pour vous ?

Lecteurs, vous êtes des industriels du logiciel (ingénieurs, développeurs, architectes et chefs de projet) qui savez depuis longtemps que créer des logiciels avec l'aide quasi exclusive de générateurs d'interfaces utilisateur graphiques (*interface builder* en anglais) est la pire des pratiques. Dans le cas contraire, félicitations, vous savez produire du logiciel dont le taux de réutilisation est médiocre et le coût de maintenance exorbitant. Félicitations, car si vos clients continuent à ne pas broncher et payer cher quand il faut faire évoluer ou adapter les logiciels que vous leur avez vendus, vous n'avez peut-être pas atteint l'excellence technique mais bien l'économique, laquelle est bien sûr la finalité de toute industrie. Mais jusqu'à quand ? En effet, la délocalisation du développement informatique prend son envol et seule une ingénierie de tout premier plan permettra d'accompagner ce phénomène durant les années qui viennent. Ainsi, soit vous fermez cet ouvrage, et tant pis pour vous, vos malheurs ne sont pas les miens, soit vous vous laissez tenter par la modélisation

informatique et UML. Avec un regard critique et vous changez pour le meilleur, je l'espère, mais peut-être aussi pour le pire si vous n'y prenez pas garde. UML ne garantit pas de réussir, loin s'en faut. En ce sens, cet ouvrage existe pour vous éviter les déboires possibles et les pièges certains du langage de modélisation UML, même si le nouveau paradigme de l'ingénierie des modèles (MDE, *Model-Driven Engineering* en anglais) améliore *a priori* les choses.

Lecteurs, vous êtes aussi des étudiants en écoles d'ingénieurs et masters informatiques (réforme LMD). Prenez et manipulez tout le code relatif aux études de cas de cet ouvrage, code téléchargeable sur Internet. En effet, vous qui croyez qu'UML n'est finalement que « très joli » et « très conceptuel » (et en effet, il peut conduire à faire du Picasso), vous vous rendrez compte qu'il peut être très bénéfique et, avec une pratique intensive, très concret : faire du « vrai » génie logiciel et donc de la qualité n'est pas antinomique de la modélisation informatique. Même si à fortes doses, l'abstraction, c'est franchement rebutant. N'étant pas différent de vous, lecteurs, j'essaie dans cet ouvrage d'aller jusqu'au bout de l'implémentation pour combler cette attente souvent lancinante, exprimée dans les cours sur UML, de voir « des choses tourner ». C'est une première approche à adopter si finalement, au contraire de beaucoup d'enseignements d'aujourd'hui, vous préférez aller de l'expérimentation à la théorie. Mais n'oubliez jamais que la théorie reste fondamentale. Mes passages répétés dans l'industrie, en conseil ou en formation, m'ont souvent confronté à des problèmes « usine à gaz », incompréhensibles (leur présentation, souvent délibérément, visait à cela), qui se ramenaient à des algorithmes de la théorie de graphes connus depuis belle lurette. Encore faut-il savoir où trouver la solution. Comment adapter ces résultats connus à un cas concret ? Etc. C'est le travail de l'ingénieur de faire ce lien.

Kesako UML

Qu'est-ce qu'UML en vérité ? Quelle est sa finalité ? Imaginez tous les constructeurs automobiles se réunissant pour produire le même modèle de voiture. Ce serait économiquement irréaliste mais en poursuivant notre délire, que se passerait-il ? Chacun, du fait de la spécificité de sa clientèle, demanderait une « substitutabilité » (néologisme signifiant « un potentiel à la substitution ») des pièces, laquelle irait peut-être jusqu'aux châssis et pièces maîtresses. Même si des constructeurs automobiles partagent de nos jours des éléments communs en conception et en fabrication, leur originalité propre demande des méthodes, des outils et des composants personnalisés. « Tout partager », les constructeurs automobiles ne l'ont pas fait, les informaticiens l'ont fait, ou presque, pour la modélisation des systèmes logiciels, d'où le *M* pour *modeling* et surtout le *U* pour *unified* (unifié). Le résultat final est qu'UML est tellement ouvert (donc bancal, mal fini…), flexible (donc imprécis, ambigu…), que sais-je encore, qu'il n'existe pour ainsi dire pas : le cerner et le synthétiser est quasi impossible. En d'autres termes, il fait tout et donc il ne fait rien. Tout le monde a vérifié en pratique cet adage dans d'autres domaines.

Prenez maintenant un langage de programmation, vous le mettez en œuvre, vous codez, vous compilez, vous exécutez. Prenez UML, vous ne pouvez pas le mettre en

œuvre. Cela commence plutôt mal… Comme la fable des constructeurs automobiles, qu'allez-vous devoir changer, adapter, spécifier à partir des composants de base (au niveau méta, c'est-à-dire dans le cœur du langage UML lui-même) avant de bâtir vos premiers modèles ? Accepteriez-vous d'ailleurs d'adopter un langage de programmation et qu'il vous faille définir la sémantique de ce langage et éventuellement écrire le compilateur avant de pouvoir l'utiliser ? C'est pourtant ce que vous devez faire avec UML et cet ouvrage se propose de vous aider dans cette tâche. N'ayez pas trop de craintes, c'est possible…

French versus English

J'ai choisi de dessiner tous les modèles UML en anglais. UML est un langage dont les mots-clefs viennent de l'anglo-américain. Traduire ces mots-clefs en français m'a paru inadéquat, car les personnes qui utilisent les outils UML n'ont pas d'aide en français à leur disposition ; et à ma connaissance, tout le monde conçoit des modèles UML en gardant la terminologie originale. Par exemple, on écrit le stéréotype *«implementation class»* dans son modèle plutôt que «classe d'implantation» ou «classe d'implémentation». Par ailleurs, traiter une étude de cas revient à créer de nouveaux littéraux venant s'intégrer dans les modèles. Par exemple, si je choisis que « distributeur automatique bancaire » est une classe, je dessine la boîte correspondante en écrivant comme nom de classe « distributeur automatique bancaire » ou « DAB ». Le résultat final est un imbroglio de français et d'anglo-américain dans les modèles. Je préfère donc nommer ma classe *ATM* ou *Automated Teller Machine* pour garantir l'uniformité. Dans la mesure du possible, pour des modèles complexes, j'ai donc préféré tout rédiger en anglais à l'exception de l'étude de cas du chapitre 4, qui n'est pas ma propriété.

Méthode de lecture et d'exploitation préconisée

Selon l'esprit scientifique évoqué dans le premier paragraphe de cet avant-propos, cet ouvrage ne contient pas un discours simpliste sur UML 2. Ce type d'approche conduit la plupart du temps à un livre de recettes de cuisine (le terme *cookbook* est d'ailleurs souvent utilisé par les Anglo-saxons pour qualifier le contenu de tels ouvrages). En d'autres termes, ce n'est pas un guide méthodologique où, « sorties du chapeau », on propose des règles et des phases immuables qui donnent un cadre canonique d'utilisation d'UML. Au contraire, ce livre dissèque UML au risque parfois de se retrouver dans des quasi-impasses quant à son usage au quotidien. Ne voulant pas laisser le lecteur dans l'expectative et le flou le plus complet, l'ouvrage offre des repères clefs pour aider au mieux le lecteur dans sa propre réflexion sur UML. Le premier type de repère est le sigle « UML 2.x ». Chaque fois qu'il apparaît, cela signifie que le texte insiste fortement sur la différence entre UML 1.x et UML 2.x. Au contraire de beaucoup d'ouvrages actuels, j'ai vraiment voulu apporter un plus concernant la découverte d'UML 2.x et sa différentiation avec UML 1.x. Des marquages « L'essentiel, ce qu'il faut retenir » et « Le danger, ce dont il faut être conscient » sont également utilisés pour assister le lecteur dans sa synthèse et lui permettre de bâtir « son » UML, en fonction de ses goûts, besoins, exigences ou contraintes.

Merci beaucoup

Un ouvrage n'est jamais le résultat du travail d'une seule personne. Il y a les re-lecteurs désintéressés qui n'ont pas hésité à sacrifier leur temps précieux pour un de leurs collègues de travail. Ce sont, par ordre alphabétique : Philippe Aniorté, Nicolas Belloir, Marie-Noëlle Bessagnet, Jean-Michel Bruel, Pierre Laforcade, Franck Luthon, Christophe Marquesuzaà, Fabien Romeo, Philippe Roose et Christian Sallaberry. Qu'ils en soient remerciés. Il y a les industriels qui en m'ayant accueilli au sein de leurs projets, m'ont permis de développer et surtout de fortifier les idées de cet ouvrage. Qu'ils m'excusent de ne pas me rappeler le nom de tous mes interlocuteurs, qui font partie des sociétés suivantes : Aérospatiale (Bourges), Alcatel Business Systems (Illkirch et Colombes), Bull (Saint-Ouen), Cisco EMEA (Les Ulis), Crédit Agricole (Saint-Quentin-en-Yvelines), DCN (Lorient), Examéca (Serres-Castet), France Télécom SNAT (Bordeaux), France Télécom R&D (Issy-les-Moulineaux et Lannion), Héliantis (Pau), La Poste (Nantes), Prologue (Les Ulis) et Sercel (Nantes).

Cet ouvrage est aussi le fruit indirect des compétences et réflexions de deux grands spécialistes d'UML, Georges-Pierre Reich (coauteur d'*UML 1.1* à *l'OMG*, 1997 via la société Reich Technologies) et Joël Bayon, PDG actuels de, respectivement, Projexion Netsoft et Cogitec. Le loisir d'écrire un livre n'étant en général pas laissé aux personnes, tels les PDG, hautement surchargées de travail, j'espère qu'ils se reconnaîtront dans ces pages et auront l'impression qu'elles leur appartiennent un peu…

Franck Barbier
Pau, Pyrénées-Atlantiques

1

Modélisation objet

1.1 INTRODUCTION

Ce chapitre fait abondamment référence au livre de Bertrand Meyer intitulé *Object-Oriented Software Construction* [26]. Ne souhaitant pas réinventer « le fil à couper le beurre », il nous a paru incontournable et nécessaire de mettre en perspective, parfois sous un angle critique, la réflexion originelle de Meyer, sachant que tous les fondements et la philosophie du développement objet sont là, dans son livre. Néanmoins, ne voulant pas nous cantonner au langage de programmation Eiffel, nous avons dans un souci de variété donné nos exemples en Java et C++ pour l'essentiel. Nous conseillons donc avec ferveur et sans modération la lecture de ce livre essentiel pour qui veut devenir un spécialiste objet.

Cet ouvrage traite globalement d'UML, et donc foncièrement de modélisation objet, mais parler de modélisation objet ne doit pas être déconnecté du génie logiciel et des facteurs de qualité inhérents à cette approche « avancée » du développement informatique. Quel intérêt à faire un modèle UML, si ce n'est pour gagner en qualité globale ? Mais comment se mesure concrètement cette qualité dans les modèles ? Y a-t-il des modèles « non-qualité » ? Etc. Ainsi, bon nombre d'écrits oublient qu'un modèle UML est toujours censé être implémenté dans un langage de programmation objet, et qu'un modèle doit présenter et générer de la valeur ajoutée dans le processus de développement en général. C'est sous cet angle que nous appréhendons ce chapitre pour poser les bases techniques et culturelles qui guident et gouvernent cet ouvrage. Malheureusement, le discours usuel sur l'objet (réutilisation innée et directe, maintenance plus intuitive à moindre effort, fiabilité mieux garantie, entre autres) s'écroule quelquefois face à la réalité et la complexité des problèmes du développement.

Ce chapitre est aussi celui de la démystification doublée d'un peu de polémique et peut-être de provocation, du genre « *le modèle objet est un mauvais modèle* », avec une

dose supplémentaire de contradiction : « *l'objet est le formalisme aujourd'hui le plus approprié en génie logiciel* ». Le débat, contradictoire ou non, est à notre sens instructif mais aussi constructif et il nous a semblé qu'il en manquait dans les ouvrages existants sur UML en particulier, et sur la technologie objet en général : trop de consensus mou pour ne pas parler d'omerta, les défaillances du modèle objet étant peu discutées. Le lecteur sera ainsi parfois en opposition de vue avec nos idées mais nous pensons que nos tentatives de déstabilisation sont nécessaires pour, ensemble, progresser.

Ainsi ce chapitre, tout en rappelant brièvement l'histoire de l'objet, tente de positionner la modélisation objet et UML dans le contexte prégnant actuel du génie logiciel. Les trois premières sections s'attachent à mettre en lumière ce positionnement, et ce avant que la quatrième section ne traite des caractéristiques techniques natives du modèle objet (l'héritage par exemple) ainsi que de leur mise en œuvre dans les langages de programmation par objets et, plus succinctement, dans UML. La cinquième section s'applique à faire un tour d'horizon complet de tous les mécanismes connexes restants (la persistance par exemple). La sixième section aborde plus en détail l'analyse et la conception objet, et notamment le processus de développement, reconnu comme caractéristique et particulier à l'approche objet. La septième et dernière section se tourne vers le futur par une migration douce vers les composants logiciels et les paradigmes liés sur lesquels il y a une forte adhésion des informaticiens : patrons (*patterns*) et canevas d'application (*frameworks*). C'est donc la célèbre égalité de Johnson : « *Frameworks = (Components + Patterns)* » [22] qui clôt ce chapitre.

1.2 ÉLÉMENTS FONDATEURS DE LA MODÉLISATION OBJET

On peut avec raison s'interroger sur l'expression « modélisation objet ». Modélisation est déjà un terme tellement galvaudé qu'il faut s'en méfier. Nous ne prendrons pas le risque de nous y attarder car une encyclopédie n'y suffirait pas. Controverses, polémiques, chacun dans son domaine s'accapare ce mot « modélisation » sans jamais imaginer qu'une discipline connexe ou éloignée en a le même besoin, tout en lui donnant parfois une signification différente. Confronter les visions d'un informaticien, d'un mathématicien, d'un psychologue et d'un économiste par exemple, sur ce qu'est la modélisation est enrichissant car cela relativise l'usage, souvent appauvri, qui en est fait dans une science donnée. Cependant, une observation importante est que l'on peut trouver des écrits sur ce thème dès l'Antiquité, écrits dont l'acuité est aujourd'hui encore réelle. Nous conseillons ainsi l'un des plus récents ouvrages de Le Moigne [23], dont le travail a la réputation d'avoir grandement influencé l'ingénierie informatique hexagonale. Dans ses ouvrages, Le Moigne évoque, entre autres, des siècles de réflexion sur la modélisation, ce qui donne une synthèse intéressante quant aux emprunts faits par le génie logiciel. Pour nous informaticiens, une certaine humilité s'impose car bon nombre d'entre nous faisons un usage souvent

décalé, voire douteux, du terme « modélisation », en regard de toutes les réflexions et résultats scientifiques à ce jour disponibles. Cela va bien au-delà des sciences exactes en général et de la technologie en particulier.

1.2.1 Modélisation « non-objet »

La définition formelle, si elle existe, des mots « modèle » et « modélisation » dépasse le cadre de cet ouvrage. Nous supposons le lecteur averti et préférons donner une idée intuitive de « modélisation objet ». L'existence même de l'expression « modélisation objet » suppose alors qu'il y a de la modélisation, action de modéliser, qui *n'est pas objet*. Il devient alors intéressant et même pertinent de définir « modélisation objet » en définissant son contraire. Exemple de modélisation non-objet : une entreprise manufacturière a un problème d'optimisation de sa production. Si ce problème est formalisable à l'aide de la théorie des graphes par exemple, il est possible d'exploiter les résultats disponibles (algorithmes) pour résoudre les problèmes de cette entreprise. Autre exemple : une carte routière touristique de la France est un modèle du réseau routier français. Dernier exemple, un psychologue peut situer ses patients dans des classes de comportement (des modèles de dysfonctionnement : paranoïaque, histrionique…) de manière à poser un diagnostic global lors de l'analyse d'un comportement donné.

Dans ces trois exemples, appelons le premier un modèle mathématique, le deuxième un modèle géographique (même si une carte touristique routière comporte des données vectorielles, elle comporte aussi des informations mathématiquement non interprétables comme « site à visiter » par exemple, d'où notre qualification) et le dernier un modèle cognitif (le psychologue modélise par appariement aux connaissances qu'il a accumulées sur les individus). Pour terminer notre démonstration sur le caractère non-objet de ces modèles, il reste à préciser le sens que donne l'informatique au qualificatif « objet ».

1.2.2 Modélisation informatique

En informatique, écrire un programme c'est par essence fabriquer un modèle puisque l'on donne à ingurgiter à un ordinateur un texte qu'il traduit par une suite de 0 et de 1. Qu'est-ce qui caractérise ce texte ? C'est la mise en exergue de propriétés (*e.g.* le domaine de valeurs d'une variable appelée *var* est l'ensemble R des réels) au détriment d'autres, voire à leur occultation délibérée (*e.g.* le format ou code qui permet de représenter *var* dans l'ordinateur). Ce sont ces facultés d'abstraction supportées par le langage de programmation utilisé qui assurent qu'il s'agit de modélisation.

Mais il est difficile d'ignorer que le domaine de valeurs de *var* est sous contraintes. C'est plus exactement un sous-ensemble borné de l'ensemble R des réels qui est offert par l'ordinateur et donc qui pose des problèmes pour manipuler de très grandes valeurs. C'est la prise en charge de cette préoccupation récurrente qui importe à l'informaticien. Peut-il trouver un moyen sûr et simple de s'affranchir une fois pour toutes de ces difficultés ? En règle générale, il lui faut user d'astuces et de subterfuges

de programmation (détection de débordement de format ou autres). Ainsi, dans le langage C++ par exemple, l'expression *double var*; est source de souci. Supposons maintenant que l'informaticien dispose d'un artefact se présentant sous forme d'un type logiciel qui, par contrat, assure de pouvoir manipuler toutes les valeurs de l'ensemble R des réels, les très grandes spécialement. L'informaticien utilisateur de ce type acquiert alors un nouveau pouvoir d'abstraction lui permettant de ne manipuler que l'« interface » du type. Interface par opposition à implémentation[1] ou aussi représentation (le format machine utilisé) : c'est le principe d'encapsulation illustré ci-dessous à travers la classe (ou le type) *R* écrit en C++.

```
// déclaration du type R :
class R {
private:                          // partie rendue inaccessible à l'extérieur
   vector<char> _implementation;// représentation machine via un type
                                  // prédéfini, ici vector
public:                           // partie destinée à l'utilisateur du type
   R(string s);                   // partie de l'interface : c'est via
                                  // une chaîne de caractères que l'on crée
                                  // de très grandes valeurs réelles
};
                                  // utilisation du type R :
R var("999999999999999");         // autant de chiffres que l'on veut...
```

Le code qui précède est caractéristique de l'approche objet. La modélisation objet se définit en partie par ce mécanisme d'encapsulation qui permet de volontairement ignorer ou cacher des propriétés qui, certes intéressantes, nuisent au programmeur utilisateur : dans sa réflexion, celui-ci doit privilégier « son » programme aux dépens de problèmes concernant l'ensemble R des réels, finalement très généraux, dont il serait souhaitable par ailleurs qu'il existe une solution canonique.

Nous ne distinguons pas pour le moment « programmation objet » de « modélisation objet », la première activité n'étant à notre sens qu'un cas particulier de la seconde.

L'essentiel, ce qu'il faut retenir

La modélisation objet est donc l'action de représenter des besoins, des situations et des systèmes logiciels selon un paradigme informatique incarné dans des langages de programmation comme Smalltalk [14], C++ [9], [27], [31], Eiffel [25], [26], Java [5], [15] ou plus récemment C# [4].

L'idée est de retenir des éléments de construction présents dans ces langages : encapsulation mais aussi dualité classe/instance, héritage, etc. Le développement d'une application, dans ses prémisses, ne doit pas être sujet à des problèmes de machine mais sujet au respect d'exigences métier, on peut imaginer manipuler cet « outil objet » pour découvrir (capturer), écrire (formaliser) et satisfaire (traduire

1. Nous utilisons indistinctement les mots *implémentation* (anglicisme passé dans les mœurs) et *implantation* tout au long de cet ouvrage.

progressivement) des contraintes de nature diverse : relatives à la logique applicative au début jusqu'aux aspects techniques informatiques les plus pointus à la fin, selon un continuum qui mène au code, aux tests et la mise en exploitation.

1.2.3 Objet ou structuré ?

Cependant, en quoi la modélisation objet s'impose-t-elle sur un autre paradigme de conception logicielle comme l'ancestrale approche dite structurée ou fonctionnelle[1] ? *Primo*, dans l'esprit génie logiciel qui est le nôtre, la qualité du logiciel, sous toutes ses formes, est et reste fondamentalement la raison d'être de la technologie objet. Ce point est très mal présenté et mis en exergue dans la littérature à l'exception de l'ouvrage de Meyer [26], bible s'il en est et démonstration éclatante du pouvoir de l'objet[2]. On ne saurait trop ici, encore une fois, encourager sa lecture. Rappelons cependant que les trois axes de la qualité du logiciel vu comme un produit industriel sont : réutilisabilité, maintenabilité et fiabilité. Nous y revenons évidemment dans ce chapitre. *Secundo*, et c'est l'essentiel, l'objectif primordial de la modélisation par objets est d'assurer une continuité, ou traçabilité, entre les différents plans de description des besoins, des situations initiales, des architectures, des micro-architectures, des composants[3] à la fin. L'élaboration du code n'est alors qu'une sorte d'aboutissement d'une telle démarche pour matérialiser le produit fini qu'est un logiciel. Imaginons un designer, un architecte et un maçon impliqués dans la même opération immobilière. Là où le designer utilise des éléments subjectifs pour séduire d'éventuels acheteurs de logements, l'architecte s'appuiera plus volontiers sur des calculs géométriques pour formaliser les propriétés du bâtiment en termes de solidité, de faisabilité, de sécurité, d'hygiène ou autre. Le maçon adoptera une démarche plus intuitive et empirique, liée à un long savoir-faire. Qu'est-ce qui prouve néanmoins que la subjectivité du designer aura une représentation dans la rigueur de description de l'architecte, elle-même pas forcément compatible avec l'approche heuristique du maçon ? Ce sont ces ruptures que la modélisation objet vise à anticiper en fédérant la fabrication de A à Z d'une application autour d'un unique paradigme de pensée : l'objet. Cette idée forte a été résumée dans l'ouvrage de Hutt [20, p. 161] comme suit : « *One of the strenghts of the object paradigm is that a common core of constructs supports modeling at all levels of abstraction – from strategic modeling of real-world objects to implementation of application classes in programming languages*. »

1. Nous utilisons ici le vocabulaire consacré. L'approche objet étant aussi structurée, opposer approche objet et approche structurée n'a de sens que si nous précisons que nous nous soumettons aux conventions de dénomination qui ont cours. Idem pour l'approche fonctionnelle.
2. Démonstration souvent contextualisée à Eiffel mais indiscutablement généralisable et convaincante.
3. L'expression *composant logiciel* ou *composant* est naturelle en approche objet et désigne l'objet (*i.e.* l'instance) ou le type. En fin de chapitre, nous en donnons néanmoins une définition plus « actuelle ».

1.2.4 Entre théorie et pratique

Pas d'illusion néanmoins, notre discours idéaliste est contrebalancé dans cet ouvrage par de nombreux cas pratiques qui montrent que modéliser par objets est compliqué, UML ou pas UML ! Le concept même d'objet peut légèrement varier d'un langage de programmation à l'autre. Par exemple, en Smalltalk ou Java, les classes sont aussi des objets (mécanisme de *reflection* en anglais) alors que cela est faux en C++. Considérer « l'objet » comme un paradigme universel, une structuration naturelle de la de pensée, est inapproprié. Une telle démarche bute en fait et avant tout sur l'absence d'une définition unique.

Au sein de la recherche scientifique, nombre de détracteurs se sont engouffrés dans la brèche pour dire que le modèle objet n'existait *de facto* et théoriquement pas, vu son manque d'assise mathématique. La contribution d'Abadi et Cardelli [1], connue comme la plus élaborée, reste à ce titre et à notre sens marginale par le faible écho qu'elle a eu sur le monde du logiciel objet. Du côté industriel, l'Object Management Group (OMG) au début des années quatre-vingt-dix a produit un cadre de définition pseudoformelle : l'*Object Management Architecture* ou OMA [28] sur laquelle toutes ses normes dont UML sont censées s'appuyer. C'est le bon sens commun et surtout la diffusion et la pratique conséquentes de la technologie objet qui rendent aujourd'hui le modèle objet incontournable.

1.2.5 Une définition du modèle objet

La technologie objet arrivée à maturité, une définition normalisée prévaut maintenant sur toutes celles qui figurent dans les articles et ouvrages sur les objets. Le *Core Object Model* de l'OMG [28, p 42] est le modèle objet minimal et originel proposé par cet organisme de normalisation pour fonder initialement CORBA (*Common Object Request Broker Architecture*). Les problématiques objet connexes comme celle d'analyse et de conception objet ne pouvant se satisfaire de ce modèle appauvri, des extensions (idée de *profile* reprise depuis par UML) ou des modèles objet « compatibles » cohabitent dans les autres technologies de l'OMG. C'est le cas du *Technical Framework for Object Analysis and Design* [20] présenté à la fin de ce chapitre. C'est aussi le cas pour UML qui via son métamodèle donne « une » vision du modèle objet, comme une extension, *a priori*, du *Core Object Model*.

Concrètement par exemple, le *Core Object Model* ne supporte pas le concept de relation alors qu'en analyse et conception, ou encore dans les systèmes de gestion de bases de données, cette notion est évidemment nécessaire. Le *Core Object Model* énonce et met ainsi en exergue les concepts de base suivants :

- objet et type d'objet ;
- identité et identifiant d'objet ;
- opération ;
- sous-typage et héritage ;
- implémentation ;
- ce qui est « non-objet ».

Notons que la notion de classe est occultée au bénéfice de celle de type. Il nous a semblé important de préciser le sens de tous ces concepts. À cet égard, un métamodèle UML aide à donner une vision rationnelle de concepts logiciels et surtout de leur interdépendance. Attention néanmoins au métamodèle de la figure 1.1 qui nous est propre et qui peut tout à fait être mis en balance avec le métamodèle d'UML 2 (chapitres 2 et 3). Le lecteur a la possibilité de vérifier si le métamodèle actuel d'UML 2 est bien une extension du métamodèle de la figure 1.1 — ou du moins lui est conforme.

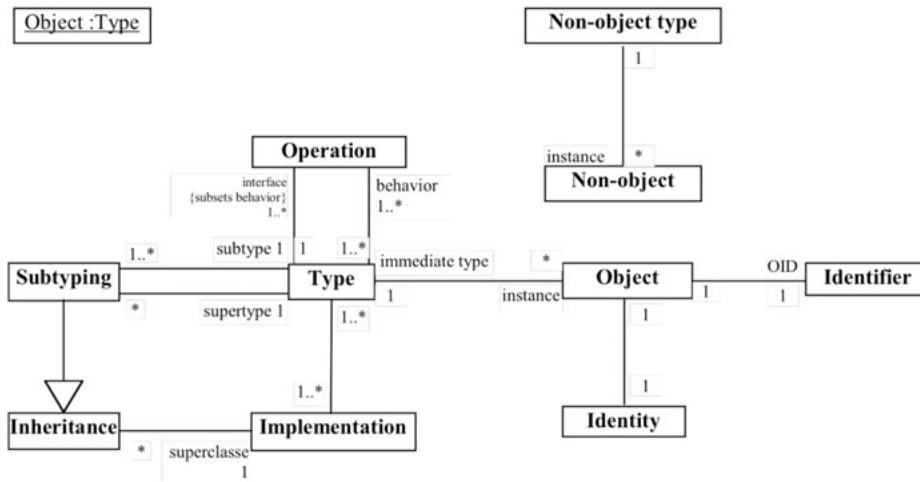


Figure 1.1 — Métamodèle UML exprimant l'esprit initial de l'OMG pour un modèle objet « noyau ».

Un objet a une identité (*Identity*) indépendante de ses caractéristiques et aussi un identifiant (*Identifier*) appelé OID qui est le moyen technique de « faire référence » à cet objet dans un système. Étonnamment, l'OMG ne met pas en rapport ces deux notions d'identité et d'identifiant : « *Each object has a unique identity that is distinct from and independent of any of its characteristics. Characteristics can vary over time whereas identity is constant.* » [28, p. 44].

La notion de type est affirmée par un ensemble d'opérations constituant son comportement (rôle *behavior* apparaissant au côté du métatype *Operation* en figure 1.1). Un type a plusieurs interfaces (au moins une) et plusieurs implémentations (au moins une). L'interface globale du type (ensemble des opérations déclarées) est un sous-ensemble des opérations formant le comportement (celles déclarées et celles héritées). L'objet est alors une représentation, instance, du type. Il n'y a pas de multi-instanciation au sens où le type immédiat (rôle *immediate type*) d'un objet est unique (cardinalité *1*), ceci n'empêchant en rien qu'une instance soit conforme à plusieurs types au sens du sous-typage et de la substitutabilité. À noter que le concept d'opération est plus développé dans la documentation que nous ne l'avons fait dans le modèle de la figure 1.1.

La relation de sous-typage (*Subtyping*) engendre un graphe orienté acyclique dont la racine est *Object*, qui est donc une instance du métatype *Type* (voir tout en haut, à gauche de la figure 1.1). Cette relation n'a d'implication que sur l'interface globale du type et donc exclut ses implémentations. C'est l'héritage (métatype *Inheritance* associé au métatype *Implementation*) qui sémantiquement désigne l'idée de *Subclassing* (voir section 1.4.2). La métarelation d'héritage entre *Subtyping* et *Inheritance* (triangle blanc en UML, figure 1.1) s'explique par un extrait de texte de [28] : « *The Core Object Model relates subtyping and inheritance. If* ***S*** *is declared to be a subtype of* ***T****, then* ***S*** *also inherits from* ***T****.* » Par ailleurs, les cardinalités vers *Subtyping* pour les deux métaassociations avec *Type* prouvent que le *Core Object Model* supporte l'héritage multiple (voir aussi section 1.4.3).

Pour gérer des types primitifs et améliorer la cohabitation entre objets et non-objets, Java et Eiffel, au contraire de C++, ont créé des passerelles (*int* et la classe *java.lang.Integer* en Java par exemple). Bien que limitées dans leur usage (on ne peut hériter de la classe *Integer* en Java alors que cela est possible en Eiffel), ces passerelles rendent la programmation plus élégante et surtout plus puissante (voir mécanisme de réflexion plus loin).L'idée de « non-objet », type et instance, n'est qu'une mauvaise influence des programmeurs C qui au moment de la fabrication du *Core Object Model* devaient bien caser quelque part leurs *short*, *int*, *long*, *float* et autre *double*. Ce choix est justifié comme suit : « *The Core Object Model has chosen to recognize that this distinction exists. Things that are not objects are called* non-objects*. Objects and non-objects collectively represent the set of denotable values in the Object Model.* » Cet esprit s'est décliné en modélisation dès l'origine dans OMT (*Object Modeling Technique*) où l'on pouvait lire : « *An attribute should be a pure data value, not an object. Unlike objects, pure data values do not have identity.* » [29, p. 23]. On trouve la même chose dans la figure 1.1 puisque le métatype *Non-object* n'est pas relié à *Identity*.

1.2.6 Origines

À notre connaissance, deux personnes, Meyer et Booch, ont avec justesse évité, ou en tout cas utilisé avec précaution, l'expression « programmation orientée objet » au profit de celle de « construction de logiciel orienté objet » pour le premier [26] et de celle de « développement orienté objet » pour le second [6].

Booch a en particulier posé la première pierre en montrant que l'organisation d'un logiciel sous forme de modules possédant des propriétés bien précises, constitue une rupture avec l'existant. L'expression « modélisation objet » est ainsi née de la nécessité d'une méthode incluant « un langage » graphique pour dessiner les modules, leurs relations et leurs interactions, ainsi qu'une démarche essentiellement constituée d'étapes à suivre pour le développeur. Les langages de programmation objet au sens large, deviennent alors des outils d'aide à la réalisation, leur spécificité ne devant pas influer sur la manière de concevoir. La conséquence directe est qu'un logiciel ainsi décrit doit pouvoir être décliné en C++, Eiffel, Java ou encore C#, et ce sans difficultés majeures. En d'autres termes, le développeur ne doit pas constater au moment de l'implémentation qu'il a perdu du temps à faire de la modélisation objet

du fait que le langage (de programmation) qu'il utilise a des particularités telles qu'il lui faut restructurer ses modèles, voire les abandonner car ils ne lui sont d'aucune utilité, sinon de documentation, ce qui est peu. Cet ouvrage insiste dans ses études de cas sur ce point en cherchant une transformation la plus aisée et la plus directe possible entre spécification et code.

Rumbaugh *et al.* ont ultérieurement assis et popularisé la modélisation objet en injectant dans OMT [29] toute la culture du moment en termes de paradigmes de modélisation et principes des langages à objets. Ce mélange, même s'il a été à notre sens maladroit, est à la source d'UML et plus généralement de la pseudorévolution qui a suivi concernant les méthodes d'analyse et de conception informatiques objet : une énumération (non exhaustive) et une intéressante comparaison apparaissent dans les ouvrages de Hutt [19] et [20].

1.2.7 Fédération des paradigmes de modélisation

On peut attribuer deux buts louables à la modélisation objet. Le premier est d'avoir essayé de combler le fossé entre formalismes de spécification et formalismes de programmation de manière à créer un continuum dans le processus de développement logiciel. Cela est lié à l'idée de processus de développement « sans couture » (*seamlessness*) évoqué précédemment. Le second but est d'avoir tenté de fédérer les paradigmes de modélisation populaires dans le monde du génie logiciel, notamment les trois grands axes : structurel, fonctionnel et comportemental (figure 1.2).

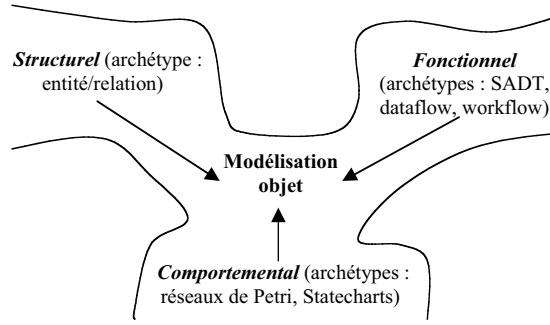


Figure 1.2 — La modélisation objet au confluent des trois grands axes historiques de modélisation en méthodologie de développement logiciel.

La modélisation objet est-elle une resucée ou une réelle innovation de la modélisation en technologie du logiciel ? En d'autres termes, le modèle objet est-il un mélange (harmonieux et homogène si possible) du modèle entité/relation, des automates à états finis, des formalismes fonctionnels avec flux d'entrées, flux de sorties, et processus de transformation de données et flux de contrôle, ou une véritable innovation, c'est-à-dire un paradigme de modélisation qui s'inspire certes d'autres, mais qui a ses propres éléments/valeurs ajoutées ?

La réponse est contrastée. Si l'on regarde la méthode OMT qui a fait foi tout au long des années quatre-vingt-dix, et maintenant UML, on a plus un agglomérat de techniques de modélisation préexistantes qu'un nouvel outil de modélisation. *A contrario*, si l'on revient aux sources c'est-à-dire aux langages de programmation objet, ces derniers ont réellement reformulé la manière de construire le logiciel. Ce qui fait dire à Hutt [20, p. 161-162] que la modélisation par objets apporte les concepts fondamentaux suivants, la différenciant des autres types de modélisation informatiques : *Abstraction*, *Encapsulation*, *Reuse*, *Specialization*, *Object communication* et *Polymorphism*. C'est parce que ces écrits émanent de l'OMG que nous y faisons référence. La réalité n'est pas si simple. Pour la réutilisation par exemple, l'erreur est souvent de croire que cette propriété du logiciel (sous forme de spécification ou de code) est intrinsèque (innée) au modèle objet lui-même. Par ailleurs, le modèle entité/relation favorise-t-il la réutilisation au même titre que le modèle objet ? Probablement non, mais l'exclut-il ? Certainement pas.

Un dernier constat est que le modèle objet d'UML, aujourd'hui assez répandu et par nature dédié à la spécification, n'a pas de cadre mathématique. Plus généralement, les langages de spécification formelle comme Object-Z (voir la section 1.10, « Webographie ») par exemple, n'ont pas réellement percé pas plus que la spécification formelle « non-objet » ne s'est imposée à l'approche structurée. Cette observation est à mettre en rapport avec les critiques faites à UML dans les chapitres 2 et 3 où le manque d'assises mathématiques pénalise globalement la qualité des modèles.

1.3 TENANTS ET ABOUTISSANTS DE LA MODÉLISATION PAR OBJETS

Le principe naïf de la modélisation par objets est de mimer par des structures de données des « entités » plus ou moins proches de l'entendement humain : des choses tangibles comme une facture, un bon de commande ou d'autres assez éloignées « du monde réel » comme un contexte d'exécution machine par exemple, qui peut être transcrit à l'aide d'une classe. L'expression « structure de données » étant en effet trop liée à la programmation, on lui préfère le terme « classe », terme consacré, ou celui de « type » ou encore « type abstrait » de données, ces deux derniers étant plus attachés à des considérations mathématiques. Selon le langage d'implémentation ou de spécification retenu, on peut parler de simples classes (au sens classes d'équivalence : deux instances sont reliées par le fait qu'elles sont issues de la même classe, par nature unique[1]) ou de types avec un pouvoir de programmation/modélisation plus grand : principe de substitution cher au sous-typage (voir section 1.4.2).

1. En génie logiciel, contrairement à l'intelligence artificielle, l'approche objet évite la possibilité de multi-instanciation (voir aussi le *Core Object Model* de l'OMG) — mécanisme présent dans certains langages de programmation plus expérimentaux qu'industriels.

Avant d'étudier les moyens techniques précis et prouvés efficaces qu'offre la technologie objet, la question simple est : en quoi la qualité du logiciel augmente avec l'approche objet ? L'appréhension de la complexité du logiciel est favorisée par une meilleure modularité induite par le modèle objet. Le modèle objet accroît la fiabilité (résistance aux pannes), la maintenabilité (évolution des besoins, réparation des erreurs) et la réutilisabilité (productivité, rationalisation et capitalisation) des programmes. Finalement, la traçabilité est sous-jacente au modèle objet lui-même présent sur tous les plans de description du logiciel.

1.3.1 Complexité du logiciel

Dire qu'un logiciel est complexe à concevoir ou à maintenir est aujourd'hui souvent un lieu commun, une lapalissade. La modélisation objet comme moyen d'affronter cette complexité n'est pas la panacée mais un mal nécessaire. L'approche objet n'est qu'une variation du précepte cartésien bien connu : « Diviser chacune des difficultés que j'examinerais en autant de parcelles qu'il se pourrait et qu'il serait requis pour mieux les résoudre. » Si nous avons fait référence aux travaux de Le Moigne au début de ce chapitre, c'est parce qu'il est utile de rappeler que ce dernier est un ardent détracteur de Descartes (« Diviser pour régner, une méthode inutile et incertaine mais prégnante. » [23]) et aussi, ne l'oublions pas, de l'approche objet.

La modélisation objet est avant tout fondée sur le pragmatisme (à l'anglo-saxonne). L'approche systémique censée pallier les défauts de l'approche « analytique » n'a jamais vraiment trouvé sa représentation dans les « méthodes » de développement du logiciel, même si Merise et ses adaptations/extensions objet le revendiquent. De façon caricaturale, la question est : l'objet est-il l'unité de décomposition/composition modulaire la plus appropriée ? On entend par là le fait que l'optimum d'un problème n'est jamais la somme des optima locaux. Autrement dit, décomposer un programme en objets et/ou l'assembler en réutilisant des objets n'assurera jamais que ce logiciel se conformera parfaitement et immédiatement aux attentes qui ont présidé à sa construction. Ainsi, il est vrai que la modélisation objet peut sous certaines considérations, épistémologiques en l'occurrence, être considérée comme rétrograde.

Là s'arrête néanmoins le discours philosophique car il faut bien des outils pour construire des applications. Disons par analogie avec Churchill que la modélisation objet, au même titre que la démocratie est le moins pire des systèmes politiques, est la moins pire des technologies de conception logicielle. Ce regard critique nous paraît ici essentiel.

Le danger, ce dont il faut être conscient

Encore une fois, cette critique permet de relativiser la portée d'UML comme langage « standard » de modélisation ; standard industriel certes mais certainement pas standard scientifique. Les imperfections d'UML perceptibles dans les modèles sont donc peut-être propres à l'approche objet elle-même, mal inspirée de la théorie

systémique ? Au-delà, notre bifurcation enthousiaste et opportuniste vers les composants logiciels en fin de ce chapitre et dans cet ouvrage, doit aussi être tempérée : l'ingénierie du logiciel basé composant est-elle une meilleure déclinaison de la théorie systémique que l'ingénierie objet ?

1.3.2 L'objet comme unité de décomposition/composition

Dans la courte polémique qui précède, nous avons parlé de « mal nécessaire » parce que le principe de découpage (ou décomposition) inhérent à l'approche cartésienne est « mauvais ». Par ailleurs, l'approche objet n'est pas uniquement fondée sur la décomposition mais aussi sur l'assemblage (ou composition) tel que Meyer le défend et le préconise dans [26]. En technologie des composants, ce sont plus récemment les néologismes « composabilité » et « compositionalité » qui dominent pour caractériser et qualifier le composant logiciel comme une entité dont le *potentiel*, l'aptitude ou encore la faculté à être composé sont forts, ou en tout cas, plus forts que pour l'objet.

Revenons aux fondamentaux. La loi de Miller dit que le cerveau peut prendre en charge 7 ± 2 problèmes simultanément d'où l'obligation pour un développeur de scinder son logiciel en sous-ensembles, et ce récursivement. La complexité des exigences (besoins applicatifs) et des contraintes à satisfaire ajoutée à la complexité des moyens (langages de programmation, systèmes d'exploitation…) à mettre en œuvre (maîtrise) imposent l'isolement de problèmes bien bornés, et dont le contenu peut être directement résolu : fin de la segmentation.

En premier lieu, un mauvais découpage peut aboutir à la fabrication d'un module (une classe probablement) incluant quelques fonctions/procédures, quelques algorithmes, quelques contrôles. Cela revient à réaliser en parallèle plusieurs tâches intellectuelles jusqu'à risquer la saturation :

- initialiser certaines variables avant la boucle *for* ;
- initialiser d'autres variables au début de la boucle *for* ;
- s'assurer que la connexion réseau est ouverte (*i.e.* en état de marche) ;
- …
- 8e réflexion à mener, cela se gâte car on est au-delà du 7 de Miller ;
- 9e chose à ne pas oublier. À moins d'être un génie, on va atteindre les frontières de notre capacité à raisonner, d'où les bogues à venir…

Deuxièmement, la synthèse des solutions aux problèmes élémentaires, qu'elle soit faite collectivement (équipe de développeurs) ou individuellement, est vouée à l'échec. L'existence même de l'activité d'intégration en développement logiciel et sa criticité avérée le démontrent par la pratique : si l'objet était l'unité de décomposition/composition parfaite, l'activité d'intégration se limiterait à rien. Qui l'a vérifié ? Personne.

En troisième et dernier lieu, si vous avez finalement réussi tant bien que mal à assembler tous vos modules et qu'ils sont de qualité médiocre, il faut savoir que la qualité globale de votre application n'est pas une fonction linéaire mais une fonction exponentielle des indicateurs qualité (s'ils existent) des modules. Réutiliser certes mais réutiliser des modules *a priori* défaillants est une catastrophe assurée. En avertissement donc, attention à tous les modèles concrets et réels de cet ouvrage : leur nature objet profonde n'est gage d'aucune qualité intrinsèque. C'est l'approche objet adossée à une longue expérience et un savoir-faire chèrement acquis, avec ou sans UML, qui est la clé du succès.

Dégénérescence modulaire en « non-objet »

Le sombre tableau que nous avons dressé jusqu'à présent pourrait amener à la conclusion précipitée que « technologie objet = mauvaise technologie ». En vérité, nous pouvons dire « approche objet = meilleure approche que approche structurée/fonctionnelle ». Booch dans le chapitre 2 de son ouvrage [7, p. 27-80] intitulé « *The Object Model* » montre en quoi les topologies (ou architectures) de programmes non-objet se délitent au cours du temps, abaissant drastiquement la qualité globale des programmes.

L'essentiel, ce qu'il faut retenir

Schématiquement, l'approche structurée privilégie la fonction comme moyen d'organisation du logiciel. Ce n'est pas pour cette raison que l'approche objet est une approche « non fonctionnelle ». Les services des objets sont en effet des fonctions. Ce qui différencie sur le fond approche objet et approche structurée, c'est que les fonctions obtenues à l'issue de la mise en œuvre de l'une ou l'autre des approches sont distinctes. L'approche objet est une approche « drivée » donnée, aussi appelée « orientée donnée » : les fonctions membres des classes se déduisent d'un regroupement de champs de données formant une entité cohérente, logique, tangible et surtout stable quant au problème traité. L'approche fonctionnelle classique privilégie une organisation des données postérieure à la découverte des grandes, puis petites fonctions qui les décomposent, l'ensemble constituant les services qui répondent aux besoins.

C'est expérimentalement et *en moyenne* que l'on observe une meilleure stabilité des objets que des fonctions. En approche objet, l'évolution des besoins aura le plus souvent tendance à se présenter comme un changement de l'interaction des objets alors qu'en approche structurée, la topologie typique de la figure 1.3 dégénère car la complexification des unités de traitements (du programme principal aux programmes et sous-programmes élémentaires) se calque sur celle des besoins. Les unités de décomposition initiales, comme à gauche sur la figure 1.3, établies sur la base d'un ensemble borné et ciblé de besoins, changent radicalement. Une fonction élémentaire croît en nombre d'arguments par exemple. Une unité de traitement accroît le nombre de types de données et leurs instances qu'elle manipule, types décrits de

manière éparpillée dans les fonctions, et instances déclarées globalement pour ouvrir leur accès.

Ces altérations président à la dégradation de modules *initialement bien formés* : de moins en moins réutilisables car de plus en plus spécifiques à l'application, maintenables à coût prohibitif car leur contenu se complique rendant leur compréhension et mise en perspective à l'application de moins en moins évidentes, et finalement, de plus en plus générateurs d'erreurs car sans discipline dans la gestion de l'ensemble des données de l'application.

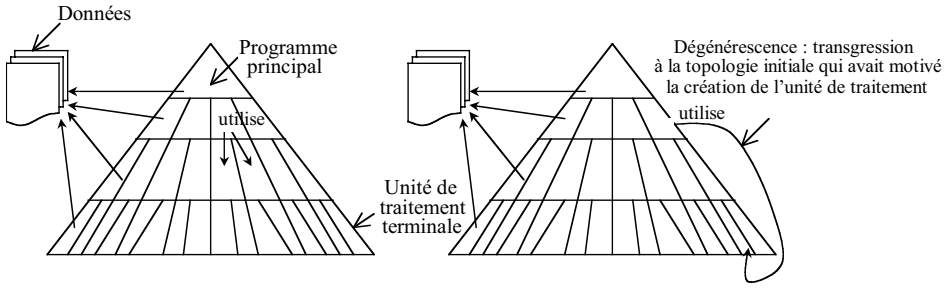


Figure 1.3 — Topologie initiale de programme en approche structurée et sa dégénérescence.

Une illustration est en approche objet une classe *Time-Date* qui regroupe et enferme le(s) format(s) nécessaire(s) à la gestion du temps et offre toutes les fonctionnalités de manipulation (conversion dans des formats imprimables, gestion d'intervalles de temps, etc.). En approche structurée, l'éparpillement de déclaration des formats, variables et calculs temporels vient du fait que la méthode de conception focalise sur les fonctions applicatives, plusieurs certainement traitant de problèmes de datation. Même si la notion de bibliothèque n'est pas étrangère à l'approche structurée (réutilisation de fonctions élémentaires comme la bibliothèque *time.h* de C par exemple), les calculs nouveaux à développer (calcul d'année bissextile ou autres) ont de fortes chances d'être dilués.

L'essentiel, ce qu'il faut retenir

La modularité n'est pas antinomique de l'approche structurée. Les modules résultant de la décomposition objet sont tout simplement différents de ceux émanant de l'approche structurée. Les unités de traitement et surtout leur dépendance dans la topologie de la figure 1.3 sont initialement probablement bons. C'est leur résistance au temps, contrairement aux modules objet, qui est source de déboires.

Modularité et approche objet

La modularité est un *moyen* de la qualité logicielle mais pas un objectif de premier plan. Si la réutilisabilité est un objectif de premier plan (objectif économique en

fait) en développement objet, la modularité est un outil de la réutilisabilité. Cette distinction objectif/moyen est primordiale.

L'intérêt de l'objet en tant qu'unité et support de décomposition/composition est qu'il est un outil qui engendre des modules ayant des propriétés remarquables, ou en tout cas « meilleures » que celles des modules qu'on obtenait avant l'avènement de l'objet. En d'autres termes, on peut par un autre paradigme de décomposition/composition logicielle aboutir à des modules. Par exemple, les unités de traitement Cobol appelées via l'instruction *perform* sont des modules. Néanmoins, ces modules ont, du point de vue du génie logiciel et en Java par exemple, de piètres qualités en regard des classes, des packages ou encore au niveau *runtime*, des unités de déploiement (*e.g.* les fichiers *.jar*).

La caractérisation de la modularité en approche objet se décline selon Meyer sous cinq formes [26] :

- Décomposabilité modulaire : « *A software construction method satisfies Modular Decomposability if it helps in the task of decomposing a software problem into a small number of less complex subproblems, connected by a simple structure, and independent enough to allow further work to proceed separately on each of them.* »
- Composabilité modulaire : « *A method satisfies Modular Composability if it favors the production of software elements which may then be freely combined with each other to produce new systems, possibily in an environment quite different from the one in which they were initially developed.* »
- Compréhensibilité modulaire : « *A method favors Modular Understandability if it helps produce software in which a human reader can understand each module without having to know the others, or, at worst, by having to examine only a few of the others.* »
- Protection modulaire : « *A method satisfies Modular Protection if it yields architectures in which the effect of an abnormal condition occurring at run time in a module will remain confined to that module, or at worst will only propagate to a few neighboring modules.* »continuité modulaire : « *A method satisfies Modular Continuity if, in the software architectures that it yields, a small change in a problem specification will trigger a change of just one module, or a small number of modules.* »

La décomposabilité modulaire est une reprise maladroite de Descartes. Notons juste que l'expression « *proceed separately* » anticipe la synthèse. On suppose en l'occurrence que résoudre séparément les problèmes équivaut à résoudre le problème dans sa globalité, ce qui est faux selon toutes les observations.

La composabilité modulaire est intéressante par le fait qu'elle apparaît comme l'assise de la réutilisabilité. La partie de phrase « (…) *in an environment quite different from the one in which they were initially developed.* » force le trait sur la fabrication de modules adaptatifs qui peuvent donc s'intégrer dans des systèmes ou avec d'autres composants *a priori* « assez » méconnus : l'usage de l'adverbe « *quite* » ici d'ailleurs plutôt opportuniste, reste néanmoins approprié à la réalité des situations.

La compréhensibilité modulaire elle, est le tremplin de la réutilisabilité et de la maintenabilité. En effet, la maîtrise intellectuelle d'une entité logicielle est vecteur de sa réutilisation par le fait qu'elle lève la méfiance naturelle (et saine) de l'utilisateur. Par ailleurs, plus un programme est aisé à comprendre, que ce soit un module individuellement ou une interaction de modules, plus il est sûr et évident que la maintenance est facilitée.

La continuité modulaire reprend ou imite les principes de forte cohésion et faible couplage chers à Booch, à savoir que l'étude d'impact d'un changement (besoins nouveaux) est courte et aisée : l'endroit où faire les modifications est *rapide à trouver*[1], les dépendances du module sont préférablement réduites ou au pire bien établies pour propager, en toute éventualité, le changement sur d'autres modules. Idéalement, toute la logique applicative associée au changement est « ramassée » ou concentrée dans un seul module, voire peu de modules environnants.

La protection modulaire enfin, touche la fiabilité en s'appuyant comme auparavant sur la forte cohésion et le faible couplage pour confiner les comportements anormaux de tout le logiciel, à des modules bien localisés ou à défaut à des petites grappes de modules dont l'interaction est la source du dysfonctionnement détecté au niveau global.

Le danger, ce dont il faut être conscient

Cette vision idéaliste doit dans la pratique être plus nuancée. Ce dont on est sûr, c'est que l'approche traditionnelle structurée supporte mal ces préceptes pour la meilleure des modularités. En réalité, l'effort et donc le coût pour qu'une classe vérifie conjointement les cinq règles qui précèdent peut être jugé exorbitant.

Prenons à titre d'illustration le cas de la composabilité, il n'est pas forcément utile de systématiquement fabriquer des modules compositionnels. Ainsi, un module implémentant un arbre rouge et noir par exemple, et les algorithmes usuels associés à cette structure de données bien connue, est un élément logiciel qui peut être intégré dans une grande variété d'applications. Dans la bibliothèque C++ STL [27] par exemple, le type de collection *set* (ensemble) est construit sur la base d'un arbre rouge et noir. Alors, effectivement, la classe *rb_tree* peut par ailleurs être réutilisée pour un tout autre objectif : un développeur peut ainsi décider que la classe *rb_tree* est un outil adéquat pour solutionner tout ou une partie de son problème : ordonnancement de tâches, gestion de priorités, etc. Nous voyons alors que *rb_tree* a été initialement construit pour proposer une version logicielle du concept d'ensemble, mais est finalement utilisable dans un environnement différent. Cette conséquence n'est cependant due qu'au caractère universel de cette structure de données. Dès qu'un module est plus spécifique à un domaine donné, sa faculté à être combiné, et

1. Cela indépendamment de la tâche elle-même pour effectuer les modifications. Peu sont conscients que la localisation d'une erreur est un problème souvent plus important que sa correction proprement dite.

surtout son aptitude à être réutilisé « dans un environnement assez différent de celui pour lequel il a été initialement développé » décroît de manière drastique. Il n'y a qu'à imaginer en quoi la classe *Chess_game* (jeu d'échecs) par exemple, peut être mise en œuvre dans de nombreuses applications ! C'est illusoire et c'est même un non-sens car *Chess_game* ne répond, par essence, qu'à un seul besoin.

La puissance d'une méthode de développement par objets est donc liée à des règles de la méthode qui permettent de distinguer les modules fortement compositionnels des modules qui le sont moins voire pas du tout.

1.3.3 Qualité du logiciel objet

Nous utilisons l'approche objet pour accroître la qualité du logiciel : c'est notre unique ligne directrice, c'est celle du génie logiciel. Principalement, la réutilisabilité, la recherche de la fiabilité, la maintenabilité et la traçabilité ne sont pour nous que des objectifs à finalité économique : produire vite et à moindre coût, maximiser la rentabilité. Les moyens de ces objectifs sont l'objet, c'est-à-dire « le » paradigme, qui par sa nature et ses atouts naturels et intrinsèques nous aide dans notre démarche. De façon détaillée maintenant, la modularité sous-jacente au modèle objet (section « Modularité et approche objet »), l'abstraction ou d'autres caractéristiques du modèle objet sont les moyens, et plus exactement les mécanismes techniques par lesquels on agit concrètement pour atteindre les objectifs précités. Ce chapitre met de côté la traçabilité plus volontiers explorée dans les chapitres 2 et 3. Pour les trois autres facteurs qualité, nous les identifions et commentons (voir les sections « Fiabilité », « Maintenance », et « Réutilisation ») avant d'en venir à des extraits de code et de modèles UML pour tenter de démontrer où et comment la recherche de la qualité s'opère (section 1.4).

Fiabilité

Présenter dans un style imagé la fiabilité du logiciel, c'est raconter la petite histoire qui suit. J'achète une voiture neuve (je la paye très cher). Deux jours plus tard, elle tombe en panne. J'invective le concessionnaire. Il s'excuse. Il répare (très vite) et me fait un cadeau (prochaine vidange gratuite).

J'achète un logiciel (j'aurais peut-être pu le trouver gratuitement). Une heure plus tard, il m'affiche des messages d'erreur. Je contacte le support technique (que j'ai dû payer par ailleurs). Il me dit que mon système d'exploitation est mal configuré et que j'ai mal installé et utilisé le logiciel. Je dois ajouter un « patch » ou mieux, j'ai la grande chance de pouvoir acheter la nouvelle version qui elle, fort heureusement, n'a plus le problème. Je l'achète et je suis content en plus !

Pourquoi accepter pour un logiciel ce qui est inacceptable pour une voiture ?

L'informatique moderne va bien au-delà de l'histoire banale de ce simple utilisateur. L'explosion d'Ariane 5 en vol (ah ! Si seulement on avait utilisé Eiffel [21]), les risques imaginés et évités lors du fameux passage à l'an 2000, et plus généralement le

logiciel omniprésent dans notre quotidien (ou *ubiquitous computing*) placent indéniablement la sûreté de fonctionnement comme critère phare de la qualité.

L'approche objet par le mécanisme de gestion des exceptions essentiellement, la programmation par contrats et l'isolation méthodique des éléments de programmes sous forme d'objets, a fourni un support innovant pour la fiabilité. Comme pour tous les critères de qualité logicielle, rien n'est cependant acquis, rien n'est gagné, beaucoup d'améliorations sont attendues. En clair, l'approche objet ne résout pas tous les problèmes et nombreuses sont les techniques annexes qu'il faut développer pour aller vers la sûreté de fonctionnement : méthodes de test propres aux systèmes objet par exemple. Ce chapitre s'attache néanmoins à démontrer qu'un niveau de robustesse correct est aisément atteignable dans les programmes objet. La modélisation objet est plus sujette à discussion, car il reste difficile d'entrecroiser préoccupations applicatives et contraintes de sécurité, sûreté ou autres. Hors du cadre de cet ouvrage, la programmation/modélisation par aspects est l'étape future des objets et la voie la plus prometteuse.

Maintenance

Inutile de disserter en profondeur à propos de la problématique de maintenance en développement logiciel : le lieu commun est que le développement proprement dit ne coûte rien en regard de la gestion de l'évolution des programmes, que ce soit sous forme de spécification ou de code. C'est la métaphore de l'iceberg : la partie visible est le développement alors que la partie immergée est la maintenance, *i.e.* le danger.

Lientz et Swanson ont montré dans les années quatre-vingt[1] à travers une étude sur la maintenance que cette dernière consiste pour 41,8 % en l'évolution naturelle et normale des besoins, le reste étant de la réparation selon le découpage qui suit :

- Réparation : 58,2 %
  - Modifications de formats de données (17,6 %)
  - Corrections d'erreurs urgentes (12,4 %)
  - Corrections d'erreurs non urgentes (9 %)
  - Modifications de matériel et couches logicielles intermédiaires (6,2 %)
  - Documentation (5,5 %)
  - Améliorations d'efficacité (4 %)
  - Autres (3,4 %)

Attention à ces chiffres qui concernent l'approche structurée. Ils ont néanmoins servi comme base de réflexion et d'amélioration lors de la genèse de l'approche objet. Ils nous intéressent tout au long de cet ouvrage pour montrer concrètement comment la gestion de la modification de formats de données par exemple, est favorisée par l'approche objet.

1. Les chiffres sont extraits de l'ouvrage de Meyer [26, p. 17]. Le lecteur attentif verra qu'il manque 0,1% dans le découpage de « Réparation » ; attribuons ce taux aux erreurs métaphysiques…

L'essentiel, ce qu'il faut retenir

Il faut concevoir objet non pas pour « être » mais pour changer, pour réparer. Un logiciel est un système vivant. Il se modifie et doit se modifier pour respecter de nouvelles exigences. Le terme « maintenabilité » est à lui seul intéressant, car il préfigure ce que doit être la conception objet : formaliser, écrire des choses qui ne se figent jamais. Intégrer à l'esprit qu'il faut sans cesse anticiper, c'est préparer, réfléchir et finalement créer la capacité d'un programme à être maintenu. Une vision de la maintenance comme l'une des dernières tâches du processus de développement logiciel est à ce titre inadéquate. En fait, en exagérant, développer du logiciel n'existe pas, seul maintenir du logiciel existe. En ingénierie des besoins (au moment de leur découverte, leur capture, leur consolidation…) par exemple, c'est-à-dire en amont même de la spécification, les fluctuations sont parfois si importantes que le logiciel que l'on construit dans la foulée, bouge perpétuellement dans son contenu pour suivre ces fluctuations. Autre cas : valider des besoins en prototypant un logiciel sur la base d'un démonstrateur (*e.g.* une interface utilisateur mimant l'interaction avec les usagers) engendre à coup sûr pour les demandeurs des questions, et donc des remises en cause, des adaptations (faisabilité technique et/ou économique), de nouvelles attentes. C'est sous cet angle qu'il nous paraît approprié, voire salutaire, d'aborder la maintenance en objet.

Réutilisation

La réutilisation, principe fondateur de la technologie objet s'il en est, est à notre sens un échec partiel de cette technologie. Beaucoup, en montrant que l'on pouvait écrire une pile (*stack*) une fois pour toutes et la réutiliser partout, ont caricaturé le problème, et surtout ont omis des facteurs non techniques.

Le premier est humain et est plus connu comme le syndrome du NIH ou *Not Invented Here*. En clair, un programmeur ne réutilise que le code qu'il a écrit. De prime abord, une telle réticence est regrettable. Sous un autre angle, réutiliser un composant « extérieur » rend prisonnier, ou dépendant au minimum. Mise à part la programmation par contrats chère à Eiffel, peu d'outils ont été développés pour instrumenter le processus de réutilisation : idée de composant à haute confiance. Imaginer ainsi de fortes contraintes de sécurité dans un logiciel, comment certifier que les composants réutilisés, individuellement et globalement par leurs interactions, ne brisent pas la politique globale de sécurité ? Ces inquiétudes louables sont évidemment un frein à tout type de réutilisation.

Le deuxième est métier. Réutiliser une pile, c'est immédiat et facile mais réutiliser des objets plus élaborés, plus fouillés se heurte à bon nombre de barrages culturels et techniques. On n'a jamais vu l'écriture unique et standard, partagée par tous, du concept de facture en Java par exemple. Pourquoi ? Parce que réellement personne ne partage la même perception sémantique de cette notion. Au sein d'une entreprise, un service comptable, un service commercial et un service achats auront des difficultés à s'accorder. L'idée d'objets métier, plus ou moins normés, n'a pas fait son chemin comme espéré.

Le troisième et dernier point d'achoppement de la réutilisation est économique : fabriquer un composant réutilisable est coûteux et les méthodes de mesure de logiciel manquent aujourd'hui encore de maturité pour établir quand et comment s'opèrent les retours sur investissement. La réutilisabilité est comparable à la maintenabilité au sens où elle doit être préfabriquée. Pourquoi alors s'astreindre à écrire un composant hautement adaptatif et/ou générique si son taux de réutilisation n'est pas garanti, économiquement en l'occurrence, par un retour sur investissement assuré ?

Heureusement[1], le sombre avenir de la réutilisation s'est estompé à l'arrivée des patrons de conception de Gamma *et al.* [13] puis ceux d'analyse de Fowler [12] donnant pour ces derniers, les assises de la réutilisation dès la spécification et donc appropriés et dédiés à des langages comme UML. Les canevas d'applications ont aussi dans une moindre mesure redynamisé la problématique de réutilisation, aujourd'hui surtout tirée par la technologie des composants logiciels (voir section 1.7).

Au quotidien, objets et composants standard inondent les bibliothèques, qui sont au cœur des langages de programmation. Tout programmeur Java, par exemple, réutilise les quelques milliers de classes et d'interfaces sans lesquelles il serait difficile d'écrire le moindre programme. Nous pensons en fait que la réutilisation a souffert d'un manque de recherche sur le processus de développement lui-même. Comment doit-elle s'y intégrer ? Comment la gérer ? Etc.

Autres facteurs qualité

Citons à titre complémentaire d'autres facteurs de qualité du logiciel dont le modèle objet ne possède pas à notre sens d'aptitudes particulières à leur prise en charge :

- l'efficacité au sens de la rapidité d'exécution et du faible encombrement des ressources ;
- la portabilité ;
- la facilité d'utilisation au sens de l'obtention d'un logiciel agréable et attrayant pour les utilisateurs ;
- la vérifiabilité au sens de la vérification, de la validation, du test… c'est-à-dire de l'aptitude d'un logiciel à être jaugé et contrôlé à n'importe quelle étape du processus de développement ;
- l'interopérabilité au sens de la compatibilité maximale des objets d'une même application ou d'applications différentes (là interviennent plus volontiers les composants logiciels) ;
- le développement rapide (RAD, *Rapid Application Development*, *timeliness*) qui n'est pour beaucoup qu'un corollaire de la réutilisation.

Nous laissons cette classification, probablement incomplète, à l'appréciation et à la critique du lecteur. En effet, même si Java par exemple intègre avec brio le facteur

1. Opinion nuancée par la suite…

portabilité, on ne peut pas généraliser pour la technologie objet tout entière. Autre cas : même si les normes d'interopérabilité les plus connues, comme CORBA de l'OMG, reposent sur le modèle objet, on peut considérer que les objets ne sont pas facilement et directement associables entre eux ou intégrables dans des environnements hétérogènes.

1.3.4 Démarche qualité et UML

Notre unique message est : UML n'a d'intérêt et de sens que s'il est utilisé en ayant toujours à l'esprit les préoccupations de la qualité. La spécification, rarement attrayante comme la programmation et souvent vue comme une tâche inutile et/ou superflue par les informaticiens, a souvent échoué par l'oubli des fondamentaux : si l'on spécifie avant de programmer c'est parce qu'il y a quelque chose à gagner. Dans le cas contraire, alors suivons la moyenne des comportements : pas de temps à perdre à « dessiner » !

La réalité du terrain nourrit notre réflexion et doit nous faire entrer dans un cercle vertueux où théorie (représentée par UML) et pratique doivent s'équilibrer et se marier harmonieusement. Ce qui suit est une histoire vraie. À l'issue de trois jours de formation/conseil sur OMT, à l'Aérospatiale pour ne pas la citer, un auditeur que je n'avais pas entendu des trois jours, au moment d'une synthèse finale, me dit : « Y'a des boîtes et y'a des flèches mais on ne voit pas bien ce qu'il y a dans les programmes. » Ce que j'ai initialement considéré comme une remarque d'un individu à l'intelligence quelque peu laborieuse, m'a plus tard ouvert les yeux sur la manière de faire adopter OMT, puis UML dans les entreprises : l'efficacité prouvée puis vérifiée par les utilisateurs de ce langage de modélisation. C'est le premier volet d'une démarche qualité en UML : il faut que ça marche !

Le second volet est plus stratégique par le fait que l'introduction d'UML à grande échelle, dans les grands groupes industriels ou les sociétés de services en particulier, dépasse le simple cocon des informaticiens. Il faut convaincre décideurs, utilisateurs finaux et bien entendu informaticiens. Voici une autre histoire vécue dans le groupe Bull, toujours au travers de formation/conseil, sur Merise cette fois, et les perspectives d'évolution (à l'époque, au début des années quatre-vingt-dix) vers l'objet. Un grand manager (non-informaticien) décidant des orientations stratégiques d'un gros service informatique interne à Bull, me dit : « Comment pouvez-vous me prouver que l'utilisation de Merise améliore la qualité globale de nos logiciels ? En d'autres termes, quand je lis un modèle Merise où d'ailleurs je ne comprends pas grand-chose, je ne vois pas comment je peux m'assurer dans ces dessins que les utilisateurs seront plus satisfaits en ayant au final en main un meilleur logiciel ! » La modernité aidant, remplaçons Merise par UML, réponse : « La satisfaction de vos utilisateurs est prédominante et prépondérante. L'accroissement de ces facteurs externes de qualité est tributaire de l'accroissement des facteurs internes de qualité. S'il existe des modèles UML clairs et précis, les informaticiens peuvent plus aisément et plus rapidement réparer ou adapter le logiciel à l'évolution des besoins, et par conséquent satisfaire les utilisateurs. Les facteurs internes comme la capacité du modèle UML et de son

code source associé à évoluer (maintenabilité) sont donc difficilement palpables pour un non-informaticien, mais croyez-moi, ils sont un moyen réel de vos objectifs. » Que l'informaticien perde de vue qu'au bout de la chaîne il y a l'utilisateur final, reste bien évidemment une erreur fatale en génie logiciel. *A contrario*, la nature technique du développement ne peut se satisfaire de pressions extérieures constantes fustigeant le visible sans avoir la moindre idée de la complexité de l'intérieur d'une application. UML est en mauvaise posture sur ce plan car apparaissant toujours comme le conceptuel en comparaison de l'opérationnel, c'est-à-dire ce qui tourne et qui génère de la recette. Adeptes de la modélisation objet et d'UML, la guerre continue pour vous.

1.4 MÉCANISMES NATIFS DU MODÈLE OBJET

Comment obtient-on concrètement en programmation par objets, et au-delà dans un formalisme comme UML, la qualité, ou comment les mécanismes d'abstraction/encapsulation, d'héritage et de polymorphisme, pour les principaux, garantissent cette qualité (tout du moins aident à sa recherche) ?

1.4.1 Abstraction

L'abstraction a deux formes en technologie objet. Ainsi, la nature même du modèle objet est un vecteur d'abstraction : une classe est un artefact (entité artificielle) qui imite, mime une entité « perçue », *i.e.* saisie par l'esprit. Abstraire et modéliser sont quasiment équivalents : on occulte délibérément des propriétés, *a priori* négligeables ou inutiles, pour mettre d'autres en exergue. L'âge d'un individu par exemple est pertinent dans un système d'information de gestion de permis de conduire alors que cette propriété sera accessoire dans d'autres types d'applications. En UML, le résultat de la modélisation est graphique (figure 1.4).

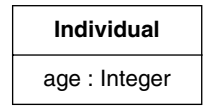


Figure 1.4 — Représentation UML de la propriété « âge » d'un individu.

Encapsulation

La seconde forme de l'abstraction découle du principe d'*information hiding* de Parnas, plus volontiers appelé « encapsulation », qui est la possibilité de cacher des données, mais au-delà, la structure détaillée de ces données ainsi que le comportement relatif au traitement de ces données. C'est une abstraction plus opérationnelle matérialisée par la dualité interface/implémentation ou encore partie publique/parte privée. Dans l'exemple Java qui suit, l'attribut caché *_birth_date* sert à mémoriser la date de nais-

sance. La fonction *Now()* est une opération interne (de classe) déterminant l'instant machine courant. L'idée est donc de masquer ces deux propriétés au contraire d'*age* qui est vue comme un service offert à l'environnement. Ce qui est occulté pour les objets clients qui interrogent la propriété *age* sur des instances de la classe *Individual*, c'est non seulement les deux propriétés citées mais aussi leur type, en l'occurrence ici les deux types Java prédéfinis *java.util.Calendar* et *java.util.GregorianCalendar*. La seule « difficulté » pour les programmes calculant un âge est donc de connaître la manipulation du type *int* puisque c'est le type de retour du service *age()*.

```
// implementation
   private java.util.Calendar _birth_date;
   private static java.util.Calendar Now() {
   return new java.util.GregorianCalendar();
  }
// interface
   public int age() {
    return Now().get(java.util.Calendar.YEAR)-
           ➥ _birth_date.get(java.util.Calendar.YEAR);
   }
```

Encore une fois, la métaphore de l'iceberg, est ainsi souvent utilisée pour caractériser l'encapsulation (figure 1.5). Dans sa logique profonde, il semble qu'il vaille mieux ne pas rencontrer la partie immergée au risque des pires désagréments. Pourquoi ?

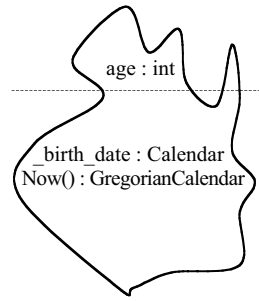


Figure 1.5 — Parties visible et immergée de l'iceberg en modélisation objet

Abstraction et maintenance

Selon les principes de continuité et de protection modulaires, les retouches inévitables du code doivent avoir un minimum d'impact, et ce en termes de tâches de maintenance additionnelles et de possibilités supplémentaires de panne. Par exemple, la modification du format des données doit se gérer au coût et au risque les plus faibles. Pratiquement, la mise en œuvre de l'interface (*int i = individual.age();*) ne doit pas bouger, la maintenance étant circonscrite à la classe *Individual* uniquement. De plus, cette utilisation du service *age()* ne doit pas accroître le nombre de bogues.

Pour le vérifier, plaçons-nous sur une période d'évolution normale et probable du code qui précède. Modéliser le temps a toujours fait appel dans les langages de pro-

grammation à des types standardisés. Les types *java.util.Calendar* et *java.util.GregorianCalendar* entrent en concurrence avec d'autres moyens de représenter des temps en Java, notamment *java.util.Date* ou encore *java.sql.Date*. Une version antérieure du précédent code aurait donc pu être :

```
// implementation
     private java.util.Date_birth_date;
```

En tant qu'utilisateur de la classe *Individual*, il n'y a pas d'intérêt à savoir que le calcul de l'âge s'opère à partir d'un type ou d'un autre puisque les exigences initiales ne faisaient apparaître que justement cette propriété « âge ». Placer dans la partie privée *_birth_date* et le format de cette donnée résultant de son type, évite les écueils. Parler de format est ici un peu exagéré car *java.util.Calendar*, *java.util.GregorianCalendar*, *java.util.Date* et *java.sql.Date* ne sont pas des formats machine, mais des classes Java sûres car elles-mêmes basées sur le paradigme objet. En d'autres termes, les moyens dont elles usent pour mémoriser le temps sont, pour assurer toute quiétude, aussi masqués.

L'absence de moyens d'abstraction a pour travers la manipulation anarchique des données et de leurs formats accroissant les taux d'erreur et compliquant les activités de maintenance. L'archétype des problèmes s'illustre facilement avec le langage C. En C, le type *time_t* de la bibliothèque *time.h* est historiquement le moyen le plus direct de coder un temps. L'accès permissif naturel à ce type en C conduit à savoir que ce type n'est qu'un *long* (grand entier) et code en millisecondes le temps écoulé depuis le 1er janvier 1970 à minuit pile. Il en résulte que l'exécution du programme suivant donne à l'écran, *Thu Jan 01 00:00:00 1970*.

```
#include <stdio.h>
#include <time.h>
void main() {
   time_t t = 0L; // on fait délibérément référence au type long de C
   printf(asctime(gmtime(&t)));
}
```

Abstraction et fiabilité

Du point de vue de la fiabilité, si l'on remplace *time_t t = 0L;* par *time_t t = -1L;*[1], le résultat est un arrêt brutal et anormal du programme. Pallier ce problème en C++, c'est enrober le type *time_t* par une classe dont l'usage est sécurisé, ou plus simplement réutiliser la classe *CTime* de la bibliothèque MFC (*Microsoft Foundation Classes*) par exemple. Cette classe supporte au contraire de *time_t*, l'addition et la soustraction, robustes, d'intervalles de temps (classe *CTimeSpan*).

Du point de vue de la maintenabilité, un changement du sens (le nombre de millisecondes écoulé depuis le 1er janvier 1970 minuit) du type *time_t*, est irréalisable et irréaliste : tous les programmes existant y faisant référence devraient être changés. En clair, le sens profond de *time_t* imprègne les programmes utilisateur créant un couplage extrême et donc une dépendance coûteuse et à haut risque.

1. Serait-on le 31 décembre 1969, 1 milliseconde avant minuit par hasard ?

Sur un autre plan, ajouter des millisecondes à une variable de type *time_t* n'est *a priori* pas sujet à erreur. Jusqu'à quelle limite temporelle néanmoins, le type *time_t* fonctionne-t-il correctement ? Le type *CTime* s'arrête en 2038, le type *COleDateTime* de MFC peut aller jusqu'en 9999. En résumé, le problème général est de s'affranchir au mieux des contraintes posées par les types primitifs et construits pour rendre le plus pérenne possible les programmes, et forcer ainsi la maintenabilité.

Pour l'humour, sachez qu'après le Y2K, il y a le Y5K. Le calendrier grégorien qui nous gouverne pose en effet à terme un problème : nous aurons environ un jour d'avance sur le soleil vers l'année 5000 (il faudra inventer le 30 février 5000 par exemple). Si l'on implante un calcul booléen déterminant si un instant donné est positionné dans une année bissextile, la règle actuelle (année divisible par 4 mais non séculaire à moins que divisible par 400) deviendra obsolète. Il est probable donc que le type *COleDateTime*, bien que certifié conforme par son constructeur, pour des datations jusqu'en 9999, ne passera pas le cap.

La solution est que l'on peut implanter de manière privée l'algorithme de test d'année bissextile sur la base de ce type jusqu'en 5000 tout en stabilisant l'interface. Par le mécanisme d'abstraction/encapsulation, il sera possible peu avant 5000 d'opérer une maintenance, plus facilement qu'on ne l'a fait pour le Y2K, pour faire face à ce handicap du calendrier grégorien.

Le danger, ce dont il faut être conscient

Plus sérieusement, le lecteur peut contester le caractère naïf de notre discours accentué par ce dernier exemple du Y5K. Revenant au présent, il a néanmoins tort, car il sous-estime le passage à grande échelle des développements logiciels « modernes » : contraintes industrielles et pressions économiques exacerbées, logiciels de taille parfois gigantesque, rigueur nécessaire du management et de l'organisation d'équipes pluridisciplinaires de recherche et de développement, etc. Tous ces facteurs environnementaux imposent de manière évidente l'application du principe d'abstraction/encapsulation dans les développements.

1.4.2 Héritage ou généralisation/spécialisation

Une structure de données, ou plus précisément une classe d'objets, doit pouvoir être décrite en fonction d'autres. En programmation objet, une structure de données est dotée d'un comportement et, selon le degré de sophistication du langage choisi, elle peut être qualifiée formellement de type (voir ci-après et aussi chapitre 2 concernant UML). L'extension et/ou la restriction du comportement d'une classe peut être « mécanisée » et systématisée à l'aide de l'héritage, en créant une nouvelle classe dont les propriétés s'expriment intimement en fonction de la classe héritée. Intuitivement, l'héritage favorise la réutilisation puisque l'on reprend l'existant.

Brièvement et schématiquement, la sémantique de l'héritage oscille entre le sous-classage (*Subclassing*) apparenté à l'héritage de structure ci-dessous, le sous-typage (*Subtyping*) assimilé à l'héritage de comportement et le lien conceptuel « est-

un(e) » (*Is-a*) dont le modèle entité/relation a forgé l'influence [33]. Tous trois correspondent à des sens différents de l'héritage, parfois mélangés, que ce soit dans les programmes ou dans les modèles UML. Nous étudions les deux premières déclinaisons de l'héritage dans ce chapitre et le lien (*is-a*) plutôt dans le chapitre 2. Rappelons que le *Core Object Model* de l'OMG supporte de manière privilégiée le sous-typage avec une tolérance pour le sous-classage.

Dans le code qui suit, la classe Java *Female* hérite (mot-clef Java *extends*) de la classe *Individual*. En effet, un postulat se cache derrière cette idée : le comportement d'une classe est donc, pour l'essentiel, repris pour éviter de coder des choses déjà codées. Il y a un phénomène d'appariement entre deux classes qui héritent l'une de l'autre. Une utilisation déplacée de l'héritage est alors de relier une première classe à une seconde qui s'accorde mal, c'est-à-dire dont les restrictions comportementales nécessitent un effort de programmation supérieur à sa création *ex nihilo*.

```
public class Female extends Individual {
  public Female(int day,int month,int year) {
     super(day,month,year);
  }
  // restriction : comportements hérités modifiés
  public int age() { // la gent féminine a une façon particulière
                     //de donner son âge…
     if(super.age() >= 20 && super.age() < 30) return 20;
     if(super.age() >= 30 && super.age() < 40) return 30;
     if(super.age() >= 40 && super.age() < 60) return 40;
     if(super.age() >= 60) return 50;
     return super.age();
  }
  // restriction : comportements hérités rendus concrets
  public boolean female() {
     return true;
  }
  public boolean male() {
     return false;
  }
  // extension : nouvelles fonctions inexistantes dans Individual
  public Male my_husband() {
     …
  }
}
```

Héritage de structure

Si l'on en revient à la classe *Individual*, on sait qu'elle dispose du champ *_birth_date*. L'héritage de structure fait donc que toute instance de *Female* a une occupation mémoire correspondant à cet attribut, inatteignable dans le code des fonctions de la classe *Female* par ailleurs, puisque déclaré *private* (pour permettre son accès, il doit être qualifié *protected* dans *Individual*). Il n'y a pas ici d'effet de bord indésirable car d'une part, le champ est invisible et d'autre part, il est utile pour mémoriser la date de naissance de toute instance de *Female*.

Ce cas de figure n'est pas toujours vrai. Hériter du comportement peut entrer en contradiction avec hériter de la structure. Supposons maladroitement que l'on

ajoute le code *private boolean _is_female;* dans la classe *Individual*, il apparaît alors de manière évidente que toute instance de *Female* encombre la mémoire par ce champ puisqu'il est stupide de mémoriser quel est le sexe d'une femme ! C'est une femme, on l'aura compris.

La classe *Female* doit-elle continuer d'hériter d'*Individual* ? Oui, bien sûr, du point de vue de l'héritage de comportement, il est intéressant de pouvoir calculer l'âge d'une femme et plus généralement de réutiliser les comportements standard des individus. Une femme est donc bien un individu particulier : c'est le principe de spécialisation.

L'héritage de structure pose donc globalement le problème des représentations et/ou de leurs formats qui peuvent s'avérer non judicieux pour les classes spécialisées. Par exemple, à supposer qu'une classe soit dotée de propriétés statiques comme des relations avec d'autres classes, toute héritière récupère une structure équivalente. Dans la figure 1.6, tout héritier du type *Complex* obtient virtuellement au final deux liens sur le type *Real* : la partie réelle et la partie imaginaire.

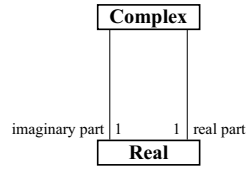


Figure 1.6 — En UML, hériter de la classe *Complex* entraîne la récupération des deux dépendances structurelles avec *Real*.

L'implantation de *Real* peut varier entre les types C++/Java *float* et *double* pour des raisons de précision, ou de concision, ou encore de performance. Les choix faits dans la classe *Complex* peuvent néanmoins s'avérer inappropriés pour ses éventuelles spécialisations. Un changement de format est malheureusement alors impossible (C++ et Java). En clair, une classe spécialisée à partir de *Complex* ne peut pas faire passer le type d'un champ hérité en *double* par exemple, s'il avait été déclaré *float* dans *Complex*.

L'héritage de structure appelé de façon ambiguë *Inheritance* dans le *Core Objet Model* de l'OMG, n'est en fait vu que comme un moyen non contractuel (*i.e.* sans contraintes ni règles précises) de réutilisation, tout simplement parce qu'il existe dans les langages de programmation.

Héritage de comportement, polymorphisme

Une dissertation sur l'héritage de comportement entraîne une étude poussée et significative du typage et du polymorphisme ainsi que les détails de définition et de redéfinition des fonctions : leur surcharge, leur masquage, enfin leur mise en œuvre régissant le comportement polymorphe des objets. Un bon tour d'horizon se trouve dans le chapitre 17 intitulé *Typing* du livre de Meyer [26, p. 611-642].

Bien que tous quatre typés statiquement (contrôle de conformité d'usage des types faits à la compilation par opposition à Smalltalk par exemple), Eiffel, Java, C++ et C# sont des langages plus ou moins sûrs avec pour les trois derniers, l'usage invétéré du *casting* ou conversion de types, souvent générateur d'erreurs à l'exécution.

Surcharge (overloading)

Deux fonctions ayant le même nom et membres de l'interface d'un type ou internes à ce type (déclarées dans, héritées, l'une héritée, l'autre déclarée) sont en statut de surcharge, si elles se différencient par leurs arguments (nombre et/ou types). Deux fonctions de même nom retournant un type différent ne peuvent pas être en surcharge mutuelle (Java et C++). En C++ (et en Java en adaptant légèrement), le code qui suit est interdit[1] :

```
float update(float);   // une première version d'une fonction update
double update(float);  // une version surchargée refusée
```

De même, en Java cette fois (et dans le même esprit en C++), les deux opérations qui suivent illustrent la surcharge :

```
public double update(float x) {return …}
public double update(float x,float y) {return …}
```

Le mécanisme de surcharge est utile pour partager sémantiquement un nom faisant référence à une tâche bien identifiée (*print*, *open*…). L'exemple le plus marquant et le plus intéressant est la surcharge pour les constructeurs. Dans la classe C++ qui suit décrivant la notion de polynôme, le constructeur est surchargé car soit l'argument est de type *double*, soit de type *list<pair<double,double> >*.

```
class Polynomial {
   private:
      map<int,double> _ai;     // coefficients du polynôme
   public:

      Polynomial(double = 0.); // ne peut pas cohabiter avec un constructeur
                               // sans arguments
      Polynomial(const list<pair<double,double> >&); // création par nuage
                                                     // de points
   …
};
```

Masquage (overriding)

Le mécanisme de masquage réside dans le fait d'associer un nouveau corps (*i.e.* une nouvelle suite d'instructions) à une opération déclarée dans une superclasse. En conservant strictement la même signature, on induit du polymorphisme par le fait qu'à l'exécution, la suite d'instructions effectivement mise en œuvre dépend dynamiquement du type de l'objet concerné. La précédente classe Java *Female* par exemple,

1. Attention aussi en C++ aux valeurs par défaut des arguments des fonctions qui interfèrent avec les possibilités de surcharge.

outrepasse (masque) le comportement associé à l'opération *public int age()* déclarée dans *Individual*, et ce en fournissant son propre code. L'illustration du polymorphisme est immédiate et contingente au masquage. Bien que l'instance *SophieDarnal* dans le code ci-dessous soit de type *Individual*, le système exécute bien le code établi dans *Female* au détriment de celui situé dans *Individual*. Le type de l'objet *SophieDarnal* est testé par le système à l'exécution pour attacher dynamiquement (*dynamic binding* ou *late binding*) le code approprié se trouvant en mémoire.

```
Individual SophieDarnal = new Female(28,07,1964);
System.out.println(SophieDarnal.age());
```

Malheureusement, l'usage intensif et généralisé de l'héritage se heurte aux fortes contraintes et limitations des langages de programmation par objets. En modélisation objet comme UML, la tâche se complique par la nécessité d'un système de typage performant, permettant de manière aisée et si possible transparente de retranscrire les mécanismes des langages de programmation. Abordons le problème de manière intuitive. Considérons la fonction qui retourne le frère aîné et la sœur aînée d'une personne, qui par coïncidence peut être cette personne elle-même. Ces deux propriétés sont formellement établies pour des individus, ce qui mène à écrire dans la classe *Individual* :

```
public Individual eldest_sister() {return …}
public Individual eldest_brother() {return …}
```

Notons que le calcul de ces deux propriétés, outre le fait de connaître la liste des frères et sœurs d'un individu, nécessite de connaître leur âge et leur sexe. Ce n'est pas là la difficulté, qui réside dans la signature de *public Individual eldest_sister()* qui devrait être *public Female eldest_sister()* dans la classe *Female*, ce qui est impossible au regard du fonctionnement de la surcharge en Java (voir section « Surcharge »). On ne peut donc pas par exemple connaître M. Anceau, le mari de la sœur aînée de *SophieDarnal* :

```
Male m = SophieDarnal.eldest_sister().my_husband();
```

La seule solution est le casting avec tous les risques d'atténuation de la fiabilité qu'il engendre. Le code qui suit n'est malheureusement pas sûr car sujet à des aléas (il dépend de ce qui a été fait avant, en l'occurrence le type réel de l'objet référencé par *SophieDarnal*).

```
Male m = ((Female)SophieDarnal.eldest_sister()).my_husband();
```

En modélisation objet, l'héritage (triangle blanc en UML, figure 1.7) est d'usage douteux dès que l'on s'aventure sur des choses, plutôt simples, mais basées sur un système de typage propre, absent d'UML malheureusement comme nous allons le voir au chapitre 2. Par exemple, *Complex* hérite de *Real* (attention ici, il n'y a aucun rapport avec la figure 1.6) mais comment dire que l'addition ou la soustraction de deux complexes est un complexe et non un réel ? Par ailleurs, comment une telle conceptualisation peut-elle être interprétée par les programmeurs au moment de l'implémentation ?

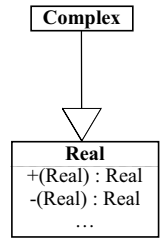


Figure 1.7 — Héritage de comportement, polymorphisme, comment en jouer dans UML ?

Même si le système de covariance d'Eiffel qui permet de changer le type des arguments et des retours résout le défaut patent du système de typage de Java, C++ ou encore C#, sa représentation en modélisation objet comme UML est encore illusoire d'où en conclusion intermédiaire, le fait que le typage, au sens formel et dans toute son efficacité, n'est pas un élément central de la modélisation objet en général et d'UML en particulier. Revenons à la solution utilisant le mécanisme d'ancrage d'Eiffel. Si l'on veut implanter une fonction self qui retourne soi-même, on a en Eiffel :

```
class A
   feature
      self : A is do
          Result := Current
      end;
   …
end

class B inherit A
   feature
   …
end

x,y : B
y := x.self -- incompatibilité de types, self retourne un objet de type A
```

En remplaçant le code de la classe A par celui qui suit, il n'est pas nécessaire de dire que la fonction *self* retourne un objet de type A si appliquée à un objet de type A et un objet de type B si appliquée à un objet de type B :

```
class A
   feature
      self : like Current is do
         Result := Current
      end;
   …
end
x,y : B
y := x.self -- OK, self retourne un objet de type B
```

Indéniablement, le gain du point de vue de la fiabilité est net et conséquent. Éviter les conversions de type est toujours bénéfique même si C++ a progressé dans ses der-

nières normalisations avec des opérateurs de conversion dédiés à la robustesse et plus évolués (*static_cast*, *dynamic_cast*… [9]) que les mécanismes originels hérités de C.

À titre de comparaison, le code Java qui suit illustre les lacunes de Java en matière de typage.

```
Individual SophieDarnal = new Female(28,07,1964);
// Female SophieDarnalBis = SophieDarnal;
Female SophieDarnalBis = (Female)SophieDarnal;
```

La contravariance empêche d'affecter une référence (ici *SophieDarnalBis*) statiquement typée via une classe descendante (*Female*), par une référence (*SophieDarnal*) typée par la classe dont elle descend (*Individual*). C'est toujours le contraire qui est possible sinon on pourrait accepter à la compilation, et donc statiquement, l'instruction *SophieDarnalBis.my_husband();* alors que dynamiquement on ne ferait référence qu'à une instance d'individu (*i.e.* en mémoire et en réalité, *SophieDarnalBis* ne pointerait pas sur une instance de *Female* mais sur une instance d'*Individual*). La deuxième ligne de code est en commentaire car elle ne passe pas la compilation. Une coïncidence (la première ligne nous assure que la deuxième n'entraîne pas un bogue) fait que le code est néanmoins robuste. Mais le plus choquant, c'est que la troisième ligne passe la compilation. Certes, elle est aussi robuste mais repose sur la même coïncidence. Le problème, c'est qu'un programme ne fonctionne pas à coup de coïncidences !

Classe abstraite

La notion de classe abstraite est également intimement liée à l'héritage et au polymorphisme. Une classe est abstraite, si et seulement si, elle offre au moins un service abstrait (celui-ci pouvant être hérité), c'est-à-dire un service dont la signature est établie mais le corps est indéfini car indéfinissable. Pour illustrer cette notion, voici le code de la classe *Individual* où le corps des fonctions *is_female()* et *is_male()* est inconnu du fait de l'absence d'information sur le sexe de l'individu.

```
public abstract class Individual {
   // implementation
   protected java.util.Calendar _birth_date;
   private static java.util.Calendar Now() {
      return new java.util.GregorianCalendar();
   }
   // interface
   public Individual(int day_of_month, int month, int year) {
     _birth_date = new java.util.GregorianCalendar(year,month,day_of_month);
   }
   public int age() {
      return Now().get(java.util.Calendar.YEAR) -
                  ➥_birth_date.get(java.util.Calendar.YEAR);
   }
   abstract public boolean is_female();
   abstract public boolean is_male();
   // public Individual eldest_sister() {return ...}
   // public Individual eldest_brother() {return ...}
}
```

Une classe abstraite n'est pas directement instanciable au simple fait que lancer la fonction *is_female()* par exemple sur une instance d'individu, est empêché par l'inexistence de code. Une classe concrète héritant d'une classe abstraite a donc pour vocation de lever les inconnus, en donnant le code à tous les services abstraits, chose faite dans la classe *Female* précédemment :

```
public boolean female() {
   return true;
}
public boolean male() {
   return false;
}
```

La question naïve et récurrente est : que faire d'un type qui ne sert, de prime abord, à rien ? Notons tout d'abord que créer un objet de type *Female* comme *SophieDarnal*t, c'est instancier *indirectement* la classe *Individual*. Le type de *SophieDarnal* est prioritairement *Female* et plus accessoirement *Individual*. C'est le principe de substitution cher au typage où toute instance des héritiers d'*Individual* est remplaçable par n'importe quelle autre dans une opération syntaxiquement et sémantiquement définie dans *Individual*.

Secundo, et plus conceptuellement (modélisation objet), à la question, peut-on, sait-on, dans la réalité déterminer si un individu est de sexe mâle ou femelle ? La réponse est oui. À ce titre, la déclaration des fonctions *is_female()* et *is_male()* n'a pas à être retardée dans la classe *Female* ou celle de *Male*, ou les deux. Imaginons des milliers de sous-classes directes d'*Individual*. Faudrait-il écrire des milliers de fois ce que l'on sait alors que l'on peut l'écrire une seule fois ? L'économie d'échelle qui apparaît là est la clé d'une partie de la maintenabilité des programmes.

Étudions ce problème global à l'aide d'un programme C++ en ne jouant pas du concept de classe abstraite. Une fonction *f()* a une première suite d'instructions fixe, *i.e.* bien établie, et une seconde qui dépend, dans un premier temps, de deux caractéristiques bien distinctes.

```
class C {
   …
   void f();
   …
};
void C::f() {
   i++;
   j += i;
   … // jusqu'ici, tout ce qui est « stable »
     // ensuite ce qui est plus « variable », et donc la nécessité de tester
   if(…
   else…
}
```

Maintenant, en mettant en œuvre la notion de classe abstraite, on crée une nouvelle fonction *g()* qui représente la partie instable. Elle est déclarée abstraite, rendant alors de fait la classe C abstraite, et est appelée dans *f()* à l'endroit requis. Le

code se simplifie drastiquement ce qui réduit potentiellement la maintenance de *f()* mais le programme n'est pas encore terminé.

```
class C { // classe abstraite en C++, pas de mot-clef réservé,

          // lire le code pour s'en rendre compte
   …
   void f();
   virtual void g() = 0; // moyen de déclarer un service abstrait
                         // en C++ : virtual ainsi que = 0
   …
};
   void C::f() {
   i++;
   j += i;
   g();
}
```

La dernière manipulation fait appel à la création de deux sous-classes de la classe C pour rendre concrètes les deux versions de *g()* nécessaires à la prise en charge des « deux caractéristiques bien distinctes », notre hypothèse initiale. La maintenance s'opère alors sur la base du principe de forte cohésion. Avec l'usage du concept de classe abstraite, il y a trois lieux d'intervention : le corps de *f()* dans C (partie générale) et les deux corps de *g()* (parties spécifiques) propres à chaque sous-classe de C. Dans le futur, s'il advient qu'une troisième caractéristique doit être prise en compte dans les besoins, la première version où C n'est pas abstraite demande de changer le contenu de la fonction *f()* (nouveau cas dans le test) ce qui est la pire des choses ! Toucher à du code existant qui marche bien demande une nouvelle mise au point. En pratique, la prise en charge de nouvelles exigences détruit, par cette méthode, les fonctionnalités déjà implantées. En clair, la mise au point de la nouvelle version de *f()* a aussi pour effet de bord de faire que ce qui marchait ne marchera plus avant longtemps. Avec C abstraite, la maintenance revient à créer une troisième classe pour donner un troisième corps à *g()*. L'existant est préservé et les effets de bord de la première façon de faire évités. Une telle approche incrémentale, ici idéalisée, a le défaut d'augmenter le nombre de classes et la complexité de la hiérarchie d'héritage. C'est une complication à ne pas négliger. Un arbitrage est possible. On peut en effet limiter l'utilisation de l'héritage et des classes abstraites, mécanismes un peu lourds dans certains cas. Il n'en demeure pas moins que l'anticipation de l'évolution, partie intégrante de la maintenabilité (dégager par avance une aptitude, un potentiel à la maintenance), demande leur mise en œuvre à grande échelle. Idem en modélisation où la rationalité, la compréhensibilité doivent primer sur les optimisations « à la petite cuillère ».

L'économie faite, de « bout de ficelle » perceptible dans cet exemple C++, devient gigantesque dans la généralisation à grande échelle de l'approche (voir les bibliothèques natives de Java ou encore d'Eiffel).

Héritage de comportement, polymorphisme : synthèse

Rappelons ici que le polymorphisme n'est pas automatique en C++ et en l'absence du label *virtual*, (voir précédemment) une fonction ne peut pas engendrer de com-

portement polymorphe. Au-delà, ce mécanisme a souvent été critiqué pour son coût d'utilisation de temps processeur notamment en programmation C/C++, où culturellement, la maîtrise bas niveau de la mémoire et du déroulement des instructions est privilégiée. Reste l'enjeu en modélisation centrée sur des descriptions des comportements d'objets par automates à nombre fini d'états, réseaux de Petri ou autres. Le chapitre 3 traite ce sujet.

1.4.3 Héritage multiple

Commençons ici à l'envers pour commenter l'héritage multiple en donnant notre conclusion sur son usage avant d'en voir les éléments techniques. Oui, la mise en œuvre de l'héritage multiple est pertinente et source de bénéfices en programmation par objets, à condition d'être utilisé par des équipes de développement de haut niveau technique. L'expérience montre que l'héritage multiple laissé à des programmeurs peu chevronnés[1] génère des programmes compliqués et donc à coût de maintenance élevé. Au constat que la technologie objet rebute ou sème le doute auprès d'un nombre non négligeable de développeurs, s'ajoute donc l'investissement conséquent dans ses mécanismes les plus pointus et les plus avancés, comme l'héritage multiple, avec souvent, des difficultés fondées pour bien en mesurer tous les avantages.

L'héritage multiple a été un sujet de recherche intensif dans les années quatre-vingt et quatre-vingt-dix dont la synthèse apparaît dans [33], et ce dans les domaines de l'intelligence artificielle et du génie logiciel. Malheureusement, parce qu'il n'a pas à notre sens la même finalité dans ces deux domaines, beaucoup d'écrits ont créé la confusion en ne distinguant pas son intérêt pour chaque domaine. Dans le domaine du génie logiciel, la seule question qui vaille est : en quoi l'héritage multiple est un facteur direct ou indirect de la qualité logicielle ? Dans le domaine de l'intelligence artificielle, de telles considérations sont de second ordre au regard de la problématique centrale de représentation des connaissances et de formalisation du raisonnement.

Venons-en à la réalité quotidienne du développement. D'une part, l'héritage multiple est « dans la nature ». En modélisation objet, comme nous allons le voir dans le chapitre 2, décrire des graphes d'héritage multiple n'est peut-être pas commun mais au moins probable. D'autre part, sur le terrain, la réutilisation inévitable des bibliothèques standard mène le programmeur à voir, et donc étudier, du code usant de l'héritage multiple. En C++, la classe *iostream* en est l'archétype :

```
class iostream : public istream, public ostream {
   …
};
```

En Eiffel, la nature même du langage invite le programmeur à très vite pratiquer l'héritage multiple du fait que les bibliothèques de base et la manière intuitive de

1. Dans le même esprit, nous pensons par ailleurs que le langage Eiffel n'a pas percé comme il le devait à cause de son usage intensif (abusif ?) de l'héritage multiple.

conception des classes, y font appel. Son absence dans Smalltalk ainsi que dans Java et C# (remarque à atténuer en regard des interfaces Java et C# qui peuvent, au contraire des classes, hériter de façon multiple) ou encore sa non-utilisation notoire dans des bibliothèques réputées comme STL ou les MFC, prête à interrogation, ou mieux, laisse entendre que l'on peut s'en passer ! À ce titre, la tentative avortée à la fin des années quatre-vingt d'introduction de l'héritage multiple dans la version industrielle de Smalltalk-80 témoigne de la difficulté des programmeurs à comprendre et bien utiliser ce mécanisme. Dans l'expérience Smalltalk-80, la mise en œuvre de l'héritage multiple a engendré des programmes qui devenaient parfois inintelligibles et par là même non maintenables, d'où un retour rapide à l'héritage simple. Pour Java, la dichotomie salutaire classe/interface dissociant héritage de structure et héritage de comportement a bien entendu atténué l'intérêt d'un héritage multiple « pur » (*i.e.* à la C++ ou à la Eiffel). Ardent défenseur de l'héritage multiple, Meyer propose les notions d'héritage « sous l'angle du type » et d'héritage « sous l'angle du module ». Il justifie ainsi l'héritage multiple par le fait qu'une utilisation simultanée de ces deux processus d'héritage doit pouvoir être offerte au programmeur lorsqu'il construit une classe. La nuance qu'il nous faut apporter est que si l'héritage multiple est bien intégré dans Eiffel par le fait que c'est un moyen de programmation efficace mais surtout incontournable, cela n'est pas forcément vrai dans un autre langage. Tout programmeur Eiffel est quasiment contraint au départ de se plonger dans ce mécanisme, la bibliothèque standard Eiffel l'utilisant abondamment. Pour les autres langages de programmation objet, la réalité est plus nuancée.

Contrairement à notre habitude, notre analyse de l'héritage « sous l'angle du module » de Meyer est très critique. Faire qu'un arbre binaire (*BINARY_TREE*) par exemple, hérite de la notion de cellule (*CELL*) est intellectuellement néfaste (gauche de la figure 1.8) même si cela est satisfaisant dans le cadre d'Eiffel. Au contraire, on préfère dire que les nœuds d'un arbre binaire contenant les données satellites de l'arbre (celles autres que les données nécessaires à sa manipulation : balayage, insertion, suppression…) peuvent être modélisés par une classe *CELL* mais avec la formalisation à la droite de la figure 1.8.

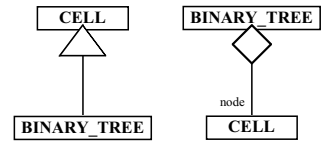


Figure 1.8 — Héritage (gauche) contre agrégation (droite).

De manière rationnelle, un arbre binaire est mieux représenté comme une « agrégation » (losange blanc en UML) de cellules : les nœuds de l'arbre (rôle *node*). De plus, une approche saine consisterait à rendre cette relation d'agrégation invisible (signe - en UML). Le modèle pourrait être ainsi développé avec d'autres constructions de modélisation UML, dont cette relation d'agrégation en particulier inexistante dans les langages de programmation mais outil de tout premier plan en

UML. Ce sujet aborde la conception objet qui est plus abondamment traitée à la fin de ce chapitre (voir en particulier la notion de composant embarqué dans la section intitulée « Conception ») ainsi que dans le chapitre 2 et les études de cas de cet ouvrage.

Au-delà des langages de programmation, il est donc nécessaire de juger de la pertinence de l'héritage multiple en modélisation, dans UML en particulier. La question embarrassante est : comment coder en Smalltalk ou en SQL99 par exemple, qui ne disposent pas de l'héritage multiple, des modèles UML faisant appel à ce mécanisme ? Attention à toute réponse irréfléchie, du genre : il faut faire cas du langage de programmation au moment de la fabrication des modèles pour éviter le problème. C'est un non-sens absolu ! Un langage de modélisation propose par essence des constructions abstraites. Faire interférer la spécificité du langage de programmation est une aberration. C'est une utilisation d'UML malheureusement courante et déviante mais qui est encouragée par UML lui-même au travers de la présence dans le langage de constructions graphiques assez proches de mécanismes de programmation. À cet égard, certains, avec justesse malheureusement, considèrent UML comme un outil d'*implementation modeling*, c'est-à-dire une juxtaposition de constructions graphiques imitant ce que l'on trouve dans les langages de programmation.

Nous pensons que l'héritage multiple est une technique d'analyse et de conception objet utile, si elle est mise en œuvre avec parcimonie et dans les cas adéquats. Que ce soit en programmation ou en modélisation, le fait par exemple que dans le règne animal, un ornithorynque soit à la fois un ovipare et un mammifère justifie son usage. En C++, cela donne :

```
class Animal {
};

class Mammal : virtual public Animal {
};

class Oviparous : virtual public Animal {
};

class Ornithorynchus : public Mammal,public Oviparous {
};
```

L'héritage virtuel de C++ (sans rapport avec les fonctions abstraites déclarées *virtual* par ailleurs en C++) est le moyen de gérer les conflits de propriétés statiques : la duplication ou non des champs. Ici, tout champ déclaré dans *Animal* n'apparaît qu'une fois dans *Ornithorynchus* bien qu'hérité selon deux chemins différents (via *Mammal* et via *Oviparous*). Plus généralement, la gestion des conflits en héritage multiple a souvent été le fond de commerce de ce mécanisme, sans intérêt en génie logiciel car gérée avec des techniques simples comme par exemple, le renommage Eiffel ou les opérateurs de portée C++.

Pour terminer, insistons sur l'approche Java et C# qui à travers la notion d'interface, c'est-à-dire une spécification de service sans corps (avec des attributs de classe éventuellement) supportent l'héritage multiple de comportement tout en ne per-

mettant que l'héritage simple de structure, moyen simple et élégant pour éviter les écueils de l'héritage en général. Ainsi l'exemple C++ qui précède pourrait être codé en C# comme suit :

```
public interface Animal {}
public interface Mammal : Animal {}
public interface Oviparous : Animal {}
public sealed class Ornithorynchus : Mammal, Oviparous {}
```

1.4.4 Gestion des exceptions

La segmentation méthodique des programmes en classes favorise la correction d'une défaillance s'il est diagnostiqué que tous les éléments à la source de la défaillance ne dépendent que de la classe elle-même. En revanche, les déficiences résultant de l'interaction des classes sont à la fois difficiles à identifier et à prendre en charge, raison de plus si les classes collaborant proviennent d'auteurs différents : concepteurs de bibliothèques et concepteurs d'applications sont les deux rôles les plus courants dans le processus fournisseur/réutilisateur ou fournisseur/client. L'idée de la gestion des exceptions est de mécaniser le processus de capture et de traitement de comportement anormal d'objet selon le bon sens qui suit : le fournisseur d'un composant est à même de détecter les erreurs se produisant lors de l'usage du composant qu'il écrit (maîtrise de la partie privée du composant). Cependant, ne connaissant pas le contexte d'utilisation (contexte appelant), il ne peut pas prendre de décision quant à la suite à donner aux anomalies : c'est la levée d'une exception. Le client d'un composant, connaissant le contexte d'exécution, est à même de décider de la suite à donner à ces anomalies se produisant lors de l'usage du composant. Cependant, il ne peut pas détecter directement les défaillances (absence de maîtrise de la partie privée du composant) : ce sont la capture et le traitement d'une exception, ou à défaut sa propagation au niveau immédiatement extérieur jusqu'au programme principal et au système d'exploitation en cas de difficulté de décider l'action à entreprendre.

L'idée maîtresse de la programmation défensive en approche objet est d'avoir des mécanismes dédiés pour en quelque sorte créer quasiment deux applications : le calcul applicatif proprement dit et son contrôle « temps réel » si l'on peut dire, afin de garantir le meilleur des fonctionnements. Les moyens de balisage de ces deux types de code donnent l'idée de deux applications même si, au final, tout se trouve dans les mêmes fichiers source.

Les problèmes, anomalies, comportements anormaux ou encore défaillances et déficiences se groupent sous le terme général d'exception. Le mécanisme de défense est la gestion des exceptions.

Défense des programmes

Ada-83 reste l'un des pères fondateurs de la programmation défensive par l'introduction claire et rationnelle de la gestion des exceptions. Le plus des langages objet comme C++, C# ou encore Java, c'est d'avoir vu les défaillances comme des objets : les attributs d'objets exceptions documentent les anomalies détectées (valeur(s) de

la cause du problème par exemple) et les opérations de ces mêmes objets offrent une interface élégante pour l'analyse, comme la lecture de la pile d'exécution.

La déclaration dans une classe des erreurs éventuelles qu'elle peut produire est avant tout une facilité de documentation pour le créateur de la classe : si la fonction *f* est mise en œuvre, l'erreur de type *e* est possible. C'est un message d'avertissement au client utilisateur de la classe où *f* est offerte.

En C++, contrairement à Java, les exceptions levées dans le corps des fonctions ne documentent pas le prototype de ces fonctions. En clair, le code qui suit est possible mais non normé (il faut une directive de compilation spéciale pour être accepté) :

```
void f() throw(std::logic_error); // logic_error est un type d'exception
                                  // prédéfini de C++
```

En Java, on suffixe la signature d'une opération par une liste de types d'exceptions possibles. Dans l'exemple qui suit, on met en œuvre la classe prédéfinie *Exception* de la bibliothèque Java :

```
public class Temperature {
   …
   public Temperature(float value,byte unit) throws Exception {
     …
     switch(unit) {
        case Celsius: _value = value;
           break;
        case Fahrenheit: _value = (value - 32.F) * 5.F / 9.F;
           break;
        case Kelvin: _value = value + -273.15F;
           break;
        default: throw new Exception("Illegal unit");
     }
     if(_value < -273.15F) throw new Exception("Illegal temperature");
   }
}
```

Pour la majorité des types d'exception Java (tous sauf ceux qui héritent de *Error* et de *RuntimeException*, voir l'ouvrage de Gosling *et al.* [15]), la capture et le traitement sont obligatoires (voir code suivant). C'est le client informé des risques associés globalement à toute la classe (mot-clef *throws* suffixant le constructeur de *Temperature*), qui doit *contractuellement* prévenir les problèmes à chaque usage qu'il fait d'une opération. La levée de l'exception proprement dite s'opère via la clause *throw* (attention, pas de *s* !), à charge du fournisseur de la classe. On remarque pour ce dernier, le test (*if*) qui est l'unique moyen en Java de déterminer s'il y a anomalie. Le client est alors contraint de réutiliser la classe *Temperature* :

```
try {
    Temperature t = new Temperature(18.F,Temperature.Celsius);
}
catch(Exception e) {
    System.exit(1);
}
```

Si l'anomalie n'apparaît pas (bon usage du client), le code (encadré dans la clause *try*), une fois effectué, poursuit sa route après l'accolade fermante de la clause *catch*. Sinon, la tentative de correction s'opère à la première instruction après cette fois, l'accolade ouvrante de *catch*. Dans le code qui précède, l'instruction de sauvetage retenue, c'est sauve qui peut, arrêt du programme (*System.exit(1);*). À éviter, bien entendu.

L'exception vue comme un objet est atteignable par le label *e*. Nombreuses sont les facilités offertes par Java pour analyser la défaillance. Il suffit pour cela d'appeler des services de la classe *Exception* via *e*. Au-delà, le système est extensible par le fait surtout que l'on peut hériter des types d'exception prédéfinis. Nous pouvons aussi le faire en C++ comme dans les lignes de code suivantes, toujours avec le concept de température :

```
class Invalid_temperature_exception : std::exception { // classe prédéfinie
                                                       // exception de C++
   public:
    inline const char* what() const {return "Invalid temperature exception";};
};
class Temperature {
   …
   public:
    void decrement() throw(Invalid_temperature_exception); // non normé
                                                           // pour l'instant
};
```

Chaque langage a ses subtilités, comme la clause *finally* de Java pour s'assurer d'un traitement final quelles que soient les circonstances : succès, échec ou mode dégradé. En Eiffel, la clause *rescue* (équivalente à *catch*) peut s'adouber de l'instruction *retry* pour redémarrer la fonction à son début. L'aspect le plus intéressant d'Eiffel est l'écriture « en dur » dans le code des règles de bon fonctionnement, et donc, par déduction, des causes formelles qui peuvent amener le système à déroger au scénario normal ou attendu. C'est la programmation ou conception par contrats qui va, comme en Java, au-delà du simple fait de traiter une exception, si elle se produit.

Conception par contrats

La notion de contrat est représentée par le créateur et le réutilisateur d'une même classe qui forment un pacte de non-agression. En Java, le concepteur de classe dit : « tu risques de générer tel problème si tu utilises ce service de cette classe » mais il n'indique pas comment l'éviter. En Eiffel, la précondition du service est une assertion logique écrite en Eiffel qui, si elle est violée, est le phénomène déclencheur de l'exception. En contrepartie, la post-condition est garantie « non transgressée » par l'auteur de la classe si le pacte de non-agression n'a pas été brisé en entrée par le réutilisateur : respect de la précondition mais aussi de l'invariant.

La programmation par contrats a deux facettes caractéristiques. La première, c'est d'être définitivement et invariablement associée à Eiffel qui est le seul langage qui la supporte de manière native (préconditions et post-conditions de routines ainsi qu'invariants de classes/boucles et variants de boucles). La seconde, c'est d'être cla-

mée comme un principe fondateur de la technologie des composants logiciels d'où une aura et une reconnaissance chaque jour raffermies. Des implantations des contrats en Java tendent ainsi à se développer comme *iContract* (voir la section 1.10, intitulée « Webographie ») non pour altérer le langage tel qu'il existe aujourd'hui, mais pour équiper les composants logiciels bâtis sur la base de Java de la toute-puissance de ce mécanisme.

Le besoin de ne pas polluer le code avec des tests de toute nature sur la validité momentanée des données, et donc des objets, fait appel à des mécanismes idoines dont le plus connu est la macro-instruction[1] *assert* de C/C++. Ainsi *assert(i >= 0 && i < n /* i.e. la capacité du tableau */);* est bien connue des utilisateurs de tableaux C/C++. Ce code n'a que vocation de mise au point, et en maintenance favorise la lisibilité des programmes par leur balisage. Les mots-clefs *require* (expression de pré-condition) et *ensure* (expression de post-condition) encadrent en Eiffel les opérations et donnent les conditions sous lesquelles elles peuvent fonctionner correctement. La violation de ces conditions génère une exception.

Dans l'exemple de code qui suit, la classe *Shell* représente un prompteur à l'écran où l'utilisateur, via un buffer interne, saisit sa commande. Le service privé *buffer_cleaning* effectue un nettoyage standard du buffer. Pour cela, il doit être alloué (*not buffer.Void*). À l'issue du nettoyage, il est certifié vide (*buffer.all_cleared*). C'est le contrat global[2] avec des propriétés à tout moment vérifiées (l'invariant : *cursor <= Buffer_size*) en l'occurrence ici un curseur de type entier qui navigue entre 1 et 256. La gestion des exceptions (*rescue*) est ramenée à son stade le plus primitif : on tente de remettre le système « en ordre » (*if … then)* avant de redémarrer (*retry*) la fonction puisqu'elle vient d'échouer.

```
class SHELL
   feature
   … -- propriétés publiques
   feature {NONE}
      Enter : CHARACTER is '\r';
      Zero : INTEGER is 0;
      Buffer_size : INTEGER is 256;

      buffer : ARRAY[CHARACTER];
      cursor : INTEGER

      buffer_cleaning is
        require -- précondition
         not buffer.Void
        do
         buffer.clear_all;
        ensure -- post-condition
         buffer.all_cleared
        rescue
```

1. Ce n'est pas une fonction. Elle est expansée dans la version binaire qui sert à déboguer.
2. Le lecteur attentif notera que la routine étant privée, le contrat est posé entre l'auteur de la classe et lui-même, car il est le seul utilisateur possible ! Ce détail ne change néanmoins rien au principe exposé.

```
            if … then
            retry
        end; -- buffer_cleaning
        … -- autres propriétés privées
    invariant -- invariant
        cursor <= Buffer_size
end -- Shell
```

La puissance du langage Eiffel est donc aussi de proposer des invariants de classe devant être valides (i.e. « vrais ») entre chaque instruction d'une opération offerte par la classe. Violer l'invariant dans l'instruction, c'est-à-dire au sein des différentes sous-instructions machine qui la composent, est bien entendu possible. Le cas contraire est irréaliste car on testerait à chaque cycle du processeur : lenteur inacceptable en pratique.

Simuler la notion d'invariant avec la clause *assert*[1] de C++ demanderait de préfixer chaque ligne de code des fonctions membres par l'assertion, sans pouvoir jouer en plus sur les fonctions héritées. En Eiffel, les invariants d'une classe de base et d'une héritière directe se conjuguent logiquement par un « et » pour s'appliquer à l'héritière. Au-delà, tout le système des contrats s'intègre par des règles formelles à l'héritage : conjugaison des préconditions, etc.

Le défaut de la programmation par contrats est son coût d'utilisation de temps processeur. Tout ou une partie des assertions est donc inhibé dans la version livrée aux utilisateurs (*release*), ce qui a toujours été l'esprit de la clause *assert* de C/C++. La persistance éventuelle du code de contrôle est plus aujourd'hui un sujet de recherche qu'une pratique industrielle réelle et maîtrisée. Voir par exemple l'article de Groß *et al.* [16] pour une technique de test de contrat en ligne, et donc de supervision du code applicatif par du code de test intégré et persistant dans les versions de *release*, et ce dans le domaine des composants logiciels.

1.5. AUTRES MÉCANISMES

Nous sortons ici des mécanismes natifs de l'approche objet pour nous intéresser à des concepts de programmation/modélisation essentiels du génie logiciel mais dont la technologie objet n'a jamais eu la primeur, et se trouve par ailleurs en compétition avec d'autres langages, méthodes ou outils.

1.5.1 Concurrence et parallélisme

La concurrence, le parallélisme et donc les moyens de synchronisation ont été supportés dans les langages de programmation de manière native (mécanismes et opérateurs dédiés comme le rendez-vous d'Ada) ou alors conceptualisés, et donc outillés, via des classes et des bibliothèques (Smalltalk, C++ et Eiffel), jusqu'à des approches

1. Voir aussi le concurrent *check* en Eiffel.

hybrides représentées par ce que l'on trouve aujourd'hui en Java ou en C#. Des mots-clefs comme *synchronized* (Java) ou *lock* (C#) montrent par exemple que la pose de verrous (appelant par ailleurs leur gestion dans la machine virtuelle) est le quotidien du programmeur. En Java, les processus (niveau système d'exploitation) et les fils de contrôle (*Thread*) au niveau machine virtuelle restent néanmoins représentés et ne sont représentables que par des instances de classes prédéfinies (bibliothèque noyau *java.lang.**, classe *Thread* par exemple). En modélisation comme UML, les besoins de tels outils sont évidents et ce sont les automates à états finis qui supportent la description de comportements parallèles, voire concurrents.

Parallélisme niveau type

De prime abord, coder en Java deux classes différentes, disons C1 et C2, ou spécifier en UML deux types distincts, disons T1 et T2, suggère que leurs instances peuvent virtuellement tourner en parallèle. Les données embarquées dans l'objet rendent ce dernier autonome (*self-contained*) jusqu'à un certain niveau de calcul. Au-delà, il doit collaborer et donc communiquer.

Deux instances de C1, ou une instance de C1 et une instance de C2, peuvent donc s'exécuter de manière concomitante, si elles ne partagent pas de données communes (variable de classe par exemple), n'échangent pas de données, ne se sous-traitent pas de services, s'ignorent tout simplement. En modélisation, T1 et T2 peuvent être spécifiés via deux automates à états finis. En prenant l'hypothèse la plus ouverte, deux instances respectives de ces deux types d'objet ont *potentiellement* un comportement parallèle. Ceci étant, plusieurs cas de figure peuvent se produire. *Primo*, les automates des deux types ne communiquent pas (en UML 1.x par exemple, la notation ^*m* correspond à un « envoi de message » et incarne donc la communication), cela signifie que deux instances respectives des deux types d'objets (quelles qu'elles soient puisque l'on raisonne en termes de type dans la spécification) *ne peuvent pas* communiquer. Implémentées, des instances de ces types à l'exécution cette fois-ci, peuvent être associées à des processus différents. *Secundo*, les automates communiquent. À l'exécution toujours, une implémentation parallèle impose le besoin de techniques de synchronisation puisque les modalités, les moyens et les moments de communication ne sont pas quelconques : les deux automates les formalisent précisément.

De manière plus générale, en modélisation objet, il peut y avoir des contraintes explicites de représentation d'aspects parallèles et/ou concurrents (terme « concurrent » utilisé au sens propre, c'est-à-dire en compétition pour l'utilisation de ressources au sens large incluant les objets eux-mêmes). Mais il peut aussi exister du parallélisme implicite, qui au regard des supports d'implémentation possibles (langage de programmation retenu par exemple) et des environnements d'exécution disponibles, peut aboutir à des variations d'implémentation importantes. Cette idée est résumée par Cook et Daniels dans [11] : « *The specification model is idealised because it assumes infinitely fast processing, and infinite amount of totality reliable persistent storage with instantaneous random access.* »

Parallélisme au niveau objet

Le parallélisme au niveau objet signifie par exemple qu'un objet fait tourner deux de ses fonctions membres en parallèle, ou pourquoi pas la même fonction de façon concomitante, ce qui dans ce second cas, interroge sur le mode de lecture et d'écriture des données internes de l'objet : « anarchique » ou contrôlé. Le code Java qui suit[1] est extrait de [15] et illustre le mode contrôlé, en l'occurrence l'utilisation du modificateur de champ *volatile* qui oblige chaque instance de *Thread* à mettre à jour les champs *i* et *j* à chaque accès. En effet, ces instances travaillent sur des copies de *i* et *j* et le processus de mise à jour des originaux en fonction des copies est justement, par défaut, « anarchique », c'est-à-dire non contrôlable par le programmeur si *volatile* n'est pas utilisé.

```
class Test {
   static volatile int i = 0, j = 0;
   static void f() {i++; j++;}
   static void g() {System.out.println("i = " + i + " j = " + j);}
}
```

Le code qui précède assure que les valeurs de *i* et *j* apparaissent à chaque affichage avec *i* ≥ *j* du fait qu'ils démarrent tous deux à zéro et que *i* est incrémenté avant *j*. Sans *volatile*, l'exécution parallèle de *f()* et de *g()* peut engendrer un affichage aléatoire où éventuellement *i* < *j*.

En modélisation objet, le parallélisme intra-objet est tout à fait naturel et commun (dans UML spécifiquement) via l'utilisation des *Statecharts* de Harel [18].

Le parallélisme au niveau objet a une autre envergure qui sous-entend l'égalité « objet = processus » nous entraînant vers les systèmes multi-agents où l'agent, par un comportement « autonome » justement, sous-entend qu'il est au minimum un processus au sens « système d'exploitation ». Outre le fait que ce thème dépasse le cadre de cet ouvrage, nous pensons que les systèmes multi-agents ne contribuent pas réellement à la problématique du génie logiciel, c'est-à-dire l'obtention de la qualité du logiciel, mais sont des supports intéressants pour mieux formaliser le raisonnement, représenter éventuellement des connaissances, et donc s'inscrivent plus volontiers dans le cadre de l'intelligence artificielle.

Autres notions pour la concurrence et le parallélisme

C'est bien entendu le concept de message, concept historique et fondateur de la programmation par objets (dans Smalltalk surtout) qui est, à notre sens, à même de fournir un cadre rationnel et efficace pour la programmation distribuée et parallèle en technologie objet.

Le parallélisme donc, l'invocation de méthode, particulièrement, si elle est distante, la concurrence via l'utilisation de ressources partageables, et la communication en général, font appel à des techniques de synchronisation dans les langages de

1. Les deux variables et les deux fonctions sont « de classe » (*i.e. static*), mais le principe reste le même.

programmation, voire à des systèmes préfabriqués (bibliothèques de classes évoluées) comme *Java Message Service* (JMS) dans le monde Java pour des architectures à composants logiciels. De manière générale, en programmation objet, chaque langage/outil a sa propre approche qu'il faut maîtriser avec parfois des déficiences qu'il faut surmonter. Par exemple, Sandén dans [30] montre les dangers et limites de la programmation des *Thread* en Java.

En spécification, c'est la notion d'événement qui est à notre sens critique et centrale mais qui malheureusement en UML (voir chapitre 3) est noyée dans un imbroglio détestable avec d'autres notions comme celle de message, signal ou autre. Szyperski *et al.* dans [32, p. 181-185] tentent une clarification sur le sujet en caractérisant les messages comme les moyens de communiquer les événements, ces derniers étant des artefacts « localisables » dans l'espace et dans le temps. Une définition complémentaire plus caractéristique est que pour un système (et donc un objet ou un composant logiciel) est événement tout ce qui entraîne une modification du système, modification qui ne peut pas se déduire du système lui-même. Cette définition souligne le caractère externe de l'événement, le phénomène que l'on subit par opposition aux actions que l'on décide.

Dans cet ouvrage, nous allons nous efforcer essentiellement de marier harmonieusement constructions de modélisation inhérentes à la concurrence et au parallélisme avec les supports retenus pour l'implémentation, dans le monde Java particulièrement. Par exemple, de nombreuses applications utilisent des services de cadencement obtenus usuellement par la notion de *Timer* associée à des mécanismes de notification ou événements appelés *time-out*. La spécification UML d'un tel système (chapitre 5) ou son implémentation Java doivent se faire sous un angle technique : la classe *Thread* Java est mise en œuvre. Par ailleurs, une bibliothèque de classes Java implémente les *Statecharts* de Harel pour gérer les états parallèles par exemple. En technologie des composants, la même approche consiste à connecter le système cadencé sur le service approprié : *Timer Event Service* dans CORBA par exemple.

1.5.2 Persistance

La durée de vie d'un objet est en général limitée au temps d'exécution d'une application dans laquelle cet objet participe (figure 1.9). En mémoire primaire, chaque langage dispose de mécanismes propres, plus ou moins à la charge du programmeur : destructeurs virtuels ou non, voire classe prédéfinie appelée *Allocator* en C++/STL, ou encore ramasse-miettes (*garbage collector*) en Java, Eiffel et Smalltalk-80 avec accès plus ou moins restreint offert au programmeur pour traitements spéciaux. En effet, les systèmes embarqués par exemple imposant une gestion fine des ressources comme la mémoire, se satisfont rarement des mécanismes de base et/ou primitifs du type ramasse-miettes.

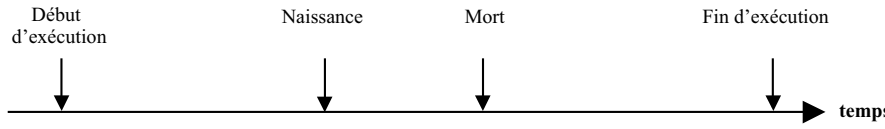


Figure 1.9 — Cycle de vie standard d'un objet non persistant.

Moyens de la persistance élémentaire

La persistance des objets ou prise en charge en mémoire secondaire (stockage) a lieu via un archivage binaire supporté par des types idoines : *Serializable* en Java, Storable en Eiffel, *BinaryObjectStorage* en Smalltalk-80 ou encore *CArchive* en C++/MFC.

Il en résulte le schéma de la figure 1.10 où les objets traversent les occurrences d'exécution des applications et peuvent, éventuellement, être échangés entre les applications sous leur forme stockée. La nécessité manifeste de disposer de moyens plus sophistiqués (le questionnement des propriétés des objets par exemple) a donné naissance à des outils plus évolués de la persistance.

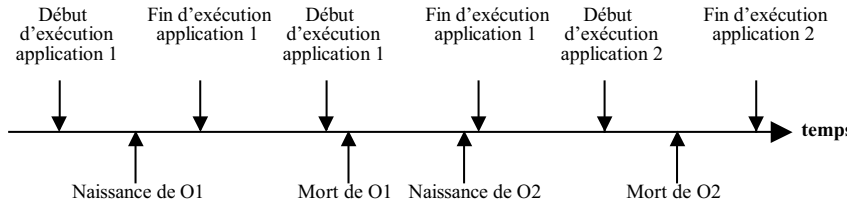


Figure 1.10 — Persistance de type élémentaire.

Bases de données objet

Le besoin de langages propres destinés à la structuration explicite des objets sur leur support de stockage en vue de les définir, de les manipuler et d'associer des mécanismes d'administration (gestion de la confidentialité, de l'intégrité ou encore des transactions) a engendré les bases de données objet dont le standard de l'ODMG[1] [8] et SQL99 (norme ANSI/ISO/IEC n°9075) extension plus naturelle des bases de données relationnelles.

Les architectures à objets ou à composants actuelles voient donc dans ces systèmes de gestion de bases de données objet ou SGBDO, des couches particulières assurant « le » service de gestion de la persistance, plus les services qui en découlent. Des techniques de mise en correspondance (voir par exemple la classe *java.sql.Types* dans le monde Java) et d'échange des objets s'imposent donc, entre une application cliente par exemple et le SGBDO qui lui assure les services de persistance. Les SGBDO incluent de véritables langages de programmation objet avec lesquels il est

1. Object Data Management Group : à ne pas confondre avec l'OMG !

possible de coder une grande partie des applications. Dans le monde Java par exemple, la confusion reste certaine sur l'affectation des tâches de gestion de la persistance, via *Java Data Objects* par exemple (voir section 1.10, « Webographie ») qui est la fonte pure et simple du modèle de l'ODMG dans Java, ou via un SGBDO de type SQL99 (aussi appelé « modèle objet-relationnel ») comme Oracle 9i avec des technologies connexes comme *Java DataBase Connectivity* (JDBC) ou encore SQL#J. Nous revenons sur ces points d'une manière concrète dans les études de cas des chapitres 4 et 6 surtout.

En résumé, il devient parfois difficile de répartir les objets. Côté client ? Côté serveur ? Pratiquement, doit-on implémenter un type d'objet UML en Java ? En SQL99 ? Les deux ? Dans un souci d'illustration de la persistance de haut niveau, nous nous intéressons ici plus spécialement à SQL99 et montrons comment, par certains aspects, il se conforme au modèle objet de l'OMA (voir section 1.2.4). Voici un exemple en SQL99 :

```
CREATE OR REPLACE TYPE Optical_amplifier AS OBJECT(network_address NUMBER);
CREATE OR REPLACE TYPE BU_module AS OBJECT(super Optical_amplifier);
CREATE TYPE Arr_BU_module AS VARYING ARRAY(2) OF REF BU_module;
CREATE OR REPLACE TYPE BU_subsystem AS OBJECT(bu_module Arr_BU_module);
```

Le code qui précède concerne l'étude de cas télécoms du chapitre 3. Le type *Optical_amplifier* représente la notion d'amplificateur de signaux dans un réseau sous-marin à fibre optique. Par manque de support pour décrire l'héritage en SQL99, *BU_module* qui dans le schéma UML du chapitre 3 hérite d'*Optical_amplifier*, est créé via un attribut nommé *super* et de type *Optical_amplifier*. Un objet *BU_subsystem* composé (agrégation UML) d'exactement deux *BU_module* a un attribut appelé *bu_module* qui est un tableau de deux références (pointeurs) sur des instances de *BU_module*. Pour information, le champ *super* n'est lui pas une référence mais fait qu'un objet *BU_module* « embarque » complètement un objet *Optical_amplifier*. Cela est concrètement fait dans des tables établies en fonction de tous ces types.

1.5.3 Réflexion (*reflection*)

La réflexion est un mécanisme en général mal connu des langages de programmation objet, mais il est essentiel par exemple, dans le fonctionnement des modèles de composants technologiques tels que *JavaBeans* ou *Enterprise JavaBeans* (EJB). Ce n'est pas aussi un hasard si C# supporte un haut niveau de réflexion dans le cadre de la manipulation de la plateforme .NET. La réflexion est par ailleurs l'exact équivalent de la notion de métamodélisation en UML. L'idée est de conserver des informations ou métadonnées sur les objets en mémoire, leur type particulièrement. À un niveau primitif, ce sont les RTTI (*Run-Time Type Information*) de C++ qu'il faut activer via une option de compilation spécifique. La mise en œuvre est alors la suivante :

```
#include <typeinfo.h>
…
Individual *i;
```

```
i = …; // affectation d'une référence sur une instance d'une sous-classe
       // de Individual, e.g. Female
const type_info& ti1 = typeid(i); // on détermine dynamiquement
                                  // le type de l'instance
if(ti1 != typeid(Female*)) throw bad_typeid(ti1.name());
```

Smalltalk va bien au-delà de cette simple approche C++ en proposant un système hautement réflexif qui a fait la célébrité du langage. En Smalltalk-80, un envoi de message (appel de fonction membre) est lui-même un objet et donc l'instance d'une classe prédéfinie représentant cette réflexion. Le système de réflexion de Java bien moins étoffé que celui de Smalltalk, offre néanmoins l'interface de programmation *java.lang.reflect* qui est riche. Elle peut être illustrée comme suit :

```
Temperature t = new Temperature();
Class c = t.getClass();
java.lang.reflect.Method m[] = c.getMethods();
```

En poursuivant ce code via un balayage du tableau *m* contenant des instances de *Method*, *i.e.* des fonctions membres, on peut dynamiquement identifier une fonction membre par son nom et plus largement sa signature (nombre d'arguments, types des arguments…), puis l'appeler. En temps normal, un appel statique aurait été :

```
t.a_given_method();
```

Quel est alors l'intérêt de la réflexion ? L'exemple précédent illustre mal qu'il est fréquent de récupérer des objets dont on ne connaît rien ou très peu de chose (c'est particulièrement vrai en programmation distribuée). Dans l'exemple précédent, on sait bien que *t* est de type *Temperature*. Dans d'autres situations, la réflexion est utile. De plus, c'est un mécanisme adéquat pour faire collaborer des codes de provenances différentes. Sans entrer ici dans les détails, la bibliothèque *PauWare.Statecharts* mise en œuvre dans les chapitres 5 et 6 est totalement adossée à *java.lang.reflect*. L'idée est de mettre en œuvre du code transformationnel (du code de calcul) à l'intérieur d'un automate UML piloté par du code de contrôle (gestion des transitions entre états pour l'essentiel). Le code de contrôle appelle le code transformationnel via l'API (*Application Programming Interface*) de réflexion.

1.6 ANALYSE ET CONCEPTION PAR OBJETS

Cette section est une introduction à l'analyse et à la conception objet, c'est-à-dire le besoin de dépasser la programmation pour finalement arriver, après nombre de méthodes (dont OMT et celle de Booch pour les plus marquantes), à UML. Le terme « méthode » recouvre langages d'expression, graphiques le plus souvent, et modèles objet. Il englobe aussi la codification des étapes de production des modèles objet, de manière à former des modèles de processus de développement logiciel, c'est-à-dire des cadres reproductibles pour guider au mieux un développement objet.

Le danger, ce dont il faut être conscient

Un défaut majeur de la réflexion est qu'elle court-circuite le contrôle de type statique et aboutit à un contrôle dynamique (*i.e.* à l'exécution) de type Smalltalk. Les erreurs apparaissent donc à l'exécution alors que certaines[a] auraient pu être détectées à la compilation. Par ailleurs, on peut aussi critiquer la performance dégradée ainsi que la sécurité ou encore la confidentialité mises à mal. Seules les fonctions membres publiques peuvent être appelées par réflexion, ce qui donne un premier degré de sécurité : l'encapsulation initiale n'est pas brisée. De manière plus générale, l'accès à des informations exhaustives sur un objet est un point discutable.

a. «Certaines», car le système de typage de Java est suffisamment pauvre (transtypage ou *casting* à outrance) pour faire perdurer bon nombre de problèmes à l'exécution (levée de l'exception *ClassCastException*).

1.6.1 Concepts et principes

Le foisonnement dans les années quatre-vingt-dix[1] des méthodes objet a amené l'OMG à fournir un cadre normatif des concepts propres à l'analyse et la conception objet [20]. Il apparaît sous sa forme la plus synthétique à la figure 1.11. De la gauche vers la droite, les concepts sont de plus en plus spécialisés : par exemple, l'agrégation est un type de relation qui est un concept de caractérisation de la structure d'un objet, etc. L'idée était de contraindre ce domaine où la multiplication des méthodes allait perturber, voire rebuter, les développeurs objet, au lieu de les convaincre d'investir dans la technologie objet. Le monde industriel soucieux d'une stabilisation et d'une arrivée à maturité de cette technologie, était demandeur de standards pour pérenniser toute adhésion à « l'objet ».

Il y a toujours eu une ambiguïté dans les méthodes d'analyse et de conception à objets, et donc dans les notions présentes en figure 1.11. En voulant monter d'un niveau d'abstraction pour s'affranchir des langages de programmation, les constructions de modélisation de ces méthodes, au cas où elles étaient mal ou insuffisamment formalisées (cas de la figure 1.11), laissaient libre champ au programmeur pour les interpréter, puis les implanter avec des sens différents d'une personne à une autre. Concrètement, qu'est-ce que précisément une relation d'agrégation par exemple ? Comment l'implanter selon une règle systématique, en C++ par exemple ? Les ateliers de génie logiciel (AGL) outillant les méthodes reproduisaient ce schéma avec parfois des générateurs de code dont le produit était plutôt éclectique. À contre-courant, certaines méthodes dont OMT fut l'archétype (bientôt suivie par UML), offrirent graphiquement l'équivalent d'un mécanisme de programmation. Par exemple, la généricité avec les classes *template* de C++, bénéficie d'une construction de modélisation dédiée en UML (voir chapitre 2). Où est alors l'abstraction ? Si une méthode se borne à cela, elle n'est qu'un moyen d'exprimer graphiquement du code d'un langage de programmation, par ailleurs plus formel car contraint par une gram-

1. Quarante-trois méthodes ont été listées et succinctement présentées dans le document [2].

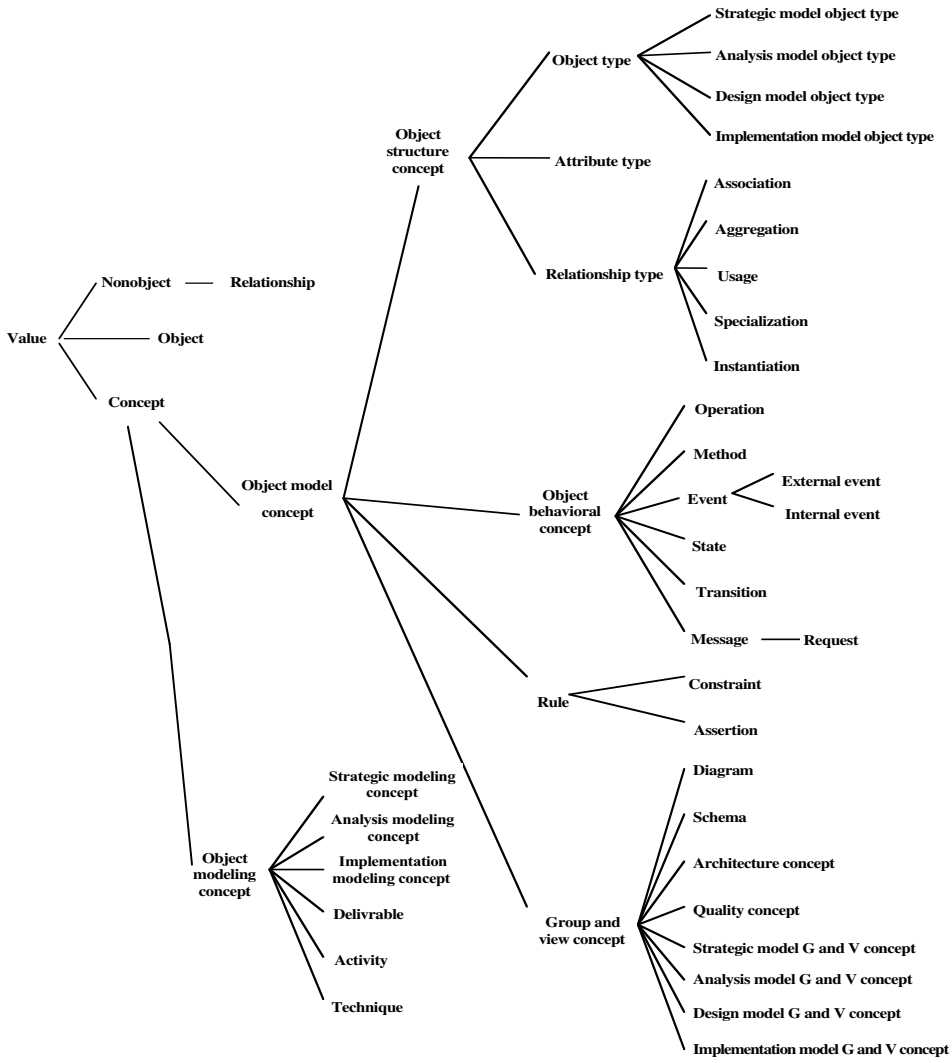


Figure 1.11 — Ensemble des concepts nécessaires et suffisants pour l'analyse et la conception objet selon l'OMG.

maire. La dérive fut à l'époque de réduire la production de modèles objet à de la fabrication de documentation, puis de façon plus intéressante, à réaliser des analyseurs lexicaux (voire grammaticaux) de code pour remonter ce code sous forme de modèles à des fins de documentation toujours, voire de compréhension, d'analyse ou autre.

Outre une meilleure formalisation des concepts même d'un langage de modélisation (par métamodélisation particulièrement, cas de UML, ou sur des bases mathématiques : VDM++, Object-Z…), il est apparu nécessaire d'avoir de « vrais isomorphismes » entre les modèles. En d'autres termes, un transformateur de modè-

les où le code n'est qu'un modèle opérationnel et où la fonction de transformation et sa réciproque sont parfaitement connues. Généralisée aux composants logiciels et aux architectures distribuées du moment, cette vision est à l'origine du MDA (*Model-Driven Architecture*) de l'OMG et plus ouvertement de l'ingénierie « drivée » modèle (MDE, *Model-Driven Engineering*) ou ingénierie des modèles plus simplement, là où UML par essence excelle !

1.6.2 Processus

Comme écrit précédemment, les méthodes d'analyse et de conception objet incluaient le plus souvent un processus (ou déroulement) codifié d'étapes propres et surtout *typées* pour le développement objet. Depuis, UML s'est clairement démarqué du sujet puisque les aspects processus y sont délibérément occultés. Bien que traiter des modèles de processus objet (voir section 1.10 pour des pointeurs sur « les » deux modèles de processus le plus compatible à UML : le RUP et le SPEM) dépasse le cadre de cet ouvrage, il nous a semblé nécessaire de donner les fondamentaux et plus spécialement, encore une fois, la norme proposée par l'OMG dans l'ouvrage de Hutt [20]. Cette norme définit un certain nombre de niveaux de préoccupation apparaissant dans la figure 1.12. Il y a différents types de modélisation objet. La modélisation objet stratégique par exemple est plutôt hors du champ du développement logiciel proprement dit : cela recouvre plus volontiers le domaine du *business process engineering*. Dans les autres types, également moins connus parce que non associés aux phases d'analyse ou de conception, *Coordination and reuse* couvre la mise en cohésion des activités, le partage des ressources logicielles interprojets, ce qui d'un point de vue réutilisation, correspond à l'extraction et à la capitalisation de composants logiciels d'un projet en vue de la sélection et de l'intégration de ces mêmes composants dans un autre projet.

Une fois arrêtées et clairement définies, ces activités de modélisation objet ainsi que leur usage dans le cycle de vie du produit (*i.e.* le logiciel à produire ou *Full definition of system* dans la figure 1.12), l'OMG a considéré le processus de développement objet comme devant avoir une stratégie d'itération incrémentale (figure 1.13) et une stratégie de progression additive.

Dans la figure 1.13, en bas, un carré se forme dans le temps. Chaque morceau du carré est un incrément qu'il faut fusionner (assembler en fait) avec le résultat du cycle précédent. Cet incrément est un actif logiciel (au sens « bien logiciel » aussi appelé *software asset* dans la documentation de l'OMG) à part entière, c'est-à-dire réutilisable et partageable entre divers projets. C'est un regroupement de types d'objet matérialisant une partie cohérente du système. Un cycle consiste donc globalement à faire évoluer un incrément d'une version A à une version C jusqu'à une version « qualifiée » qui complète le système à construire, puis à passer à une autre partie.

L'approche de progression additive, quant à elle, s'oppose à l'approche de progression transformationnelle. L'approche additive ne considère qu'un seul modèle qui traverse toutes les phases du développement et se détaille au fur et à mesure de

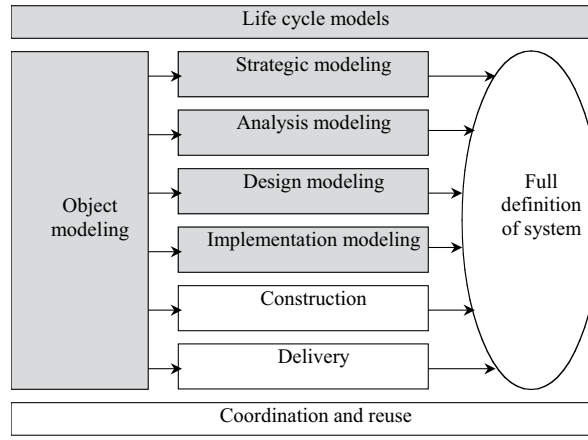


Figure 1.12 — Différents types de modélisation objet (stratégie, analyse, conception…) et cycle d'enchaînement de la production des modèles (du haut vers le bas) selon l'OMG.

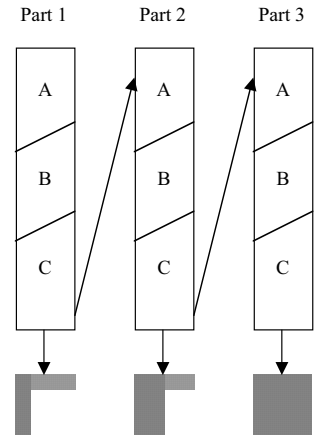


Figure 1.13 — Approche incrémentale de l'OMG.

la prise en compte de contraintes opérationnelles. À l'époque donc, l'OMG était loin de choisir le MDA/MDE qui établit plusieurs modèles dérivant les uns des autres selon une logique bien définie : PIM (*Platform-Independent Model*) puis PSM (*Platform-Specific Model*) pour les deux modèles phares du MDA. Il nous a ainsi semblé nécessaire d'aller au-delà de ce cadre aujourd'hui assez « théorique » finalement[1], en montrant empiriquement et intuitivement à l'aide d'un exemple ce qu'est un modèle d'analyse, ce qu'est un modèle de conception et comment enchaîner les activités de modélisation, tout cela dans l'esprit du MDE. La figure 1.14 donne le cadre général sans préjuger d'un modèle de processus particulier, d'une compatibilité

1. Spzyperski *et al.* [32, p. 260] ont des mots assez durs sur le MDA, entre autres : « *Effective bridging of divergent technologies and platforms at the level of MDA remains a largely untackled challenge.* »

MDA ou non, etc. Bien qu'essentielles et incontournables, la phase amont d'ingénierie des besoins ainsi que les activités de vérification et de validation de modèles sont, pour des raisons pédagogiques, occultées de notre exemple. Elles sont considérées de tout premier plan dans la figure 1.14 et illustrées dans des études de cas plus appropriées : chapitre 4 pour l'ingénierie des besoins, chapitre 5 pour la vérification et la validation.

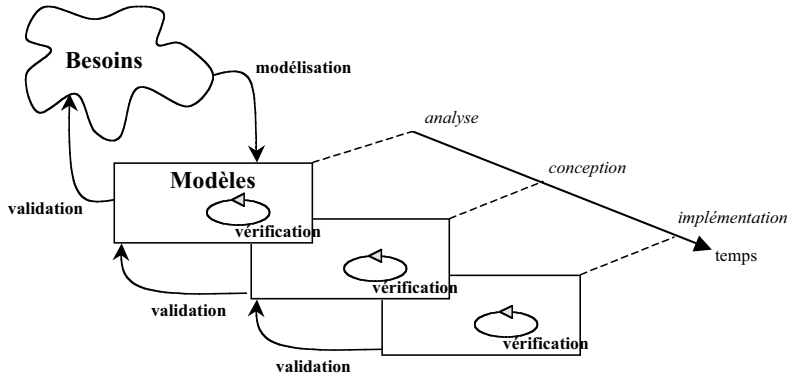


Figure 1.14 — Processus d'ingénierie des modèles.

1.6.3 Distinction entre analyse, conception et programmation par objets

Dans les sections 1.6.3 et 1.6.4, nous donnons une vision la plus limpide et pédagogique possible, et donc délibérément caricaturale des différences entre analyse, conception et programmation ainsi que l'enchaînement cohérent de ces trois activités dans le processus de développement logiciel objet. L'analyse orientée objet exprime plus ou moins formellement des besoins. D'un point de vue méthodique, elle ne s'attache pas aux techniques de capture, de découverte, de consolidation ou encore de validation des besoins, qui à notre sens relève d'une activité amont : l'ingénierie des besoins (voir chapitre 4 avec une étude de cas représentative et illustrative de l'ingénierie objet des besoins). Plus précisément, l'analyse objet a pour livrable une représentation *informatique* (figure 1.15) qui d'une part améliore une expression des besoins qui existerait en langage naturel, et ce même si le langage graphique utilisé manque d'assises formelles. D'autre part, la représentation informatique (abstraite

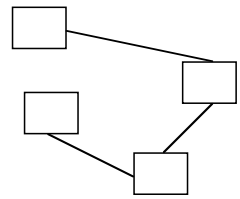


Figure 1.15 — Modèle d'analyse, caricature.

par nature), bien qu'indépendante des considérations d'implémentation, s'appuie sur un langage dont la traduction vers un langage de programmation (pas forcément immédiate mais en plusieurs phases) est aisée. Cette forte attente a présidé à la naissance des méthodes objet qui se sont substituées aux méthodes dites fonctionnelles (Merise, SADT ou autres) dont les modèles produits se transformaient mal en C++, Eiffel ou autres.

La conception objet est, en terme d'activité, souvent vue comme une expansion du modèle d'analyse (figure 1.16). Cette vision idéalisée se heurte cependant à la réalité du terrain et à l'hétérogénéité des applications ainsi qu'à l'éclectisme des plateformes supports. Dans un système embarqué par exemple, les contraintes matérielles comme une mémoire limitée peuvent engendrer des révisions notables des classes issues de l'analyse, dans leur contenu, mais aussi dans leurs relations et interactions. Cette observation n'est pas originale car dans un autre domaine comme les bases de données, la normalisation des relations est souvent remise en cause (la « dénormalisation ») sur des considérations de performance.

Si l'on reste dans le cadre idéal, les boîtes grises de la figure 1.16 font référence à des entités informatiques ayant une réalité opérationnelle (typiquement, une classe d'une bibliothèque). La conception dit que pour que l'entité conceptuelle représentée par la boîte blanche (entité d'analyse) existe un jour dans un logiciel, il faut l'associer (simple réutilisation) avec une ou plusieurs entités grises qui ne sont jamais que des moyens de mes objectifs. Jusqu'où écrire et détailler cette association ? Par essence, elle doit rester abstraite. Par exemple, écrire un lien d'héritage entre une boîte blanche et une boîte grise reste conforme à l'esprit de la modélisation objet. Au contraire, décrire des relations et interactions dont l'implémentation dans un langage à objets donné serait impossible, est déplacé mais non rédhibitoire. La nuance est ici qu'entre un modèle de conception préliminaire de nature plutôt abstraite et un modèle de conception détaillée plus concret, ou en tout cas plus proche de l'environnement et des contraintes d'exécution, il y a une voie étroite possible à explorer.

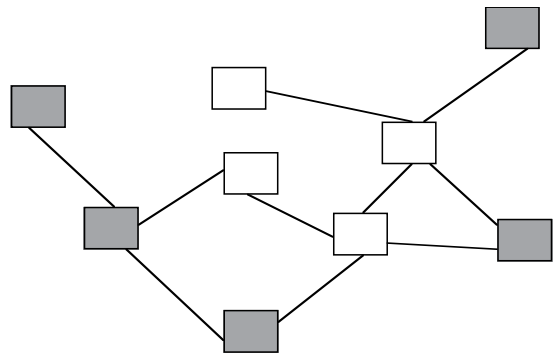


Figure 1.16 — Modèle de conception dérivant d'un modèle d'analyse.

La figure 1.17 boucle la boucle au sens où il faut écrire maintenant via du code (symbolisé par « … » dans la figure 1.17) les relations et interactions de façon complète. Le code dans les boîtes est « sain » : la prise en charge (post-spécification) des besoins naît dans le code. Certaines boîtes grises ne comportent pas en leur sein de « … » : elles sont réutilisées comme telles. Le plus étrange sont les « … » dans les boîtes grises : ils figurent la nécessaire adaptation d'un existant dont la réutilisation ne s'opère qu'à 90% par exemple et dont 10% sont à écrire pour obtenir la nécessaire correspondance avec les entités d'analyse.

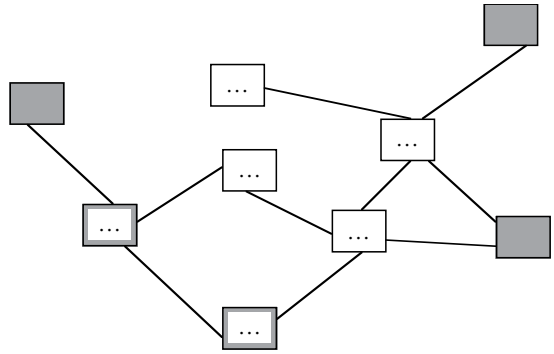


Figure 1.17 — Modèle d'implémentation dérivant d'un modèle de conception.

Ici, n'apparaît nulle part la réitération de ce cycle. Est-ce que les boîtes blanches deviennent des boîtes grises ? Quand ? Comment ? Où ? Dans le même logiciel ? Dans un autre logiciel ou une autre famille de logiciels (approche ligne de produits) ? D'autres questions sont relatives à la maintenabilité. Les modèles ont-ils une base pérenne lorsque les besoins évoluent ou quand il faut réparer les bogues ? La réponse à toutes ces questions est éminemment clé et facteur d'adhésion à/rejet de la modélisation objet. En d'autres termes, si les modèles produits n'ont pas de qualité avérée et mesurable, il est naturel de s'intéresser à la qualité intrinsèque de l'objet.

Le danger, ce dont il faut être conscient

Notre démarche est celle d'un doux rêveur ! La pratique montre le plus souvent le contraire : les modèles sont plus ou moins « cassés » au fur et à mesure de la réalisation finale de l'application. Même si le MDE, dans ses fondements, veut combattre la production de modèles sans lien entre eux, au moment de la rédaction de cet ouvrage et comme le dit Szyperski dans [32], cela reste à valider à grande échelle.

1.6.4 Une expérimentation de développement par objets

Dans cette section, nous allons vous présenter le plus intuitivement possible le développement objet et surtout de forcer le trait sur les aspects réutilisation et mainte-

nance, pour ensuite critiquer tout ou partie de l'approche suivie. L'idée est de s'appuyer sur le cheminement de la section précédente mais cette fois avec un exemple réel. L'exemple est volontairement caricatural et nous prévenons d'ores et déjà les lecteurs experts du caractère naïf et de la qualité médiocre des premiers modèles. Néanmoins, l'observation montre que beaucoup d'étudiants en informatique tendent à produire, puis à coder ces premiers modèles, qui s'avèrent ensuite inadéquats. Le lecteur regardera donc l'exemple avec plus ou moins de recul selon son niveau d'expertise.

Cahier des charges

Le cahier des charges de cette étude de cas est donné par un rapport « écran » sur deux années (2000 et 2001 en figure 1.18). Au sein d'un tableau annuel (ou carte d'activité), en 1ère ligne apparaissent les douze mois d'une année, en 1ère colonne des clients et finalement le chiffre d'affaires (CA) réalisé par client et par mois. En bas de chacun des deux tableaux figurent alors deux CA indicateurs calculés comme la

CA (en K€) *Année 2000*	**Janvier**	**Février**		**Décembre**
André	**3,3**	**0**	**…**	**6,4**
Barbier	**0**	**5,6**	**…**	**8,4**
Royer	**9,7**	**6,8**	**…**	**2,3**
…	**…**	**…**	**…**	**…**
CA indicateur (norme quadratique) : 17,3 **CA indicateur (norme maximum) : 9,7**				OK

CA (en K€) *Année 2001*	**Janvier**	**Février**		**Décembre**
André	**0,8**	**0**	**…**	**2,9**
Barbier	**1,4**	**6,8**	**…**	**6,9**
Royer	**2,9**	**3,5**	**…**	**0**
…	**…**	**…**	**…**	**…**
CA indicateur (norme quadratique) : 11,2 **CA indicateur (norme maximum) : 6,9**				OK

Figure 1.18 — Résultats d'activité et indicateurs économiques associés (deux dernières années glissantes).

norme quadratique (racine carrée de la somme des carrés des CA), respectivement, la norme maximum (valeur absolue du maximum des CA), de la matrice du tableau.

Une synthèse bisannuelle (figure 1.19) s'ajoute au rapport d'activité de la figure 1.18. Elle est composée de deux tableaux, le premier qui somme les CA par client et par mois et le second qui fait la différence (année 2001 – année 2000 par exemple, figure 1.19). Dans cette synthèse, à chaque tableau est associé un coefficient de pondération prédéfini λ qui multiplie les mêmes indicateurs économiques que précédemment, à savoir, les normes quadratique et maximum pour le tableau de sommation et pour le tableau de différence.

ΣCA (K€) *2000 + 2001*	Janvier	Février		Décembre
André	4,1	0	…	9,3
Barbier	1,4	12,4	…	15,3
Royer	12,6	10,3	…	2,3
…	…	…	…	…

ΣCA indicateur (λ * norme quadratique) : 13,81
ΣCA indicateur (λ * norme maximum) : 7,65
λ (coeff. pondération *2000/2001*) = 0,5 OK

ΔCA (K€) *2001 - 2000*	Janvier	Février		Décembre
André	-2,5	0	…	-3,5
Barbier	1,4	1,2	…	-1,5
Royer	-6,8	-3,3	…	-2,3
…	…	…	…	…

ΔCA indicateur (λ * norme quadratique) : 2,79
ΔCA indicateur (λ * norme maximum) : 2,04
λ (coeff. pondération *2000/2001*) = 0,3 OK

Figure 1.19 — Cumul et différence bisannuelles ainsi qu'indicateurs économiques associés.

Analyse

L'analyse faite en UML (figure 1.20) fait apparaître la classe ou boîte *Activity card* dont chaque instance représente un tableau de la figure 1.18. Pour simplifier, on omet les données de base qui sont les CA par client par mois pour focaliser sur trois propriétés de cette classe, à savoir *year()*, *turnover indicator-qn()* (norme quadratique) et *turnover indicator-mn()* (norme maximum), toutes trois vues comme des opérations ou services. On voit en effet les propriétés plutôt comme des services que des champs (voir discussion sur le principe de référence uniforme au chapitre 2). Typiquement, la fonction *year()* sur *Activity card* retourne un entier alors qu'en conception, on utilisera probablement un champ de type *Time-Date* pour stocker l'année. En implémentation, on activera l'extraction de l'année sur une date complète mais à ce niveau d'abstraction (analyse), on occulte délibérément ces considérations de programmation en n'utilisant que des opérations. Les champs donnent en effet trop l'idée de la façon dont on code « en dur ».

En résumé, dans l'esprit de l'analyse, on essaie de s'abstraire le plus possible « de la machine » pour mettre l'accent sur la bonne compréhension et intégration des besoins dans la spécification. En continuant ainsi, on obtient la classe *Biennal synthesis* qui correspond à six propriétés observées dans la figure 1.19. En figure 1.20, le trait plein d'*Activity card* sur elle-même ainsi que la relation « trait pointillé » que nous allons étudier plus formellement dans le chapitre 2, disent que chaque couple d'instances d'*Activity card* formé correspond à une instance de *Biennal synthesis*. La contrainte en bas à droite de la figure 1.20 indique que les deux années propres à chaque composante du couple sont contiguës (*i.e.* on ne fait des synthèses bisannuelles que sur deux années qui se suivent).

À ce terme mais sans technique de preuve particulière, on peut imaginer que les besoins sont pris en compte. On ne s'intéresse pas aux contingences de restitution

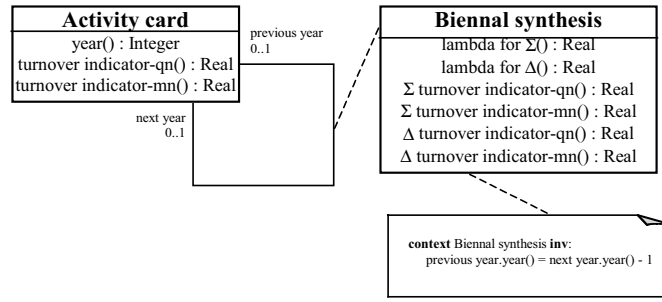


Figure 1.20 — Modèle d'analyse.

des données/résultats de calcul sur une interface utilisateur, de stockage en base de données ou autres qui s'imposent naturellement plus tard au moment de la finalisation de l'application.

Conception

L'approche instinctive centrée sur le résultat immédiat veut que l'on cherche des composants logiciels prédéfinis qui assurent des services de calcul de norme sur une matrice (c'est finalement le fond du problème de cette étude de cas) et ensuite, comment connecter ces composants à des versions codées des classes d'*Activity card* et *Biennal synthesis* de la figure 1.20. Prenons le cas le plus pénalisant : ces composants n'existent pas. Dans ce cadre, une approche plutôt réductrice veut que l'on implante les calculs de norme dans les composants *Activity card* et *Biennal synthesis*, ce qui relève d'une mauvaise logique : ces calculs sont suffisamment généralistes pour être associés à un composant technique dont la vocation est d'être réutilisé dans d'autres logiciels que le logiciel d'intérêt.

Ces véritables préoccupations de conception sont des éléments de réponse aux questions que l'on se posait dans la section 1.6.3. On conçoit en vue de réutiliser en engendrant, sur le moment, des coûts supplémentaires mais en espérant parallèlement l'amortissement au plus tôt. Dans cet esprit, on construit le composant *Vector* de la figure 1.21 ou un concurrent comme celui de la figure 1.22.

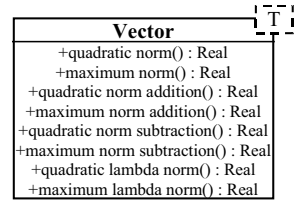


Figure 1.21 — Composant *Vector* avec éclatement des calculs de norme.

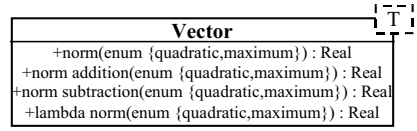


Figure 1.22 — Composant *Vector* avec paramétrage des calculs de norme.

Le composant de la figure 1.21 comporte huit services de calcul qui correspondent aux deux de la classe *Activity card* (on exclut la fonction *year()* bien entendu) et aux six de la classe *Biennal synthesis*. Le composant de la figure 1.22 fait dans l'astuce (on verra néanmoins que c'est une mauvaise voie) en paramétrant le calcul. On imagine alors le code : *si* type est le type *quadratic alors* tel calcul *sinon*…

Une version tierce, concurrente des deux précédentes, apparaît dans la figure 1.23. On a un modèle de conception détaillée en faisant référence à un composant prédéfini *std::vector* (le type *T* dans un carré en pointillé désigne un type générique paramétrant *std::vector*) de la bibliothèque standard de C++ (dans la section « Programmation » qui suit, on voit néanmoins que notre approche est aussi valide en Eiffel ou en Smalltalk qui ont un composant équivalent, le détail donné est donc non contraignant). Notre composant *Vector* hérite ensuite structurellement de ce composant pour bénéficier d'une zone de stockage pour la matrice. On fait supporter cependant, en toute opposition avec ce qui est fait dans les figures 1.21 et 1.22, quatre services dont un abstrait, *norm()*, qui rend de fait la classe *Vector* abstraite (en italique en UML). L'idée est simple et surtout capitalise des propriétés mathématiques bien connues : le calcul de la norme de la somme (ou de la différence) de deux vecteurs est indépendant du type de calcul de norme. Il en est de même pour une norme multipliée par un scalaire : $|\lambda| * ||V|| = ||\lambda * V||$.

La réutilisation par héritage d'une classe collection comme un tableau (*Vector* en Java ou *std::vector<T>* en C++) n'est jamais une bonne idée. Les développeurs C++ savent qu'ils peuvent verrouiller le lien d'héritage en figure 1.23 par la qualification du lien en *private* ou *protected* mais cela reste moins satisfaisant que la solution de la figure 1.24 : le rôle *implementation* côté *std::vector* est délibérément préfixé par un - pour bien signifier que *std::vector* est caché aux utilisateurs potentiels de *Vector*. C'est l'idée d'agrégation[1] évoquée précédemment mais qui est ici réductrice : nous préférons parler d'embarquement de composant dans un autre.

Il ne reste plus qu'un seul problème : quelle solution retenir ? Plus précisément, d'après ce que nous avons déjà dit, pourquoi le modèle de la figure 1.24 est-il le meilleur ?

Du point de vue de la réutilisabilité, nous avons extrait de la logique métier (fiches d'activité économique, synthèses bisannuelles…) un composant à vocation de réutilisation future que nous espérons intensive : *Vector*. Si ce n'est pas le cas, notons quand même que nous avons investi à perte puisque pendant que nous dis-

1. C'est réellement le terme utilisé dans l'ouvrage de Meyer [26] ou dans celui sur OMT [29].

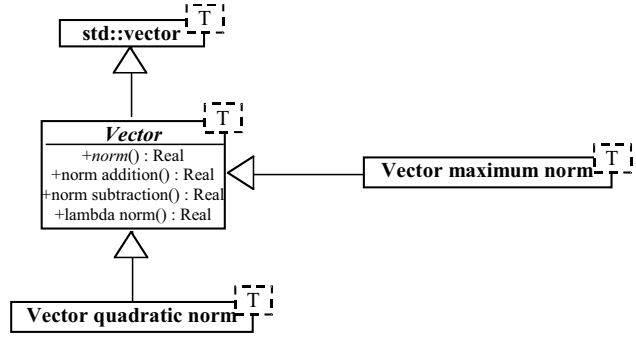


Figure 1.23 — Composant *Vector* avec typage des calculs de norme et réutilisation d'une classe « collection » par héritage.

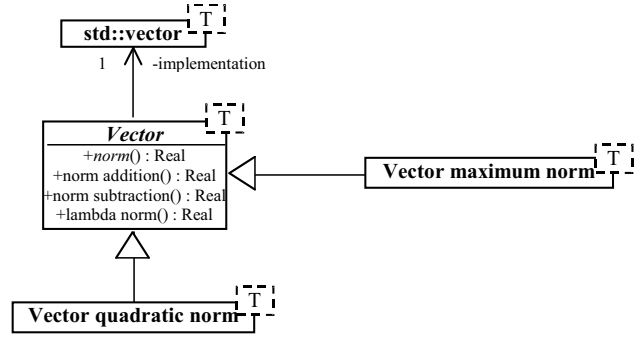


Figure 1.24 — Composant *Vector* avec typage des calculs de norme et réutilisation d'une classe « collection » par association (embarquement et masquage du composant *std::vector* dans *Vector*).

sertons sur *Vector*, l'application tarde à apparaître, les utilisateurs attendent. Seul un retour sur investissement court et conséquent peut justifier ces désagréments.

Du point de vue de la maintenabilité, imaginons un troisième type de calcul de norme, le modèle de la figure 1.22 est immédiatement rejeté. Le modèle de la figure 1.21 impose la réalisation de quatre nouveaux services et donc beaucoup de travail, c'est-à-dire un prix d'extension élevé, mais surtout, génère une activité de maintenance à forte charge. Rappelons que le nombre de services croissant et ceux-ci étant mal factorisés (ils ont des similitudes avérées), le processus de maintenance va s'alourdir. Les modèles de la figure 1.23 et de la figure 1.24 demandent la création d'une nouvelle classe qui hérite de *Vector* et la réalisation d'un seul service. C'est la solution la plus intéressante économiquement.

Pour la fiabilité, un gain est immédiat. En effet, le modèle de la figure 1.22 a un défaut extrême, il faut retoucher du code existant et tout bon programmeur que l'on soit, c'est la pire des choses qui puisse arriver. D'ailleurs, rien ne garantit que c'est le même programmeur qui fait les retouches d'où un accroissement de la probabilité de

bogues. Pour les autres risques plus classiques (par exemple la gestion mémoire de la matrice), le niveau de confiance dans *std::vector* s'il est quantifiable ou au moins reconnu, est un bon point. Sinon, dans la section intitulée « Programmation » on montre comment « blinder » le composant *Vector* car à taux de réutilisation élevé, il doit être sans reproche.

Après l'analyse de tous ces critères qualité, soyons honnêtes néanmoins sur la microarchitecture retenue (figure 1.24), le développement d'un graphe d'héritage par l'apparition de nouvelles classes n'est pas un phénomène anodin surtout si l'introduction d'une classe ne s'impose que par le besoin d'un seul nouveau service. Soulignons fortement qu'un risque économique est lié au modèle de la figure 1.24, si le composant *Vector*, certes techniquement sûr et sophistiqué et intellectuellement riche, n'est pas utilisé fréquemment dans le futur. Dans les faits, l'échec est, ou certainement sera, manifeste. Nous n'avons pas de religion à ce sujet. Au lecteur de disposer des éléments de réflexion et de construire « sa » solution qui normalement ne sera pas toujours reproductible d'un contexte à l'autre.

Assemblage

Le contrôleur de gestion, ordonnateur du logiciel, demande pourquoi ce logiciel n'est toujours pas livré et mis en exploitation. La raison est que nous avons fabriqué des composants *réellement* réutilisables et bien que cela ait pris plus de temps que prévu, les économies faites sur le long terme vont largement compenser les dépenses inhérentes au projet. Pour satisfaire ce contrôleur, il suffit de combiner classes d'analyse et classes de conception comme représenté à la figure 1.25.

En embarquant un composant logiciel *Vector* (association d'*Activity card* à *Vector* avec le rôle *implementation* masqué et de cardinalité *1* côté *Vector*), *Activity card* est instrumentée de manière à assurer ses services *turnover indicator-qn()* et *turnover indicator-mn()*. Dans la figure 1.26, une autre méthode s'appuie sur l'héritage multiple et a donc le défaut de limiter les possibilités en terme de langage de programmation : comment implanter le modèle de la figure 1.26 en Java par exemple ? Nous proposons d'aller dans le détail au sein de la section « Programmation » qui suit pour étudier le code « de glu » et surtout des implémentations dans des langages différents. Le modèle de la figure 1.26 pose aussi le problème de l'héritage répété qui fait hériter *a priori* deux fois de *Vector* et donc avoir une structure d'accueil de la matrice dupliquée pour rien.

Programmation

Puisque c'est en C++ que cette étude de cas a été le plus poussée, voici une partie du code de la classe *Vector* :

```
template<class T> class Vector {
   private:
   … // des mécanismes de gestion mémoire spécifiques sont nécessaires ici
   protected:
   … // des mécanismes de gestion mémoire spécifiques
     // sont aussi nécessaires ici
```

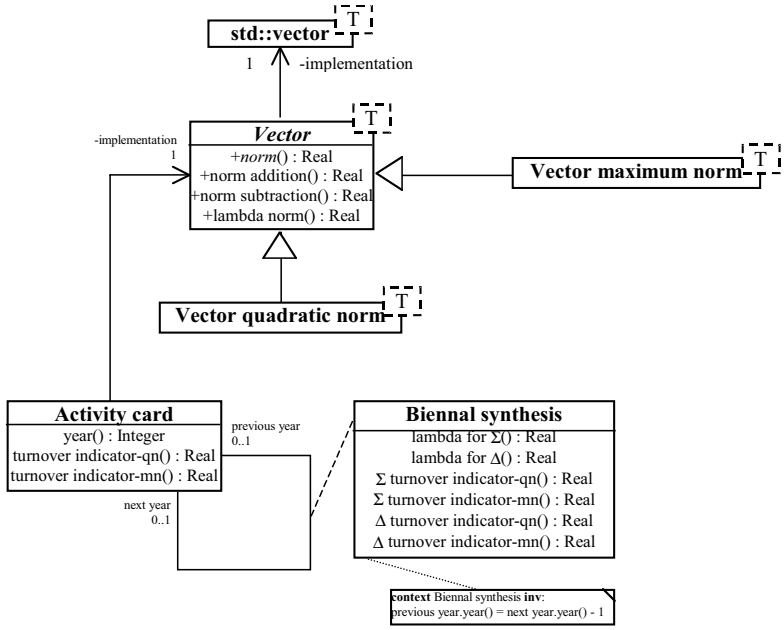


Figure 1.25 — Composition des classes pour obtenir les composants logiciels applicatifs.

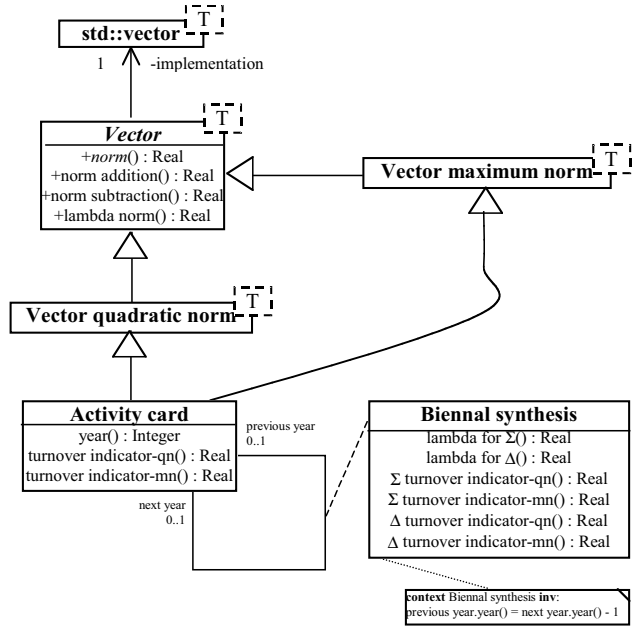


Figure 1.26 — Composition des classes, autre méthode, avec héritage multiple et répétée.

```
   vector<T> _implementation; // modèle de la figure 1.24
   public:
   … // constructeurs et destructeurs ici
   virtual double norm() const = 0; // calcul de norme abstrait,
                                    // dépend du sous-type
   double norm_addition(const Vector& v) const {return (*this + v).norm();}
   double norm_subtraction(const Vector& v) const {return (*this - v).norm();}
   double lambda_norm(const double lambda) const {return ::fabs(lambda)
          ➥ * norm();}
    // fonctions utilitaires (i.e. non-membres des besoins initiaux)
    // comme l'addition de deux vecteurs :
    Vector<T>& operator +(const Vector<T>&) const throw(std::logic_error);
    // code déporté
   …
};
```

On peut noter l'effort de recherche de la fiabilité via la levée d'exceptions standard de C++ (fonction opérateur +, exception *std::logic_error* : attention, cette notation est non normée). Un sous-type comme le calcul de norme quadratique est alors simple à implanter :

```
template<class T> class Vector_quadratic_norm : virtual public Vector<T> {
// l'héritage virtuel (virtual ci-avant) en C++ va empêcher d'avoir deux fois
// la matrice en cas d'héritage multiple et répété
   private:
   …
   public:
    …
   double norm() const; // le calcul de norme quadratique ici rendu concret
                        // et appelé dans les fonctions de la classe de base
   operator double();   // rend les scalaires compatibles aux vecteurs,
                        // très utile pour les calculs !
};
```

L'assemblage au sens du modèle de la figure 1.25 est direct :

```
class Activity_card {
   private:
     // l'implémentation retenue mais pas unique amène
     // (voir celle d'Eiffel plus loin)
     // pour chacun des 12 mois de l'année à avoir un objet
     // de type Vector<double> : les CA des clients
    Vector<double> _implementation[12]; // réutilisation du capital créé !
   public:
   … // constructeurs et autres ici
     // la fonction « conceptuelle » year() est finalement réalisée
     // sous forme de champ en lecture seule
   const unsigned short year;
     // Ci-dessous, un code glu intéressant mais aussi vital qui va permettre
     // d'accéder aux calculs de norme quadratique et maximum car on n'a pas
     // de lien direct avec Vector_quadratic_norm et Vector_maximum_norm
     // mais avec Vector (champ _implementation[12] ci-dessus)
   Vector_quadratic_norm<Vector_quadratic_norm<double> > asVQN() const;
     //code déporté
```

```
   Vector_maximum_norm<Vector_maximum_norm<double> > asVMN() const;
     // code déporté
   double turnover_indicator_qn() const; // Attendu par le contrôleur
                                         // de gestion !
   double turnover_indicator_mn() const; // Aussi attendu par le contrôleur
                                         // de gestion !
};
```

Enfin, on atteint le dernier type d'objet d'analyse :

```
class Biennal_synthesis {
   private:
     const Activity_card& _next_year;
     const Activity_card& _previous_year;
   public:
     … // les fonctions initialement demandées
};
```

Par simple curiosité, voici l'équivalent Eiffel (testé en ver. 2.3) de *Vector* :

```
deferred class Vector export norm,norm_addition,norm_subtraction,lambda_norm
   inherit
    BASIC; -- pour accéder à real_absolute
    FLOAT -- pour se conformer à FLOAT
     redefine infix "-",infix "+",infix "/",infix "*",prefix "-",prefix "+";
    ARRAY[FLOAT]
     rename Create as create_Array
   feature
     … -- constructeurs ou autres ici
     norm : FLOAT is
       require
       deferred
       ensure Result >= 0.0
       end; -- norm
       norm_addition(a_Vector : like Current) : FLOAT is
      …
       end -- norm_addition
       norm_subtraction(a_Vector : like Current) : FLOAT is
      …
       end; -- norm_subtraction
       lambda_norm(lambda : FLOAT) : FLOAT is
       …
       end; -- lambda_norm
       -- fonctions utilitaires (i.e. non-membres des besoins initiaux)
       -- comme l'addition de deux vecteurs :
       infix "+" (a_Vector : like Current) : like Current is
       …
      end; -- infix "+"
   invariant
   …
end -- Vector
```

La classe *Vector_quadratic_norm* est immédiate à construire. Comme en C++, le service *norm* est rendu concret et est implanté via un service interne *sygma* qui balaie tous les éléments de la collection pour les sommer (note : en C++, la fonction prédéfinie et générique *accumulate* de STL est utilisée). Le résultat est le suivant :

```
class Vector_quadratic_norm export
   repeat Vector,sygma{Vector_quadratic_norm}
   inherit Vector
      rename Create as create_Vector
      define norm
   feature
      … -- constructeurs ou autres ici
      sygma : FLOAT is
         …
      end; -- sygma
      norm : FLOAT is
        require
        local
          multiplication : like Current
        do
          multiplication := Current * Current;
          Result.Create;
          Result := multiplication.sygma;
          Result := Result ^ 0.5
        ensure
          Result >= 0.0
      end -- norm
   invariant
      …
end -- Vector_quadratic_norm
```

L'implémentation des classes d'analyse en Eiffel diffère de celle retenue pour C++. Quoi qu'il en soit, elle est masquée aux clients du composant *Activity_card* et n'est donc que de l'adaptation bas niveau. En l'occurrence, si l'on imagine la création d'une classe *Activity*, on obtient le code qui suit (la fiche activité est un vecteur de vecteur d'activités) :

```
class Activity_card export year,turnover_indicator_qn,turnover_indicator_mn
   feature
      _implementation : Vector[Vector[Activity]];
      … -- Etc.
end -- Activity_card
```

1.7 DE L'OBJET AU COMPOSANT

L'extension de la problématique objet à celle des composants logiciels résulte du constat que l'objet, par sa faible granularité mais aussi par d'autres facteurs, pose raisonnablement mal le problème de la réutilisation dans tout son ensemble et toute sa complexité. Les notions de patrons et canevas d'applications ont percé dans les années quatre-vingt-dix pour laisser plus volontiers la place à un concept plus générique et plus fédérateur : celui de composant. Ce qui est sous-entendu mais qui est essentiel, c'est qu'un composant est une entité logicielle *fait pour* la composition : « *Components are for composition.* » [32]. Autrement, c'est la dualité interface/implémentation et le mécanisme de déploiement résultant de la programmation distribuée qui concourent à la caractérisation complète de Szyperski qui fait le plus souvent

foi : « *A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third-parties.* » [32].

1.7.1 Patrons de conception et d'analyse (*pattern*)

Ce concept phare de la réutilisation a été déclaré thème de recherche stratégique de la recherche en programmation objet en 1996 dans [17]. Les patrons sont des formes abstraites (spécifiés en UML par exemple) ou concrètes (archétypes d'implémentation) d'usage général et dont l'adaptation à un contexte de développement spécial est techniquement directe et économiquement à moindre frais. Si Gamma *et al.* dans [13] ont définitivement popularisé les patrons de conception, l'idée avait été émise et admise bien avant, comme la vision de Coad dans [10] : « *What is a pattern? Pattern. A fully realized form, original, or model accepted or proposed for imitation: something regarded as a normative example to be copied; archetype; exemplar (…)* ».

Pour les patrons de conception, l'un des plus familiers est le *Composite* (revu, augmenté et corrigé[1] à l'aide d'UML[2] dans la figure 1.27).

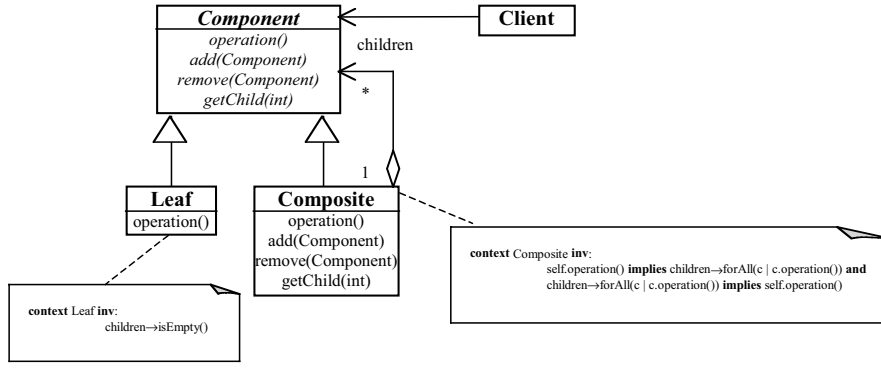


Figure 1.27 — *Composite* pattern de Gamma *et al.* classifié comme « structurel ».

Plus tard, Fowler dans [12] avec moins de succès que Gamma *et al.*, a tenté d'imposer les patrons d'analyse et donc de domaine ou « métier », et plus généralement la notion de *business object* devenue depuis celle de *business component* [3]. Le patron de la figure 1.28 qui nous est propre a été défini dans le cadre d'un système de gestion de vol pour la société Sextant Avionics devenue depuis Thalès Avionics. Il met en œuvre la relation de *metadata* d'OMT qui est aussi le patron dit *Item description* de Coad et est devenue depuis le stéréotype *«instantiate»* d'UML 2.x. Nous l'expliquons au chapitre 2.

Dans la figure 1.28, *Way point* désigne un point de passage lors d'un vol doté d'un identifiant international normalisé (*area code*). *Way and vertical point* « instancie »

1. À l'origine, le rôle *children* se situe vers *Composite* ce qui semble un contresens.
2. La spécification originale est en OMT.

un et un seul *Way point* et hérite d'un point de passage, vertical cette fois, qui fait référence au relief et aux couloirs de navigation aérienne en vigueur : *Vertical point*.

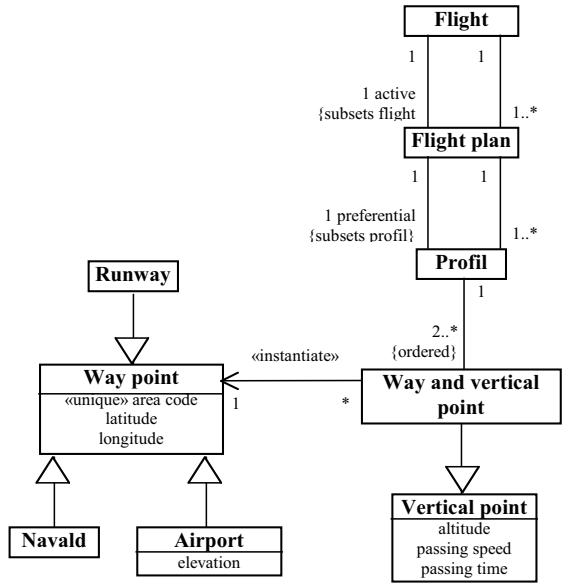


Figure 1.28 — Patron *Item description* défini dans [10] et appliqué ici à l'avionique.

Dans l'esprit de Szyperski, et plus précisément dans le titre de son livre qui souligne le dépassement de la technologie objet : « *Logiciel composant*, au-delà *de la programmation orientée objet* » (c'est nous qui soulignons), les composants logiciels sont le plus souvent[1] des agglomérats d'objets dont la structure et le comportement sont suffisamment bien connus et arrêtés qu'ils en deviennent canoniques (par exemple, patron *Factory* en programmation distribuée). Les patrons formatent donc les composants en termes d'objets.

1.7.2 Canevas d'application (*framework*)

La notion de canevas est simple à déchiffrer. Soit une application donnée, s'il est possible de voir tout ou une partie de l'architecture logicielle de cette application comme un élément (par analogie à l'automobile, disons un châssis) réutilisable, construire ladite application consiste à ajouter des muscles, des organes… sur un squelette. Ce squelette étant le même d'une application à une autre, construire une autre application cette fois, c'est tisser (le maintenant célèbre terme *weaving* en anglais) d'autres composants logiciels qui en font sa spécificité. Wirfs-Brock *et al.* ne disent pas autre chose dans [34] : « *Frameworks are skeletal structures of programs that must be fleshed out to build a complete application.* »

1. On peut aussi faire du composant logiciel hors du cadre objet.

Là encore, l'idée manque de fraîcheur, car rien de nouveau sous le soleil. Les langages de description d'architecture (ADL, *Architecture Description Language*) [24] ont depuis longtemps cerné le problème, introduisant cette notion de *framework* mais de manière complémentaire, celle de composant logiciel puis plus pertinemment celle de *connector* comme moyen du tissage. Cette dernière notion émerge d'ailleurs fortement en UML 2.x comme nous allons le voir au chapitre 2.

L'essentiel, ce qu'il faut retenir

L'idée de Johnson qui met en rapport composants logiciels, canevas d'applications et patrons [22] est maintenant tangible au sens où les canevas d'applications, décrits abstraitement via un ADL, sont étoffés en tissant des composants logiciels entre eux et dans l'architecture. Les patrons donnent les règles et formes standard de composition.

1.7.3 Objets, composants ou services ?

L'expression *service-oriented computing* et l'engouement, c'est un euphémisme, pour les *Web Service* pourraient augurer d'un basculement irréversible vers une nouvelle technologie de développement logiciel sans rapport , ou si peu, avec celles qui l'ont précédée. L'histoire de l'objet prouve le contraire (la programmation objet est structurée, que diable !). Les composants logiciels n'ont pas remplacé les objets, ils les utilisent mieux en les englobant, les canalisant dans tous leurs pouvoirs mais aussi leurs défauts. Les services feront la même chose avec les composants. Dans les faits, un *Web Service* n'est-il pas un client d'un *Stateless Session Bean* en technologie EJB ? Si, donc pas de révolution, juste une évolution.

1.8 CONCLUSION

Ce chapitre sur la modélisation objet est nécessaire avant d'aborder UML. Il arrive malheureusement souvent que des personnes traitant d'UML perdent de vue (ou ne l'aient jamais connu) le cadre du génie logiciel objet. Partons du principe un peu provocateur suivant : UML ne sert *a priori* à rien, n'apporte rien. En revenant aux fondamentaux de la qualité comme le fait ce chapitre, en se focalisant sur la maintenabilité, la réutilisabilité et la fiabilité, les questions, les envies, les exigences que nous avons à propos de la technologie objet doivent nous convaincre qu'un langage de spécification (formel au sens mathématique ou juste semi-formel pour concerner le plus grand nombre) doit s'imposer en amont de la programmation objet, composant et finalement orientée service.

Ne pas adhérer à cette idée, c'est aller à contresens de l'ingénierie de l'intégration, qui prend en compte la difficulté future d'assembler des briques hétérogènes de provenance multiple et de forme variée. UML peut raisonnablement être considéré comme un outil sérieux de cette ingénierie en devenir. Ne pas être conscient de

l'histoire et de l'environnement technologique d'UML, c'est l'utiliser pour ses programmes et dans ses projets informatiques avec une quasi-certitude d'échec. C'est pourquoi le leitmotiv de cet ouvrage est : « UML, ça fonctionne, mais en toute conscience de ses défauts et de ceux de la technologie objet. »

1.9 BIBLIOGRAPHIE

1. Abadi, M., and Cardelli, L. : *A Theory of Objects*, Springer (1996)
2. Barbier, F. : *Continuité du processus de développement logiciel objet*, mémoire d'Habilitation à Diriger des Recherches (1998)
3. Barbier, F. : *Business Component-Based Software Engineering*, Kluwer (2003)
4. Box, D., and Sells, C. : *Essential .NET – Volume 1 – The Common Language Runtime*, Addison–Wesley (2002)
5. Arnold, K., Gosling, J., and Holmes, D. : *The Java Programming Language*, Third Edition, Addison–Wesley (2000)
6. Booch, G. : "Object-Oriented Development", *IEEE Transactions on Software Engineering*, 12(2), (1986) 5-15
7. Booch, G. : *Object-Oriented Analysis and Design with Applications*, Second Edition, Benjamin/Cummings (1994)
8. Cattel, R., and Barry, D. : *The Object Data Standard: ODMG 3.0*, Morgan Kaufmann (2000)
9. Charbonnel, J. : *Langage C++ – Le standard ANSI/ISO expliqué*, seconde édition, Dunod (1999)
10 Coad, P. : "Object-Oriented Patterns", *Communications of the ACM*, 35(9), (1992) 152-159
11. Cook, S., and Daniels, J. : *Designing Object Systems – Object-Oriented Modelling with Syntropy*, Prentice Hall (1994)
12. Fowler, M. : *Analysis Patterns – Reusable Object Models*, Addison-Wesley (1997)
13. Gamma, E., Helm, R., Johnson, R., and Vlissides, J. : *Design Patterns – Elements of Reusable Object-Oriented Software*, Addison-Wesley (1995)
14. Goldberg, A., and Robson, D. : *Smalltalk-80 – The Language and its Implementation*, Addison-Wesley (1983)
15. Gosling, J., Joy, B., Steele, G., and Bracha, G. : *The Java Specification Language*, Second Edition, Addison-Wesley (2000)
16. Groß, H.-G., Atkinson, C., Barbier, F., Belloir, N., and Bruel, J.-M. : "Built-in Contract Testing for Component-Based Development" in *Business Component-Based Software Engineering*, Kluwer (2003) 65-82
17. Guerraoui, R. : "Strategic Directions in Object-Oriented Programming", *ACM Computing Surveys*, 28(4), (1996) 691-700
18. Harel, D. : "Statecharts: A Visual Formalism for Complex Systems", *Science of Computer Programming*, 8, (1987) 231-274
19. Hutt, A. : *Object Analysis and Design – Description of Methods*, Wiley (1994)

20. Hutt, A : *Object Analysis and Design – Comparison of Methods,* Wiley (1994)
21. Jézéquel, J.-M., and Meyer, B. : "Design by Contract: The Lessons of Ariane", *IEEE Computer*, 30(2), (1997) 129-130
22. Johnson, R. : "Frameworks = (Components + Patterns)", *Communications of the ACM*, 40(10), (1997) 39-42
23. Le Moigne, J.-L. : *La modélisation des systèmes complexes*, Dunod (1990)
24. Medvidovic, N., and Taylor, R. : "A Classification and Comparison Framework for Software Architecture Description Languages", *IEEE Transactions on Software Engineering*, 26(1), (2000) 70-93
25. Meyer, B. : *EIFFEL – The Language*, Prentice Hall (1992)
26. Meyer, B. : *Object-Oriented Software Construction*, Second Edition, Prentice Hall (1997)
27. Musser, D., Derge, G., and Saini, A. : *STL Tutorial and Reference Guide – C++ Programming with the Standard Template Library,* Second Edition, Addison–Wesley (2001)
28. Object Management Group : *Object Management Architecture Guide* (1992)
29. Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., and Lorensen, W. : *Object-Oriented Modeling and Design*, Prentice Hall (1991)
30. Sandén, B. : "Coping with Java Threads", *IEEE Computer*, 37(4), (2004) 20-27
31. Stroustrup, B. : *The C++ Programming Language*, Third Edition, Addison-Wesley (1997)
32. Szyperski, C., Gruntz, D., and Murer, S. : *Component Software – Beyond Object-Oriented Programming*, Second Edition, Addison-Wesley (2002)
33. Taivalsaari, A. : "On the Notion of Inheritance", *ACM Computing Surveys*, 28(3), (1996) 439-479
34. Wirfs-Brock, R., Wilkerson, B., and Wiener, L. : *Designing Object-Oriented Software*, Prentice Hall (1990)

1.10 WEBOGRAPHIE

iContract : www.reliable-systems.com/tools/iContract/iContract.htm
Eiffel : www.eiffel.com
Java Data Objects (JDO) : java.sun.com/products/jdo/ et jdocentral.com/
Object-Z : www.itee.uq.edu.au/~smith/objectz.html
Object Data Management Group (ODMG) : www.odmg.org
Rational Unified Process (RUP) : www-306.ibm.com/software/awdtools/rup/
Software Process Engineering Metamodel : www.omg.org/technology/documents/formal/spem.htm
Standard Template Library (STL) : www.sgi.com/tech/stl

2

UML *Structure*

2.1 INTRODUCTION

Le langage UML (*Unified Modeling Language*) a été d'abord défini dans trois versions préliminaires officieuses [4], [5] et [6] puis dans la version 1.1 [13], première version officielle, de la façon suivante : « *The Unified Modeling Language (UML) is a general-purpose visual modeling language that is designed to specify, visualize, construct and document the artifacts of a software system.* » Au moment de la rédaction de cet ouvrage, la version 1.5 [15] est la plus « officielle » alors que la version 2.0 est adoptée [16-17] mais pas tout à fait finalisée, et le tout dernier document paru est [18]. Il corrige la partie [17] de la version dite « adoptée ». La caractérisation d'UML dans la version 2.0 est : « *The Unified Modeling Language is a visual language for specifying, constructing and documenting the artifacts of systems. It is a general-purpose modeling language that can be used with all major object and component methods, and that can be applied to all application domains (e.g. health, finance, telecom, aerospace) and implementation platforms (e.g. J2EE, .NET).* » [16, p. 22].

Il est intéressant de noter l'apparition des mots *component*, *domain* et *platform* qui montrent la primauté du développement basé composant, et ce sous un angle technique, la prise en considération croissante des aspects métier à travers le terme « domaine » et finalement les serveurs d'applications, de composants et de services banalisant le développement logiciel pour les architectures ou plateformes distribuées : Internet ou autres. En outre, la référence à J2EE ou à .NET n'est pas sans opportunisme et montre que les années qui viennent vont voir la bataille de ces deux paradigmes industriels, bataille dans laquelle UML peut jouer un rôle essentiel.

Ce chapitre aborde la modélisation des aspects statiques des systèmes et systèmes logiciels, champ d'investigation appelé *Structure* dans UML 2.x ou encore *structural modeling*. Après avoir énoncé tous les types de diagramme d'UML 1.x et UML 2.x, nous nous concentrerons sur les *Structure Diagram* en tentant de présenter toutes les

notations ainsi que les changements en UML 2.x. Le chapitre 3 aborde la modélisation des aspects dynamiques aussi appelée *Behavior* en UML 2.x. Dans ces deux chapitres, nous utilisons une marque spéciale **UML 2.x**, en marge du texte, pour forcer le trait sur UML 2 justement, dès que des précisions s'imposent.

Le langage OCL (*Object Constraint Language*) [24] [19] est introduit par touches progressives. Une connaissance préliminaire et élémentaire n'est pas inutile, il suffit pour cela de se reporter à l'annexe de ce chapitre comportant une présentation concise. Une étude plus fouillée renvoie à l'ouvrage de base [24]. Dans la même logique, notre discussion sur UML, que ce soit sur la version 1.x ou sur la version 2.x peut sembler ardue. En cas de difficulté, nous conseillons une excellente introduction didactique à la spécification en général et à UML en particulier sous la forme de l'ouvrage de André et Vailly [1].

2.2 LES DIFFÉRENTS TYPES DE DIAGRAMME

En UML 1.x, ce sont :

- Use Case Diagram
- Class Diagram
- *Behavior Diagram*
  - Statechart Diagram
  - Activity Diagram
  - *Interaction Diagram*
    - Sequence Diagram
    - Collaboration Diagram
- *Implementation Diagram*
  - Component Diagram
  - Deployment Diagram

Dans la liste des types de diagramme proposés par UML 1.x, ceux apparaissant en italique n'ont pas d'existence réelle : ce sont des entités abstraites, non instanciables donc, comme les classes abstraites du chapitre 1. Par exemple, les *Behavior Diagram* n'existent qu'au travers des *Statechart Diagram* et des autres types de diagramme comportemental. Il est fondamental de comprendre qu'il y a un recouvrement important dans ces types. En d'autres termes, face à un problème de modélisation bien identifié, ils sont souvent des outils redondants et donc concurrents. Par exemple, les *Activity Diagram* introduisent une notation différente de celle des *Statechart Diagram* mais constituent un sous-ensemble sémantique : « *An activity diagram is a special case of a state diagram in which all (or at least most) of the states are action or subactivity states and in which all (or at least most) of the transitions are triggered by completion of the actions or subactivities in the source states*. » [14, p. 3-151] Toute la difficulté réside donc dans une classification mettant en exergue ceux qui sont importants et ceux qui le sont moins. Nous reprenons à ce titre la classification de Mellor [12] à laquelle nous

adhérons : *Essential Models* (*Class Diagram*, *Statechart Diagram*), *Derived Models* (*Activity Diagram*, *Sequence Diagram*, *Collaboration Diagram*) et *Auxiliary Models* (*Use Case Diagram*, *Component Diagram*, *Deployment Diagram*).

D'après Mellor : « *Essential Models capture the complete scope and behavior of the system and support model translation to code* », « *Derived Models show additional views of the essential models* » et « *Auxiliary Models augment the essential models* ». Pour compléter et commenter, disons que les modèles essentiels sont nécessaires et suffisants pour spécifier un système logiciel. Notons qu'un langage de modélisation objet peut remplir d'autres fonctions que celle du développement proprement dit (modéliser une organisation plutôt qu'un logiciel), et donc à ce titre, les *Statechart Diagram* par exemple, n'ont pas d'intérêt marqué et avéré pour l'activité d'ingénierie des besoins. Classer maintenant les *Use Case Diagram* comme des diagrammes non fondamentaux résulte aussi plus volontiers de leur manque d'assises formelles, leur absence de pertinence en conception (description d'architectures logicielles), etc. Pour les *Component Diagram* et *Deployment Diagram*, on trouve au contraire qu'ils conviennent mieux aux phases aval du développement qu'aux phases amont comme l'ingénierie des besoins. Bon nombre de types de modèle sont donc plutôt des outils de documentation que de modélisation, car leur nombre d'interprétations possibles est trop grand.

2.2.1 Intérêt et prédominance des types de diagramme

Les points de vue qui précèdent ne sont pertinents que s'ils sont confirmés par le terrain. Autrement dit, qu'en pensent les utilisateurs ? Par ailleurs, UML 2.x en tant qu'évolution d'UML 1.x entérine-t-il cette vision ? En fait, les changements profonds qu'apporte UML 2.x demandent réflexion. Avant de les aborder, il est intéressant de donner les résultats d'une enquête intitulée *Delphi study of the Institutional Review Board* réalisée par l'université du Nebraska-Lincoln concernant UML 1.x. Scindant les applications en trois catégories (temps réel, Web et gestion), les utilisateurs/experts ont voté et donné leurs préférences (*cf.* tableau 2.1).

Tableau 2.1 — Résultat de l'enquête *Delphi study of the Institutional Review Board*

	All applications	Real-time applications	Web applications	Enterprise applications
Use Case Diagram	Rang 2	Rang 4	Rang 2	Rang 2
Class Diagram	Rang 1	Rang 1	Rang 1	Rang 1
Statechart Diagram	Rang 4	Rang 2	Rang 4	Rang 7 (Rang 3 occupé par *Activity Diagram*)
Sequence Diagram	Rang 3	Rang 3	Rang 3	Rang 3

Le point de vue de Mellor et le nôtre s'opposent donc aux utilisateurs/experts d'UML 1.x sur *Use Case Diagram* et *Statechart Diagram*, les *Use Case Diagram* se

dégageant nettement en seconde position pour l'utilisateur/expert « lambda » alors que nous privilégions les *Statechart Diagram*. L'ensemble de cet ouvrage revient sur ce désaccord et tente bien entendu de justifier scientifiquement notre position.

2.2.2 Refonte des diagrammes en UML 2.x

Les changements conséquents apportés par UML 2.x sont (voir aussi figure 2.1) :

Diagram

- *Structure Diagram*
  - Class Diagram
  - Component Diagram
  - Object Diagram
  - Composite Structure Diagram
  - Deployment Diagram
  - Package Diagram
- *Behavior Diagram*
  - Activity Diagram
  - Use Case Diagram
  - State Machine Diagram
  - *Interaction Diagram*
    - Sequence Diagram
    - Collaboration Diagram
    - Interaction Overview Diagram
    - Timing Diagram

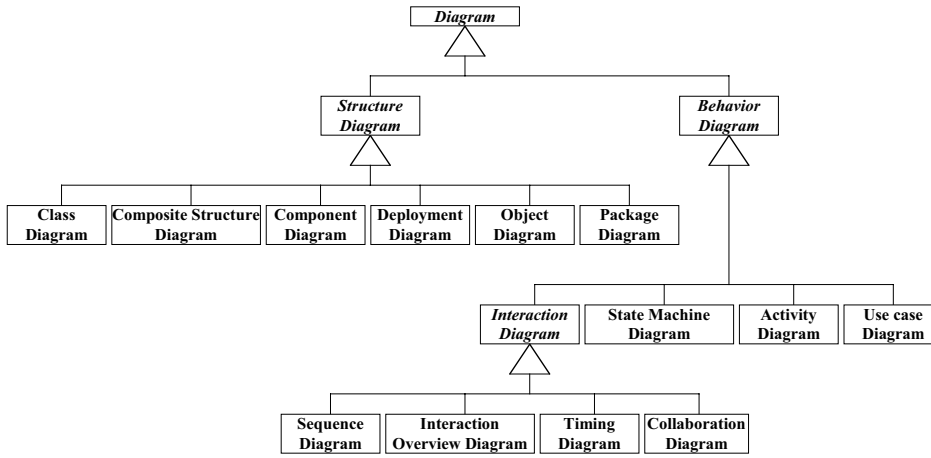


Figure 2.1 — Extrait du métamodèle d'UML 2.x présentant la réorganisation des types de diagramme.

On observe d'abord une hausse du nombre de types de diagramme. Ensuite, ces derniers sont organisés en types et sous-types (figure 2.1), faisant que ceux listés en italique ne sont que de grandes catégories, sans existence directe autre que celle de leurs sous-types.

2.3 DIAGRAMME DE CLASSE (*CLASS DIAGRAM*)

Le package *Classes* du métamodèle UML 2.x donne les éléments de modélisation de base et notamment le concept de « classe » de la technologie objet (figure 2.2). Pour information, on peut remarquer que ce package *Classes* recouvre bon nombre de notions développées dans la partie *Infrastructure* [16] (par opposition à *Superstructure* [17-18], qui correspond plus à UML lui-même). Ces notions sont plus spécialement présentes dans ce qui est appelé *Infrastructure Library*.

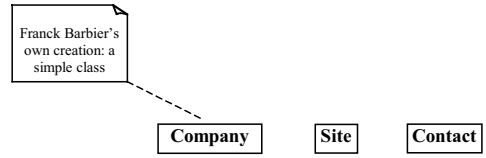


Figure 2.2 — Exemples de classes.

La description de classes (figure 2.2) s'opère dans des boîtes avec le nom de la classe en gras. Une note d'information préférablement appelée « commentaire » ou *comment* en UML 2.x peut être liée à un élément de modèle comme une classe via un trait pointillé sans orientation et avec un rectangle annoté dont l'angle en haut à droite, est replié vers l'intérieur.

2.3.1 Stéréotypes de base pour les classes

Il est difficile d'éviter l'évocation des stéréotypes dès lors que l'on présente UML. Un stéréotype est, dans tous les cas et dans l'esprit initial d'UML, une annotation s'appliquant sur un élément de modèle (*i.e.* une instance de *ModelElement*). Il n'a pas de définition formelle, mais permet de mieux caractériser des « variétés » d'un même concept.

Les stéréotypes fondamentaux associables aux instances du métaélément *Class* sont en UML 2.x :

- «metaclass» ;
- «type» ;
- «implementationClass».

Les autres stéréotypes qui peuvent être adjoints aux classes mais qui sont de moindre importance sont :

- «focus» ;
- «utility».

Les stéréotypes *«interface»*, *«realization»* et *«specification»* peuvent être associés au métaélément *Classifier* et jouent donc aussi un rôle privilégié car *Class* hérite (flèche dont l'extrémité est un triangle blanc en figure 2.3) de *Classifier*. La figure 2.3 fait apparaître une manière de décrire le package d'appartenance d'une entité : libellé *from…* sous le nom de la classe. Le métamodèle de la figure 2.3 donne la façon dont des notions fondatrices de la technologie objet comme *Class* ou encore *Interface*, étayent UML. Leur spécialisation via l'héritage amène à des concepts spécifiques comme *Component* (notion de composant logiciel du chapitre 1) par exemple.

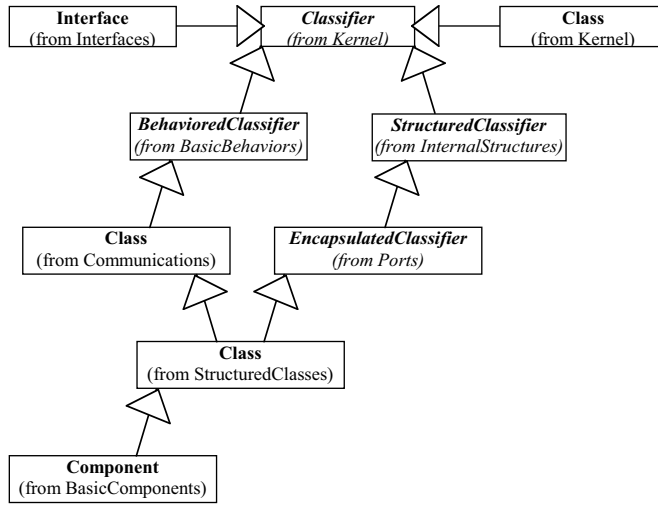


Figure 2.3 — Sous-hiérarchie partielle du métaélément *Classifier* en UML 2.x d'après [17].

Il persiste une confusion en UML 2.x sur l'usage cohérent et concomitant de ces stéréotypes. Il faut ainsi savoir que le stéréotype *«class»* est le stéréotype par défaut. Il est donc sous-entendu que dans des modèles comme celui de la figure 2.2, *Site*, *Contact* et *Company* sont des classes. De manière générale, toute boîte non stéréotypée dans un *Class Diagram* est implicitement une classe. Bien que le stéréotype *«class»* ne soit pour ainsi dire jamais utilisé dans tout le document de spécification d'UML 2.x [17], il provient d'UML 1.x et perdure dans UML 2.x. Il n'est en particulier pas présent dans la liste des stéréotypes obsolètes qui figure en fin de la documentation. En d'autres termes, dans la documentation d'UML 2.x, il n'est pas qualifié « caduc » mais vu qu'il n'est quasiment jamais utilisé, on pourrait croire qu'il a définitivement disparu ce qui n'est pas le cas. Quand l'utiliser ? Nous allons voir que dans les *Component Diagram*, il est utile pour bien différencier toutes les entités des-

sinées dont beaucoup sont marquées avec les stéréotypes *«component»* bien évidemment, et *«interface»*.

Le danger, ce dont il faut être conscient

Poursuivant notre analyse des stéréotypes de base, le métagraphe d'héritage de la figure 2.3 est bancal à plus d'un titre (voir aussi *Appendix E. Classifiers Taxonomy* [17]). Exemple d'incohérence : une classe, disons *C*, se voit doter du stéréotype *«interface»* ce qui est possible puisque *Class* hérite de *Classifier* et que, comme nous l'avons déjà dit, *«interface»* peut être assigné à une instance de *Classifier*. Par hypothèse, *C* est une instance de *Class*. *C* est aussi une instance du métaélément *Interface* car le stéréotype *«interface»* n'est destiné qu'à qualifier des interfaces. C'est contradictoire au regard de la figure 2.3 car *Class* et *Interface* sont des éléments bien distincts : *C* ne peut donc pas être l'instance de ces deux métatypes en même temps.

Autre cas, les stéréotypes *«type»* et *«specification»* se marchent sur les pieds car ils désignent tous deux des éléments de modélisation faisant abstraction de toute implémentation « physique » (figure 2.4). Les stéréotypes *«implementationClass»* et *«realization»* s'opposent aussi de la même manière (figure 2.4) même si la documentation UML 2.x tente de faire croire le contraire en précisant que pour *«realization»* : « *This differs from «implementation class» because an «implementation class» is a realization of a Class which can have features such as attributes and methods which is useful to system designers.* » [17, p. 596] Cette explication alambiquée tient au fait que le stéréotype *«realization»* s'applique plutôt aux composants logiciels qu'aux classes. Pourquoi alors l'avoir assujetti à *Classifier* dont hérite *Class* ? Pause aspirine.

En outre, le métaélément *Classifier* hérite du métaélément *Type* [17, p. 61]. Ainsi, une instance de *Class* qui porte le stéréotype *«type»* est donc une classe qui est un *Classifier* qui est un type : c'est rassurant. En revanche, une instance de *Class* qui *ne porte pas* le stéréotype *«type» n'est pas* un type mais une classe qui est un *Classifier* qui est un type : c'est déroutant mais surtout contradictoire. Pause valium.

En clair, l'organisation dans le métamodèle des notions de base de la modélisation objet (type, classe…) reste critiquable. Dans toute la suite de cet ouvrage, nous utilisons ces termes de « type », « classe » ou autres souvent de façon interchangeable et avec peu de rigueur afin de nous conformer à/nous fondre dans UML. Dans un cadre plus strict et plus formel, on parle plus volontiers et le plus souvent possible de « type d'objet » au détriment du mot « classe », et ce en logique avec la spécification en informatique qui veut que l'on s'affranchisse des considérations d'implémentation fortement induites par le terme « classe ».

On considère ainsi en figure 2.4 le type *java.lang.Double* de Java qui offre un moyen élégant de manipulation des nombres flottants et dont l'implémentation repose sur le type primitif et natif *double*. On rappelle ici que *Double* (avec une majuscule) est manipulable comme toute classe Java alors que *double* est un type à accès fermé : le type *double* n'est pas instance de la classe Java *Class* alors que l'est *Double*. La figure 2.4 propose diverses alternatives de modélisation. Quelle est la

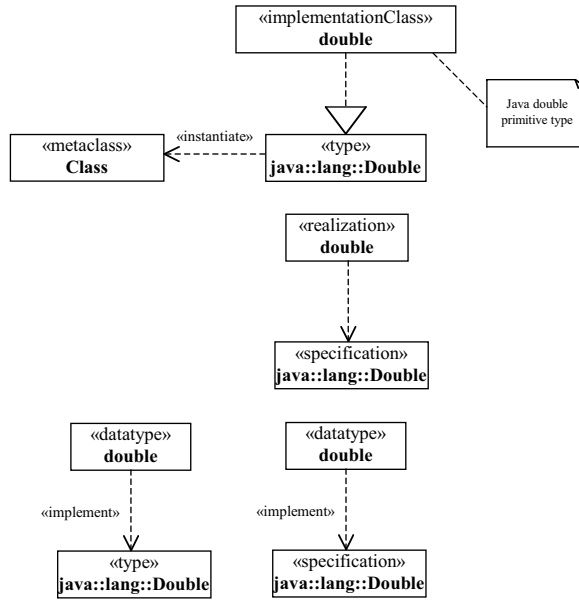


Figure 2.4 — Exemple d'utilisation des stéréotypes de catégorisation des instances de *Class* et *Classifier*.

bonne ? Le schéma du haut est conforme au style UML 1.x où *double* « réalise » *Double*, la relation de réalisation étant caractérisée par un trait pointillé et une extrémité sous forme de triangle blanc. Le schéma du milieu est son pendant. Seule une relation de dépendance (trait pointillé, extrémité non fermée) est alors mise en œuvre d'après les recommandations de la documentation. Faut-il user ou non, en plus, du stéréotype *«implement»* comme cela est fait dans les deux dernières versions du bas de la figure 2.4 ? Dans celles-ci, on utilise le stéréotype *«datatype»* spécialement fait pour des types primaires comme *double* et on lie alors *double* à *Double* stéréotypé comme *«type»* (gauche) ou comme *«specification»* (droite). On remarque en plus que chaque boîte ne comporte qu'un stéréotype. Aurions-nous pu en placer plusieurs au sein d'une même boîte ? On en frémit…

De manière évidente, il faut s'autodiscipliner. Par exemple, l'usage de la relation de réalisation (pointe de flèche triangle blanc et trait pointillé ou encore le stéréotype *«realize»* sur une relation de dépendance[1]) peut se limiter à ne lier que des éléments de modèle à stéréotypes bien arrêtés. Par exemple, en programmation Java là encore, un cas bien connu est celui de la classe *Thread* qui implémente l'interface *Runnable* (figure 2.5). Il faut pour ce cas (limité dans sa portée malheureusement car orienté programmation) se cantonner à la relation de réalisation alors qu'intuitivement le stéréotype *«implement»* de la figure 2.4 semblait plus naturel. En fait, ce der-

1. Ce dernier est utilisé tout au long de la documentation d'UML 2.x tout en étant déclaré obsolète à la fin [17].

nier, depuis UML 2.x, est dédié aux composants logiciels alors qu'il s'utilisait plus librement en UML 1.x comme nous l'avons fait dans la figure 2.4.

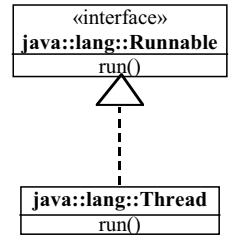


Figure 2.5 — Du bon usage de la relation de réalisation.

L'essentiel, ce qu'il faut retenir

UML n'est pas assez directif. C'est son principal défaut : les archétypes de modélisation sont trop nombreux pour un cas aussi bien ciblé et commun que celui du couple *double*/*Double* de Java par exemple. Ce qui peut apparaître comme une souplesse initiale (*i.e.* le vaste spectre de possibilités d'UML) devient pénalisant dans la pratique car les ingénieurs et les équipes logiciels délaissent vite des outils « usines à gaz ». La règle que nous allons tenter d'adopter par la suite consiste à nous limiter à ce qui fonctionne, ce qui passe parfois par le fait d'attribuer sa propre sémantique à des constructions de modélisation UML vagues (approche suivie dans les études de cas). Cette façon de faire incontournable doit s'accompagner de guides méthodologiques à destination des personnes et équipes partageant les modèles : même compréhension, et surtout même rationalité et même précision dans la fabrication et interprétation des modèles.

2.3.2 Attribut

La description d'attributs s'opère dans la partie intermédiaire des boîtes (figure 2.6), séparée du nom du type d'objet par un trait plein. Pratiquement, toutes les décorations de notation des attributs sont optionnelles (nous en développons d'autres par la suite). Dans la figure 2.6, les deux attributs sont typés et le second a une valeur par défaut qui est la valeur *#red* d'un type *Symbol* prédéfini en OCL (voir annexe en fin de chapitre). Une remarque qui a son importance est que les attributs sont visuellement distinguables des opérations par l'absence de parenthèses à la fin de leur nom.

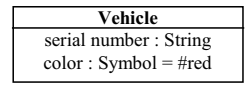


Figure 2.6 — Description d'attributs.

UML 2.x Le métatype *Enumeration* a été introduit en UML 2.x comme sous-type de *DataType* et s'utilise comme préconisé en figure 2.7.

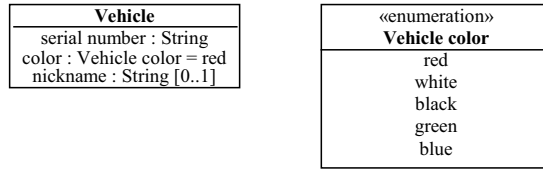


Figure 2.7 — Type énuméré.

Une caractéristique importante des attributs est qu'ils peuvent être multivalués et à ce titre parfois être indéfinis d'une instance à l'autre comme le surnom d'un véhicule (*nickname*, figure 2.7) qui existe ou non : cardinalité *0..1*. Les autres types de cardinalité (*e.g.* *, *1..**) étudiés plus loin sont bien entendu aussi utilisables dans les boîtes. Le corollaire de cette approche est que les entités UML ne respectent en rien les formes relationnelles, la première forme normale ici en l'occurrence. Le lecteur intéressé et averti peut à travers cette remarque faire une comparaison avec la méthode Merise [23] qui force le respect de la première forme normale dans les modèles conceptuels de données (MCD).

UML 2.x De manière tout à fait énervante, UML 2.x a changé la position de la cardinalité, *après* le type en UML 2.x (figure 2.7) et *avant* le type en UML 1.x (figure 2.8).

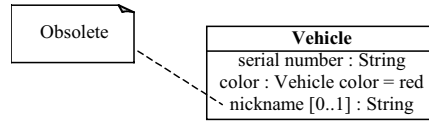


Figure 2.8 — Problème existentiel des créateurs d'UML : faut-il placer la cardinalité avant ou après le type ?

2.3.3 Stéréotypes utilisateur

Le but ultime des stéréotypes est de pallier les manques d'UML ou plus généralement d'étendre la notation à des points spécifiques. L'utilisateur peut ainsi créer des stéréotypes et les personnaliser, c'est-à-dire leur donner un sens clair et précis à l'aide d'OCL. Un stéréotype pratique que nous introduisons est celui d'identifiant et est intitulé *«unique»* (figure 2.9).

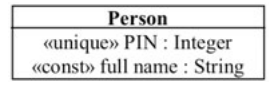


Figure 2.9 — Exemples de stéréotypes utilisateur.

Sa signification formelle est en OCL :

```
context Person inv «unique»:
   Person.allInstances→isUnique(PIN)
```

En d'autres termes, la valeur de la propriété *PIN* (*Personal Identification Number*) est discriminante pour distinguer deux instances de *Person*. L'expression OCL s'appuie sur l'opérateur *isUnique* des collections OCL et la caractéristique *allInstances* propre à tout type OCL. Nous utilisons aussi parfois le stéréotype *«const»* pour signifier que l'attribut est constant, équivalent au mot-clef *const* en C++ ou *final* en Java.

UML 2.x Notons que la relation d'extension n'existe pas en UML 1.x.

La métaclasse *Stereotype* autorise la création de stéréotypes utilisateur. Le processus de création de stéréotypes est formalisé en UML 2.x par la relation d'extension dont l'extrémité est un triangle noir (figure 2.10).

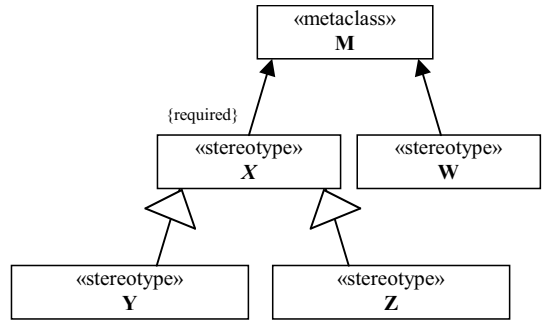


Figure 2.10 — Nouveaux stéréotypes, relation d'extension.

Soit donc M appartenant au métamodèle, *X* et *W* sont des stéréotypes devant (clause *{required}*), respectivement pouvant, être associés à des instances de M dans les modèles utilisateur. D'autres stéréotypes dérivent alors éventuellement de *X* et *W*. Comme *X* n'a pas d'existence directe (car en italique en figure 2.10), ce métamodèle dit que toute instance de M se voit toujours doter du stéréotype *Y* vu qu'il est l'unique sous-type de *X*. C'est un peu stupide, avouons-le, mais utile pour sensibiliser le lecteur au risque de rapidement écrire des bêtises si l'on n'y prend pas garde.

2.3.4 Association

L'association binaire (figure 2.11) est la relation la plus courante pour relier les classes. Les terminaisons d'association peuvent être nommées (rôle *affiliation* en figure 2.11) et sont le plus souvent décorées de cardinalités. Notons juste que l'usage de rôles est fort utile dès lors que deux associations relient deux mêmes classes (voir figure 2.15 par exemple).

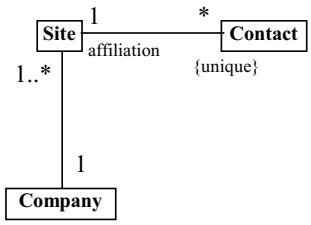


Figure 2.11 — Exemple d'associations binaires.

La contrainte prédéfinie *{unique}* apparue en UML 2.x et à ne pas confondre avec notre stéréotype *«unique»*, concerne (et donc s'utilise avec) les cardinalités dont la borne supérieure est elle-même strictement supérieure à *1*. Elle désigne un ensemble au sens mathématique, et donc le fait qu'il n'y a pas de doublon dans les instances qu'énumère la cardinalité. Dans la figure 2.11, les personnes instances de *Contact* reliées à une instance donnée de *Site*, sont distinctes en regard d'un opérateur de test d'égalité défini sur l'ensemble *Contact*. Par défaut en UML 2.x, la contrainte *{unique}* « est vraie » pour toute terminaison d'association ce qui signifie que sa présence en figure 2.11 est *a priori* inutile. En UML 1.x, par défaut aussi, le résultat d'une navigation était un ensemble. Mais certaines navigations UML 1.x rendaient une séquence ordonnée (si la contrainte prédéfinie *{ordered}* était mise en œuvre) sans qu'il n'y ait la possibilité de préciser qu'une séquence était, éventuellement, un ensemble (*i.e.* absence de doublon). Associée à la contrainte prédéfinie *{ordered}*, la contrainte *{unique}* a donc un intérêt à apparaître dans les modèles. On trouve un tel exemple dans la documentation [17, p. 43].

UML 2.x Malheureusement la plus grande confusion apparaît dans UML 2.x. Premièrement, la documentation se contredit sur le fait que par défaut une terminaison d'association retourne ou non un ensemble : « *If the end is marked as unique, this collection is a set; otherwise it allows duplicate elements.* » [17, p. 82]. Dans la page qui suit, il est écrit : « *Note that by default an association end represents a set.* » [17, p. 83].

Cas 1, une terminaison d'association ne rend pas un ensemble ce qui justifierait l'intérêt de la contrainte *{unique}*. Mais alors pourquoi la contrainte *{unique}* « est vraie » pour toute terminaison d'association dans le métamodèle ?

Cas 2, une terminaison d'association rend un ensemble. *{unique}* ne devient intéressante que si utilisée avec *{ordered}*. Oui mais la sémantique de *{ordered}* a changé en UML 2.x et désigne maintenant un ensemble ordonné. C'est maintenant *{sequence}* ou *{seq}* qui remplace *{ordered}* d'UML 1.x et *{bag}* qui représente une collection sans propriété (pas d'ordre et de doublons éventuels).

En résumé, il est affligeant d'en arriver à ce niveau d'imbroglio. Il n'y a qu'à espérer que la spécification/documentation des versions 2.x avec x > 0 s'améliorera. Pour s'en sortir dans la suite de cet ouvrage, on se place dans l'hypothèse du cas 2.

Cardinalité aussi appelée multiplicité

La liste non exhaustive des cardinalités UML utilisables dans les associations, agrégations et compositions est :

- *0..1* ;
- *1* (*1..1*) ;
- *0..** (***) ;
- *1..** ;
- *1..6* ;
- *2,4,6,8,10..**.

Les deux derniers cas permettent de donner des intervalles ou d'énumérer des valeurs. En UML, la cardinalité à une extrémité d'une relation peut ne pas être spécifiée. Le fait de ne rien noter peut être cohérent avec le fait d'orienter les relations (*cf.* section suivante, « Navigabilité »). En effet, on peut convenir que l'absence de cardinalité (à la source de la flèche matérialisant l'association) indique qu'à partir d'un type d'objet, il n'est pas possible (ou mieux, il n'est pas souhaitable) d'obtenir des informations sur le type lié.

Le danger, ce dont il faut être conscient

Notons pour les habitués du modèle entité/relation que les cardinalités sont en UML « à l'envers » (par référence à Merise) pour les binaires et « à l'endroit » pour les n-aires (n > 2).

Navigabilité

La navigation sur les associations est utile pour exprimer des contraintes ainsi que, en implémentation, pour spécifier des couplages. Les figures 2.12, 2.13, 2.14, et 2.15 montrent qu'il est possible de procéder explicitement en mettant en exergue (flèche apparente au bout de l'association) la navigabilité ou en l'empêchant purement et simplement (croix apparente au bout de l'association).

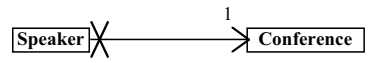


Figure 2.12 — Navigation explicite, un sens précisé, l'autre interdit.

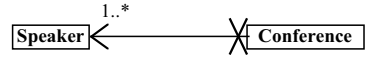


Figure 2.13 — Navigation explicite, un autre sens autorisé, l'opposé interdit.

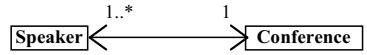


Figure 2.14 — Navigation explicite, deux sens possibles.

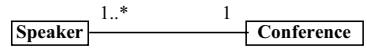


Figure 2.15 — Navigation implicite, deux sens possibles.

UML 2.x La documentation UML 2.x précise que le modèle de la figure 2.14 est à prohiber, et que, de manière générale, soit on travaille en mode explicite pour toutes les terminaisons d'association, soit on supprime toutes les flèches et toutes les croix.

> **L'essentiel, ce qu'il faut retenir**
>
> Nous vous conseillons d'utiliser les croix au besoin et de réserver les flèches à des modèles proches des considérations d'implémentation où s'ajouteront en particulier aux flèches les notations de visibilité : *public* (+), *private* (-) et *protected* (#).

Association classe

La figure 2.16 est un modèle qui introduit d'autres constructions, l'association classe en l'occurrence : trait pointillé entre l'association ici nommée *Work contract* pour contrat de travail et la classe du même nom. Un nom identique est choisi pour des raisons de clarté et de rationalité mais cela n'est pas obligatoire. L'association entre *Company* et *Person* est donc vue comme une classe à part entière puisqu'elle se lie à d'autres classes comme la législation du travail (*Labour laws*). Une association classe peut aussi intervenir dans des liens d'héritage, des liens de dépendance, etc.

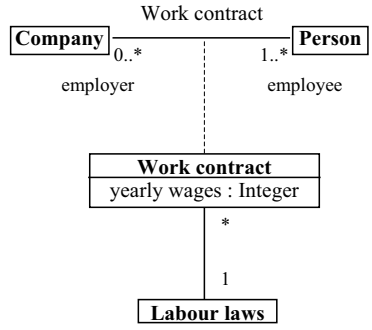


Figure 2.16 — Association vue comme une classe.

Fondamentalement, l'association entre *Company* et *Person* et la classe *Work contract* sont la même chose, visuellement séparées mais sémantiquement équivalentes.

Il y a autant de couples *Company*/*Person* formés qu'il existe d'instances de la classe *Work contract*. Sinon, observons que l'association classe permet des navigations de *Company* à *Work contract* ainsi que de *Person* à *Work contract* et ce dans les deux sens. Exprimées en OCL, les navigations *company.work contract.labour laws*, *person.work contract.labour laws*, *work contract.company* ou encore *work contract.person* sont donc toutes correctes.

Il n'y a pas de changement entre UML 1.x et UML 2.x. Rappelons cependant l'intérêt de ce concept assimilable aux propriétés des relations en Merise. Une personne au chômage n'a pas de revenus annuels (*yearly wages*). Un VRP, par exemple, salarié de quatre entreprises différentes a au contraire, quatre contrats de travail indépendants entre eux, et donc quatre montants pour sa propriété « revenus annuels ». Même si UML autorise les propriétés multivaluées, il est souvent plus élégant de produire le schéma de la figure 2.16 que de placer la cardinalité * derrière l'attribut *yearly wages* que l'on aurait incorporé dans la boîte *Person*. En d'autres termes, faire de la première forme normale, « à la Merise », n'est pas le fond de commerce d'UML mais rien ne vous en empêche.

Manque d'expressivité des associations classe

L'expression d'une cardinalité vers l'association classe n'est pas prévue en UML 1.x et 2.x alors qu'elle existe depuis fort longtemps dans d'autres méthodes objet [21]. Cela pose le problème suivant : dans l'exemple de gauche de la figure 2.17, ,un homme et une femme peuvent entretenir une relation maritale (*Husband's relation*) sans contrat de mariage au sens légal. La cardinalité vers *Marriage contract* est donc *0..1* et non pas *1..1*. En figure 2.17, à gauche toujours, l'utilisation d'un commentaire est un subterfuge pour pallier le problème, car l'on n'a pas la possibilité d'exprimer *0..1*.

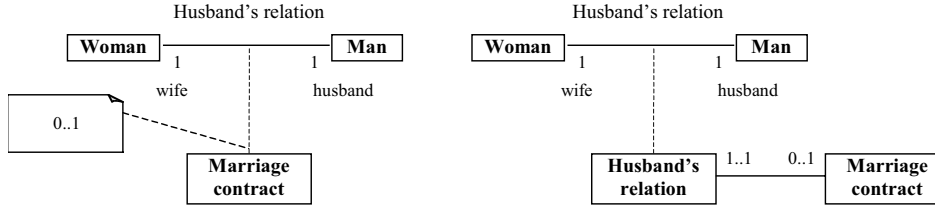


Figure 2.17 — Du bon usage des associations classe.

Même si dans le modèle de droite de la figure 1.17, on pallie le problème, de manière plus formelle cette fois-là, la solution reste inélégante et lourde : une relation maritale donne lieu ou non (cardinalité *0..1* côté *Marriage contract*) à un contrat de mariage. La lourdeur résulte du fait qu'il faut introduire la classe *Husband's relation*.

Associations n-aires

L'utilisation de relations n-aires est possible en UML. Le cas de l'emprunt (*Borrowing*) de livre dans la figure 2.18 est caractéristique : un abonné (*Subscriber*), une date et un exemplaire d'ouvrage (*Book copy*). On peut juste noter que la position des cardinalités est quelque peu perturbante en comparaison des relations binaires. Dans l'exemple, il faut lire qu'un exemplaire de livre participe dans *0* ou *1* relation de type emprunt, autrement dit, il est emprunté ou non.

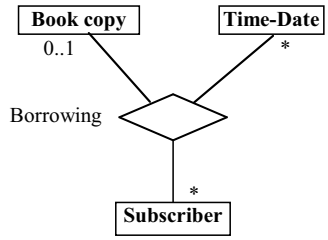


Figure 2.18 — Relations n-aires (n > 2), exemple de relation 3-aire.

UML 2.x Il semble que le nom des associations (*Borrowing* dans l'exemple) ne s'écrive plus en italique comme en UML 1.x.

Des associations n-aires aux associations binaires

La primauté des relations binaires sur les relations ternaires ou supérieures n'est plus à démontrer. Il est connu et avéré que les relations binaires sont plus simples et surtout plus faciles à implémenter en approche objet. Formellement, les relations n-aires (n > 2) sont superflues et peuvent donc être remplacées par des relations binaires appropriées. Notons à ce propos la remarque de Cook et Daniels : « *However, all ternary and other higher-order associations can be modelled more precisely using binary associations together with new types that represent the association explicitly, attached properties and/or qualifiers*. » [7, p. 42]. De manière plus générale, l'approche dite pseudo-binaire pratiquée dans le chapitre 4 de cet ouvrage et utilisée dans nombre de méthodes de spécification, est réputée pour sa qualité, et par essence, sa capacité à éviter les relations n-aires (n > 2). La figure 2.19 illustre cette approche par la transformation d'une 3-aire en trois binaires, ainsi que l'introduction de la classe *Borrowing* sur laquelle se greffent les trois binaires : attention aux cardinalités en totale inversion (dans l'espace mais aussi d'un point de vue sémantique) entre la figure 2.18 et la figure 2.19.

D'un point de vue formel cependant, le modèle de la figure 2.19 ne se supplée pas complètement à celui de la figure 2.18. Chaque occurrence de la relation 3-aire *Borrowing* en figure 2.18 est un 3-uplet, c'est-à-dire une combinaison différente de trois instances de *Subscriber*, *Book copy* et *Time-Date*. Le problème est que l'on peut former deux instances *distinctes* de la classe *Borrowing* en figure 2.19 constituées chacune *du*

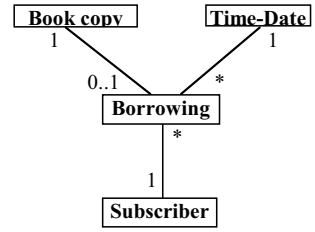


Figure 2.19 — Passage d'une 3-aire en 3 binaires.

même 3-uplet. Vu le concept d'emprunt lui-même, cela n'est pas tolérable. Que cela ne tienne, il faut l'empêcher :

```
context Borrowing inv:
   let b : Borrowing in
     self.book copy = b.book copy and self.time-Date
                    = b.time-Date and self.subscriber
                    = b.subscriber implies self = b
```

Cette contrainte OCL s'ajoute obligatoirement à la figure 2.19 si l'on veut être rigoureux dans le remplacement de la 3-aire par les trois binaires.

Association binaire contre attribut

Le danger, ce dont il faut être conscient

UML prête à confusion quant à la possibilité de décrire des associations sous forme d'attributs suivis d'un indicatif de cardinalité entre *[]*. En figure 2.20, à droite, il y a une évidente perte d'information. En effet, la cardinalité de l'association, partant de *Borrowing* et allant à *Book copy* (partie gauche de la figure 2.20), est *1*. Cette cardinalité disparaît dans la forme compressée (partie droite de la figure 2.20) : on ne sait donc plus que dans un emprunt, il y a un et un seul exemplaire de livre. Soit c'est un problème, soit c'est une information que l'on veut délibérément ignorer (cacher ?).

Au-delà, et de façon plus grave, est le conflit avec la relation de composition (losange noir) présentée dans la section 2.3.6. En effet, la notation compactée peut aussi se substituer à cette relation de composition qui a une sémantique plus poussée que l'association. Une fois les modèles compactés, comment alors distinguer l'association de la composition ?

Dans les modèles de gauche et de droite de la figure 2.20, vu que *Time-Date* est un attribut de cardinalité *1* (notation implicite à gauche, notation explicite à droite), cela revient à une composition avec pour composite *Borrowing* et partie *Time-Date*, ce qui est incohérent avec la vision expansée apparaissant en figure 2.19 : là, on a une association et non une composition.

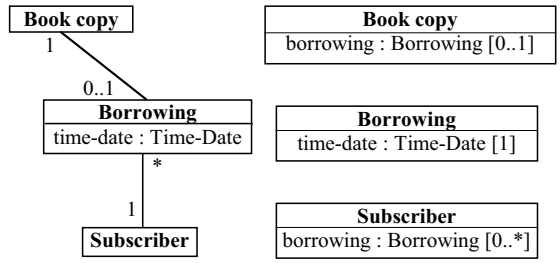


Figure 2.20 — Formes compactées des associations.

Association dérivée

Une association dérivée est calculée. Son nom est préfixé d'un / en UML 1.x et si possible d'une contrainte formelle OCL précisant à quoi elle équivaut. La documentation UML parle surtout d'attribut et d'association dérivés mais cela n'exclut pas de considérer aussi des classes (voire d'autres choses) dérivées comme cela existait en OMT [20]. Au niveau conceptuel, un élément dérivé est par définition source de redondance par le fait qu'il est déductible.

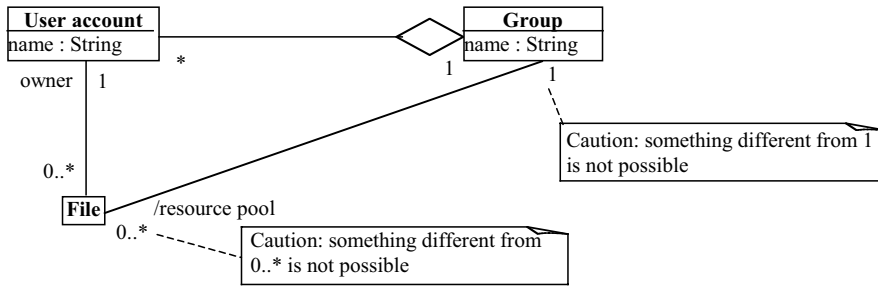


Figure 2.21 — Exemple d'association dérivée.

UML 2.x En UML 2.x, c'est le nom de rôle qui est préfixé d'un / plutôt que le nom d'association lui-même. La documentation le montre mais le métatype *Association* continue de posséder l'attribut booléen *isDerived* ce qui suppose que l'association elle-même peut toujours être notée dérivée (/ sur le trait matérialisant l'association ou / préfixant le nom de l'association si ce dernier est fourni).

Dans l'exemple de la figure 2.21, on utilise la forme UML 2.x. Ainsi, le rôle *resource pool* donne pour un groupe d'utilisateurs UNIX, l'ensemble des fichiers disponibles en réunissant tous les fichiers propriété des utilisateurs du groupe. On obtient ce pool de ressources par navigation :

```
context Group inv: -- on caractérise comment calculer l'association
   resource pool = user account.file
```

Le danger, ce dont il faut être conscient

Par souci de précision, les notes UML en figure 2.21 disent que les cardinalités ne peuvent être quelconques à chaque extrémité de l'association dérivée. Ainsi, du fait que l'introduction d'éléments dérivés est source de redondance et d'incohérence, on peut s'interroger sur sa pertinence. Une première réponse est relative à la documentation. L'usage d'éléments dérivés augmente la compréhensibilité des modèles, et plus généralement met au grand jour des informations qu'il paraît naturel et évident de lister. Une seconde réponse concerne l'implantation : la notion d'élément dérivé a un rôle clé. Soit l'élément dérivé est implanté « en dur » accentuant la nécessité de maîtriser les phénomènes de redondance et d'incohérence au niveau du code. Au-delà, le couplage s'intensifie car des attributs, associations et classes s'ajoutent sans nécessité *a priori*. Elles n'ont pas lieu d'être parce qu'elles sont calculables mais leur présence court-circuite des navigations parfois coûteuses en performance. Dans le code C++ qui suit, tout nouveau fichier créé demande la mise à jour du champ privé *_resource_pool* dans la classe *Group*. De plus, l'appel de la fonction *resource_pool()* doit vérifier l'invariant OCL exprimé précédemment. En contrepartie, on voit dans l'implantation que la fonction *resource_pool()* est rapide du fait qu'elle ne retourne qu'un simple champ.

```
class Group {
   private:
     set<File> __resource_pool; // l'association est codée en « dur »
   public:
     set<File> resource_pool() const {return _resource_pool;};
};
```

En revanche, l'inexistence du champ *_resource_pool* dans la classe *Group* ferait qu'il n'y a pas de redondance. Cela serait néanmoins pénalisant en temps puisque le code de la fonction *resource_pool()* devrait mettre en œuvre la navigation OCL. On aurait ainsi un champ *_user_account* de type collection par exemple. Il faudrait itérer sur cette collection pour obtenir tous les comptes utilisateur, puis pour chaque compte, récupérer tous les fichiers lui appartenant. Le résultat final de la fonction *resource_pool()* à retourner serait alors tous les fichiers trouvés.

Un phénomène marquant, propre à la notion d'élément dérivé, est la transformation des modèles de l'analyse à la conception, et plus généralement la traçabilité. Une classe dérivée dans un modèle d'analyse par exemple, peut disparaître d'un modèle de conception par le simple fait qu'il est décidé de ne pas l'implanter. En termes de traçabilité, la fonction entre l'ensemble des éléments de conception et ceux d'analyse, pour un logiciel donné, est donc partielle et pas totale. En d'autres termes, le raffinement des modèles d'analyse n'est pas uniquement un processus d'enrichissement, problème bien mal pris en compte dans les ateliers de génie logiciel UML, type transformateur MDE.

Qualifier

La notion de *qualifier* est « historique » car dans OMT dès l'origine et en l'occurrence encore dans UML à ce jour. Son intérêt est limité car il y a toujours moyen de

l'exprimer en OCL. La question est de savoir si on a souvent besoin de *qualifier* dans les modèles et donc d'un symbole idoine plutôt que de répéter à chaque fois l'écriture de contraintes. Dans la figure 2.22, un répertoire (*Directory*) est une sorte de fichier (*File*) : lien d'héritage. Mais il y a aussi deux associations entre *Directory* et *File*. Celle de gauche voit l'attribut *file name* de *File* comme un *qualifier*. Il est ainsi spécifié que pour un répertoire et un nom de fichier donnés, il n'y a pas ou il y a un fichier (cardinalité *0..1*) portant ce nom.

UML 2.x Il semble qu'en UML 2.x on spécifie aussi la cardinalité juste à proximité du *qualifier* représenté par le petit rectangle accolé à la boîte incarnant la classe *Directory*. Nous ne l'avons pas fait dans la figure 2.22, ce que nous pourrions appeler « le style UML 1.x ». Nous préconisons une manière de faire plus précise qui est l'ajout d'une seconde association dont les rôles sont ici en l'occurrence *container* et *contained element*. Cette association dit qu'un répertoire contient de zéro à *n* fichiers (et donc des sous-répertoires, puisque *Directory* hérite de *File*). On utilise alors ensuite la contrainte prédéfinie *{subsets <property name>}* pour donner le rapport entre les deux associations.

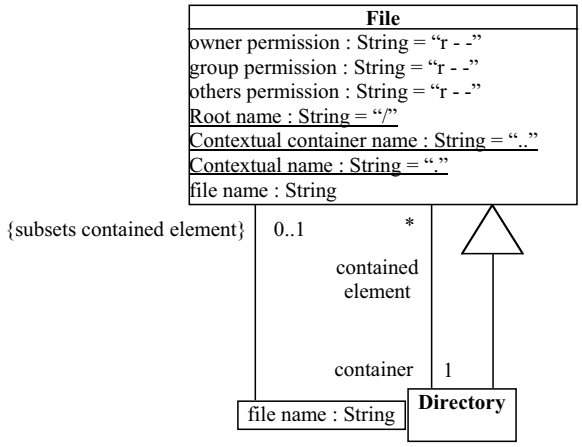


Figure 2.22 — Exemple de *qualifier* et de contrainte {*subsets <property name>*}

La contrainte *{subsets <property name>}* véritablement manquante dans UML 1.x s'inspire de Syntropy [7] qui elle-même s'inspirait de méthodes qui depuis longtemps disposent de tels outils de spécification. En Syntropy, on écrivait *{Directory::subset}* pour préciser que l'on navigue de *Directory* à *File* (*i.e.* un ensemble de fichiers est inclus dans un autre ensemble de fichiers). On dessinait aussi une flèche en pointillé portant le texte *{Directory::subset}* et qui allait de l'association de gauche à l'association de droite de la figure 2.22. Quelle que soit la notation, on souhaite dire que pour un répertoire et un nom de fichier donnés, il y a un ensemble de fichiers vide ou un singleton (cardinalité *0..1*) qui est inclus dans l'ensemble de tous les fichiers (rôle *contained element*) contenus dans ce répertoire.

Pour en terminer avec la figure 2.22, notons juste que nous avons ciblé l'exemple sur Unix via des attributs de classe (soulignement en UML équivalent à *static* en Java ou C++, voir chapitre 1) dont *Root name* par exemple, désignant à la fois le nom de la racine du système de fichiers et le label de séparation en Unix.

Petit jeu maintenant : ne peut-on pas attribuer la cardinalité *2..** au lieu de * au rôle *contained element* ? Autrement dit, pourquoi y a-t-il toujours deux fichiers « contenus » (ou mieux « répertoriés ») dans tout répertoire Unix ? Faites *cd /* puis *cd .* puis *cd ..* : la dernière commande donne-t-elle lieu à un message d'erreur ? Apparemment, non… Il y a donc bien au moins deux fichiers répertoriés sous la racine (figure 2.23) : la racine elle-même du système de fichiers qui fait référence à elle-même. La figure 2.23 est un *Object Diagram* comportant des instances des métaclasses *InstanceSpecification*[1] et *Link*. On instancie le modèle de la figure 2.22 pour faire apparaître la valeur de la navigation *contained element*, en l'occurrence pour la racine d'un système Unix, un ensemble singleton composé de cette racine elle-même. Rappelons que la figure 2.23 n'est conforme à la figure 2.22 que si la cardinalité *2..** se substitue à * vers *contained element*.

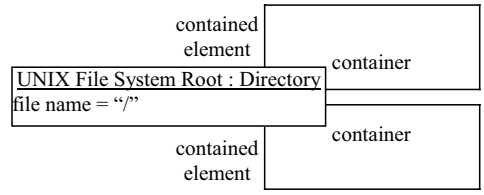


Figure 2.23 — Diagramme d'objet pour le cas d'étude Unix.

L'usage inapproprié d'un *qualifier* se situe lorsqu'un attribut identifiant sert de *qualifier*. Dans ce cas, la cardinalité est toujours *0..1* (modèle du haut de la figure 2.24), ce qui n'apporte rien : pour un numéro d'identification donné (*PIN*), il y a *ou non* un client qui a ce numéro ! Ce qui est certes intéressant mais peu informatif ! Le modèle du bas (figure 2.24) est plus approprié par le fait qu'il évite le problème du modèle en haut.

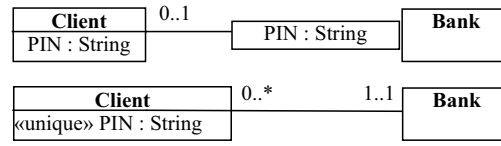


Figure 2.24 — Usage inadapté de *qualifier* (haut de la figure).

Les deux modèles de la figure 2.24 sont en fait en opposition. En bas, *PIN* est l'identifiant absolu de *Client*. En haut, si *Bank* a un identifiant, l'identifiant de *Client*

1. Attention à ce métatype nouveau en UML 2.x qui remplace *a priori* le métatype *Instance*.

est la concaténation de cet identifiant avec *PIN* (cardinalité *0..1*, côté *Client*). En résumé, si dans la réalité *PIN* est l'identifiant de *Client*, surtout ne pas proposer le modèle du haut, qui bien que « juste », fait perdre de l'information. En outre, le modèle du haut avec le stéréotype *«unique»* devant *PIN* (non représenté en figure 2.24) serait encore plus maladroit.

2.3.5 Contraintes et OCL

Nous nous limitons dans cet ouvrage à l'utilisation de la version d'OCL incluse dans UML 1.x (voir la section 2.9, annexe de ce chapitre). La version adjointe à UML 2.x (aussi appelée UML 2.0 OCL [19]) a subi de nombreuses évolutions dont la présentation dépasse le cadre de cet ouvrage. L'idée consiste surtout à montrer l'intérêt et la puissance d'expression résultante de l'utilisation d'OCL.

Donnons brièvement quelques types de base d'OCL : *Boolean*, *Integer*, *Real*, *String* (*e.g.* 'Franck Barbier'), *Collection*, *Bag*, *Set* et *Sequence*. Il faut faire attention en UML 1.x au type *String* d'OCL dont les constantes sont entre '' (cela reste en OCL 2 !) et au type *String* d'UML dont les constantes sont entre "". Cela n'a l'air de rien, mais c'est franchement pénible à l'usage ! Les deux autres types bien utiles en OCL sont *Enumeration* et *Symbol* qui vont ensemble. Exemple :

```
enum Color {blue, white, red}
Symbol, e.g. #blue, #white, #red
```

L'objet de cette section est de compléter notre réflexion sur les anciennes contraintes d'UML 1.x et les nouvelles contraintes d'UML 2.x utilisables dans les *Class Diagram*. Nous en avons déjà étudié quelques-unes que nous complétons par *{xor}*, *{redefines}* et *{union}*, ainsi que des contraintes utilisateur qui nécessitent un usage plus étendu d'OCL. Ces contraintes sont propres aux associations mais la plupart d'entre elles s'appliquent aussi aux attributs ainsi qu'aux agrégations et compositions (voir la section 2.3.6).

Contraintes usuelles

Comme nous l'avons déjà fait remarquer, la contrainte *{ordered}* prend un nouveau sens en UML 2.x parce qu'elle sous-entend un ensemble ordonné. *{xor}* est une contrainte bien connue des spécialistes du modèle entité/relation. Elle désigne l'exclusion, en l'occurrence en figure 2.25, le fait d'être lié à un mandat d'accusation (*Arrest warrant*) empêche d'être assimilé à un témoin (*Witness*), tout cela pour une même affaire (concept non modélisé). Formellement, pour une personne donnée, l'intersection entre un sous-ensemble du produit cartésien *Person* x *Arrest warrant* et un sous-ensemble du produit cartésien *Person* x *Witness*, est vide.

Si l'on a, non pas un ensemble ordonné mais une collection ordonnée (doublons possibles) alors on utilise la contrainte *{seq}* (figure 2.26). L'intérêt de définir des ordres dans les modèles, c'est qu'ils sont exploitables en OCL avec des opérateurs dédiés : *first*, *last*, *at*…

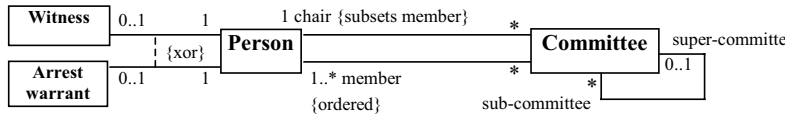


Figure 2.25 — Contraintes {*xor*} et {*ordered*}.

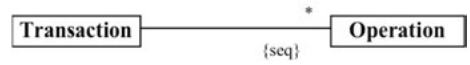


Figure 2.26 — Une transaction possède une suite ordonnée d'opérations.

Voici un exemple de contrainte OCL :

```
context Transaction inv:
   operation→first().oclIsKindOf(Operation) = true
                                        -- le premier est de type Operation
   operation→last().oclIsKindOf(Operation) = true
                                        -- le dernier est de type Operation
   let i : Integer = 1 in               -- si la collection est non vide,
                                celui à la position 1 est de type Operation
     operation→notEmpty() implies operation→at(i).oclIsKindOf(Operation) = true
```

La contrainte *{redefines}* était cruellement manquante dans UML 1.x. Elle vient de Syntropy [7] et est très utile. Elle permet de préciser plus finement les propriétés d'une terminaison d'association après un héritage. La figure 2.27 montre qu'un salarié (*Employed person*) hérite de l'association entre *Person* et *Company* du fait qu'il est un sous-type de *Person*. La cardinalité change contextuellement : sa borne basse vers *Company* devient *1* alors qu'elle était initialement à *0*. En clair, une personne, puisqu'elle est dite « employée », a au moins un lien avec une compagnie.

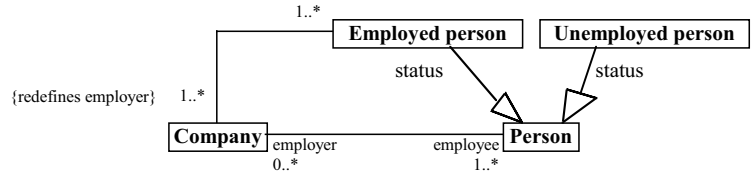


Figure 2.27 — Contrainte {*redefines*}.

Au moment de la rédaction de cet ouvrage, la syntaxe exacte de la contrainte est *{redefined}* ou *{redefines}* avec une probabilité plus forte d'acuité pour la seconde forme. Il y a en effet ambiguïté dans la documentation.

Finalement, la contrainte *{union <property name>}* également importante car beaucoup utilisée dans le métamodèle, se met en œuvre en complémentaire de *{subsets <property name>}*. Comme son nom l'indique, elle désigne la réunion d'autres

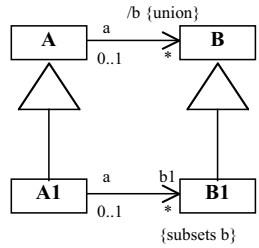


Figure 2.28 — Contrainte {*union <property name>*}.

navigations et s'apparente donc aux associations dérivées. La figure 2.28 l'illustre. Cet exemple est extrait directement de la documentation UML 2.x [17, p. 86].

Dans la figure 2.28, il faut lire que l'ensemble des *B* associés à une instance de *A* est calculé (/ devant *b*) ; il est la réunion des ensembles qui sont *ses* sous-ensembles, ce qui est certes évident. En l'occurrence le schéma n'en montre qu'un de ces sous-ensembles : *b1*.

L'idée qu'il faut comprendre, c'est que des instances de *A* existent à travers des instances de *A1*. Autrement dit, toute instance de *A1* est par définition (héritage en figure 2.28) instance de *A*. La navigation *b* somme donc les instances de *B1* attachées à une instance donnée de *A1*. Pour cette même instance, il y a *aussi* dans la navigation *b* des instances de *B* qui ne sont pas instances de *B1*. C'est possible, puisque *B* est une classe concrète (elle n'est pas notée en italique). Il semble néanmoins que *b* = *b1* car *b* est dérivée, ce qui limite beaucoup par conséquent l'intérêt du modèle de la figure 2.28.

Première synthèse et mise en œuvre avancée d'OCL

À la différence de beaucoup de modèles précédents, celui de la figure 2.29 est un cas réel. Une entreprise, appelons-la 6KO-EMEA, veut implanter un *call/contact center* pour améliorer sa relation client (*Customer Relationship Management*, CRM). Cette société vend du matériel informatique réseau : routeurs…

Un contact est un individu d'un site (affiliation) d'une entreprise donnée. Sites et entreprises sont vus comme des organisations (*Organization* : classe abstraite). Une organisation a un nombre bien défini et arrêté de rôles choisis parmi vingt-quatre possibilités au maximum : ingénieur réseaux, responsable des achats, administrateur d'infrastructures informatiques…

Un même contact peut avoir plusieurs « casquettes » dans son site et sa compagnie. Il faut donc connaître ces « casquettes » en fonction de l'organisation. Exemple : O.B.-D. est ingénieur réseaux dans son site, mais il est aussi responsable des achats réseaux pour toute sa compagnie. Il téléphone donc souvent à 6KO-EMEA mais ses demandes sont très différentes selon la casquette avec laquelle il intervient. La qualité du suivi de ses demandes est donc grandement liée à la bonne connaissance de ses responsabilités. Cette exigence est modélisée dans la figure 2.29 par l'association de *Contact* vers *Organizational role*. Elle est orientée pour signifier

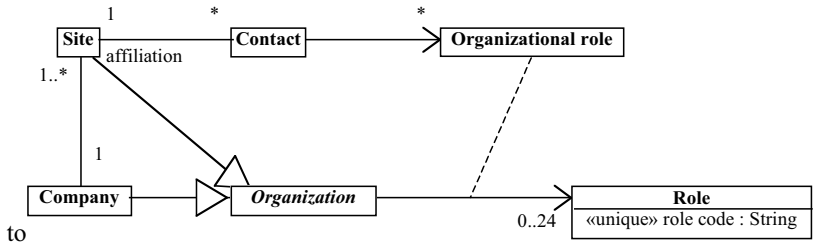


Figure 2.29 — Cas 6KO-EMEA avec contraintes OCL.

qu'à partir d'une instance d'*Organizational role*, on ne veut pas pouvoir déterminer ses contacts.

Le modèle de la figure 2.29 se complète avec des contraintes OCL :

```
context Site inv: -- les rôles d'un site en tant qu'organisation sont inclus
                  dans l'ensemble des rôles de la compagnie de ce site
   company.role→includesAll(role)
context Contact inv: -- si l'on réunit tous les rôles du site
                        et de la compagnie d'un contact donné,
                        on sait que les rôles que joue ce contact
                        appartiennent nécessairement à la réunion
(affiliation.company.role→union(affiliation.role))→
      ➥ includesAll(organizational role.role)
```

Mise en œuvre avancée d'OCL, exemple du concept de metadata *d'OMT*

La relation *metadata* est une relation d'instanciation entre types. Rien de choquant car les types d'un modèle utilisateur sont bien des instances des types du métamodèle d'UML. Cependant, on considère alors qu'il y a une différence de niveau méta entre le modèle utilisateur et le métamodèle.

L'idée derrière la relation *metadata* est de créer des relations d'instanciation au sein même d'un modèle utilisateur. On le simule avec l'association (figure 2.30).

Concrètement, quel est le lien sémantique entre un exemplaire de livre (*Book copy*, notion physique s'il en est, avec un numéro d'inventaire) et un livre (*Book*), au sens conceptuel et logique (un écrit, un titre, un auteur…), parfaitement caractérisé par un numéro ISBN. Autrement dit, quel est le lien sémantique entre l'exemplaire physique que vous avez entre les mains et l'ouvrage, ensemble d'idées intitulé : *UML 2 et MDE* ? L'héritage ? Perdu, certainement pas. Donnons sa sémantique :

- la cardinalité à côté du rôle *model* (*Book*) est toujours *1..1* ;
- il y a préexistence de l'objet *model* (*Book*) par rapport à l'objet « exemplaire » ou « physique » : *Book copy* ;
- il y a finalement immutabilité de l'objet *model*. En d'autres termes, étant donné un objet exemplaire, son modèle est *ad vitam aeternam* le même.

Si nous devons implanter cette sémantique en C++, nous disposons de tous les moyens pour le faire :

```
class Book_copy {
   private:
      const Book& _model;
      // or static const Book & _model;
   …
};
```

L'usage du *&* de C++ assure la préexistence et l'unicité d'une instance de *Book* sur celle de *Book copy* (le constructeur de *Book copy* va obligatoirement nécessiter un argument de type *Book*). De plus, le lien est figé grâce au *const* de C++.

La relation de *metadata* n'est pas native en UML 1.x car elle n'est pas exprimée au niveau méta par un sous-type de *Relationship* d'où la simulation par association en figure 2.30 : en bas de la figure, un item avec un numéro de série n'est jamais que la représentation physique d'un item en catalogue.

UML 2.x En UML 2.x, notre sentiment est que la relation *metadata* est bien modélisée avec l'usage du stéréotype *«instantiate»* (voir figure 2.30).

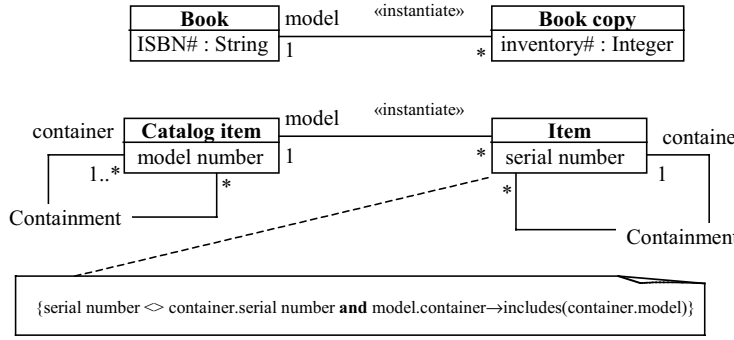


Figure 2.30 – Approche en UML de la relation metadata d'OMT.

2.3.6 Agrégation et composition

L'agrégation et la composition posent historiquement des problèmes en UML [10]. Basées toutes deux sur une sémantique bancale, elles sont néanmoins intensivement utilisées (la composition ou losange noir surtout) dans le métamodèle lui-même, rendant *de facto* ce dernier inconsistant. Des corrections formelles sont proposées dans l'article de Barbier *et al.* [3], mais UML 2.x continue d'offrir des relations dont l'interprétation varie d'un utilisateur à l'autre, ce qui est un non-sens en modélisation. Imaginons une spécification faite par un informaticien Dupont dont un autre informaticien Durand doit réaliser l'implémentation. Que va-t-il se passer si ces deux personnes n'ont pas la même compréhension d'une et/ou de ces deux relations ?

UML 2.x On peut dire que la situation empire en UML 2.x car on élude le problème : « *Precise semantics of shared aggregation varies by application area and modeler* » [17, p. 80]. En

d'autres termes, l'agrégation ou losange blanc a la sémantique que l'on veut bien lui donner en fonction des circonstances !

La figure 2.31 donne le métamodèle pour UML 1.x et met en particulier en exergue le métatype *AssociationEnd*, source de tous les maux (voir par exemple l'article de Henderson-Sellers *et al.* [10] pour un diagnostic détaillé).

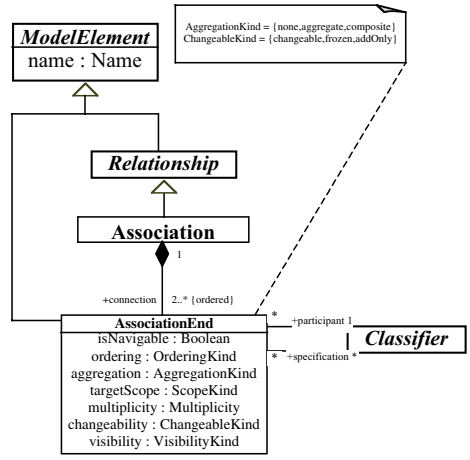


Figure 2.31 — Métamodèle UML 1.x pour l'agrégation et la composition.

UML 2.x Le métamodèle d'UML 2.x s'appuie sur le métatype *Property* qui d'une certaine manière se substitue à *AssociationEnd* (ce dernier disparaît d'UML 2.x). Malheureusement *Property* conserve le métaattribut *aggregation* de type *AggregationKind* vu comme un type énuméré avec les valeurs possibles *none*, *shared* et *composite* (*shared* remplaçant *aggregate* d'UML 1.x en figure 2.31). Il s'agit de la principale modification mais après vérification, l'agrégation et la composition restent boguées en UML 2.x. Passons à leur utilisation dans les modèles et renvoyons le lecteur au texte de Barbier *et al.* [3] pour une refonte totale du métamodèle UML qui donnerait une bonne assise à ces deux relations.

Agrégation (diamant ou losange blanc)

La spécificité de l'agrégation est la possibilité, sans que ce soit systématique, que le composant soit partagé par plusieurs composés. Dans la figure 2.32, un distributeur automatique bancaire (*ATM*) possède plusieurs périphériques (*Device*), mais la cardinalité vers *ATM* étant *1*, il ne peut y avoir partage. Certes, l'utilisation du losange blanc paraît assez étrange, mais reste tout à fait acceptable. La question qui vient alors naturellement est : n'y aurait-il pas plutôt fallu utiliser la composition ? Non, car en UML elle suppose des dépendances de vie et/ou de possession forte entre composés et composants (voir la section « Composition (diamant ou losange noir ») d'où un usage que nous conseillons modéré.

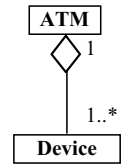


Figure 2.32 — Exemple d'agrégation.

L'agrégation peut poser des problèmes

N'ayant plus de sémantique en UML 2.x, on pourrait croire que l'agrégation n'engendre plus de problèmes. Malheureusement, il a été conservé la notation compactée illustrée par cet extrait de la documentation : « *If there are two or more aggregations to the same aggregate, they may be drawn as a tree by merging the aggregation ends into a single segment. Any adornments on that single segment apply to all of the aggregation ends.* » [17, p. 84]. Le résultat est en figure 2.33 où le modèle de droite est la forme dite compactée : un stylo (*Pen*) est composé d'un bouchon (*Cap*) et d'un corps (*Pen body*). Dans le modèle de droite, quelle cardinalité écrire ? *1* ? Oui, mais c'est là une coïncidence que *Cap* et *Pen body* se relient à *Pen* avec la même cardinalité côté *Pen*. Le modèle de gauche en revanche permet d'exprimer les cardinalités relation binaire par relation binaire. D'ailleurs le modèle de droite laisserait croire à une relation ternaire.

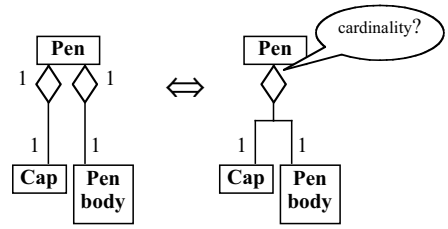


Figure 2.33 — Usage délicat, voire inapproprié, de l'agrégation.

L'essentiel, ce qu'il faut retenir

Dans l'extrait de la documentation UML 2.x susmentionné, il est dit que dans un modèle comme celui de la droite de la figure 2.33, toutes les propriétés de la terminaison doivent être les mêmes : même cardinalité a priori, mais aussi même nom de rôle ? Mêmes contraintes ? Etc. Le cas est suffisamment rare pour ne pas s'y aventurer à notre goût. En résumé, ce modèle de droite prête à confusion : il faut l'éviter.

Cas d'école pour l'agrégation

L'expérience montre que les informaticiens « aiment » ce concept d'agrégation/composition qui d'ailleurs prend de plus en plus d'importance en développement

basé composant logiciel. La figure 2.34 est un cas standard. Pour dire qu'une pièce (*Room*) est constituée de plusieurs murs (*Wall*), l'agrégation est appropriée, car elle permet de dire qu'en plus un mur peut être partagé par plusieurs pièces (cardinalité *1..**). La composition est hors sujet, puisque supprimer une pièce d'un bâtiment n'amène pas à détruire tous ses murs, ceux-ci bornant d'autres pièces : pas de dépendance de vie donc !

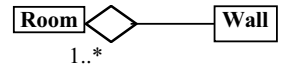


Figure 2.34 — Exemple d'agrégation avec partage des composants entre composés.

Composition (diamant ou losange noir)

La valeur *composite* que peut prendre l'attribut *aggregation* de type *AggregationKind* est caractérisée comme suit : « *Indicates that the property is aggregated compositely, i.e., the composite object has responsibility for the existence and storage of the composed objects (parts).* » [17, p. 80]. Dans la figure 2.35, une commande (*Order*) est composée de lignes de commande (*Order line*). Par définition, la cardinalité côté losange noir est toujours *0..1* ou *1* (pas de partage, quoi qu'il en soit).

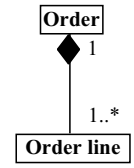


Figure 2.35 — Exemple de composition.

L'exemple de la figure 2.35 est bien dans l'esprit des dépendances de vie : supprimer la commande, c'est supprimer toutes ses lignes. On parle aussi de dépendance « existentielle » ou « ontologique » : une ligne de commande est une notion qui n'a pas de sens hors de la commande à laquelle elle appartient.

Composition, modèle(s) à problème

On pourrait écrire une encyclopédie sur la relation tout-partie. Il existe d'ailleurs aujourd'hui de nombreux ouvrages à son sujet dans des domaines aussi divers que les mathématiques, la philosophie ou encore la linguistique — citons l'ouvrage de Leniewski [11] pour la référence théorique la plus connue sur le sujet. C'est par rapport à cette littérature surabondante que l'on prend conscience de la déficience de cette relation universelle en UML, et surtout, de la façon dont elle est introduite dans ce même UML. Un exemple parmi d'autres est celui de la figure 2.36. Un camion (*Truck*) possède un moteur (*Engine*). Une voiture (*Car*) possède aussi un moteur. On obtient le modèle de la figure 2.36 mais il faut ajouter que ce n'est pas le même moteur !

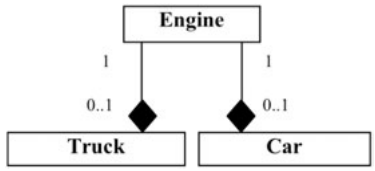


Figure 2.36 — Cas problématique pour la composition.

Une contrainte OCL s'impose alors :

```
context Engine inv:
   self.car→notEmpty() implies self.truck→isEmpty()
```

La composition est certes fondée sur l'absence de partage mais pour un type composé donné. Dans un modèle comme celui de la figure 2.36 où il y a deux types jouant le rôle de composé, l'usage de la composition ne va pas imposer de suite une exclusion de partage d'où dans ce cas particulier, une contrainte OCL supplémentaire, et pour le cas général, la plus grande vigilance sur ce qui est véritablement modélisé.

Dans la suite de cet ouvrage, nous utilisons la composition sans sémantique particulière, c'est-à-dire sans implication forte et stricte sur les dépendances de vie, la transitivité ou toute autre propriété réputée assignée à cette relation. En effet, la composition est aussi reconnue transitive alors que fort peu de relations tout-partie sont transitives. Mon bras fait partie de moi. Je fais partie de l'université de Pau. Mon bras ne fait néanmoins pas partie de l'université de Pau. Encore une tare parmi d'autres de la composition UML.

Notations de la composition

Revenons à la notation compactée des associations de la figure 2.20. Il y avait en UML 1.x un conflit car cette notation compactée était censée concerner et les associations et les compositions. Si l'on regarde le modèle de droite de la figure 2.37, il est évident que l'on ne sait pas si la relation entre bras (*Arm*) et respectivement, main (*Hand*), poignet (*Wrist*) et coude (*Elbow*), est une association ou une composition. En d'autres termes, transformer le modèle de gauche pour obtenir le modèle de droite n'est pas vraiment une bonne idée !

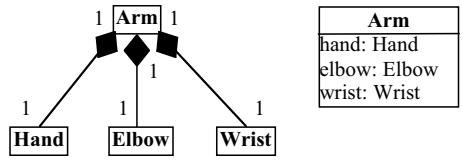


Figure 2.37 — Variations sur l'expression de la composition

UML 2.x En UML 2.x, l'innovation réside dans les *Composite Structure Diagram* qui généralisent une relation connexe à l'agrégation/composition : la relation d'inclusion topo-

logique aussi appelée *Containment*. Les *Composite Structure Diagram* restent néanmoins plus cantonnés aux composants logiciels (voir la section 2.4 de ce chapitre et la fin du chapitre 6 plus particulièrement).

Il en résulte que le modèle le plus à droite dans la figure 2.38, tout à fait commun dans UML 1.x, semble maintenant plutôt réservé en UML 2.x à décrire des relations de *Containment* entre composants logiciels. De plus, la sémantique de la composition et de l'inclusion topologique étant mal formée, il est difficile de distinguer ces deux relations. En termes de notation, il vaut donc mieux éviter le modèle de droite de la figure 2.38 pour « signifier » la composition.

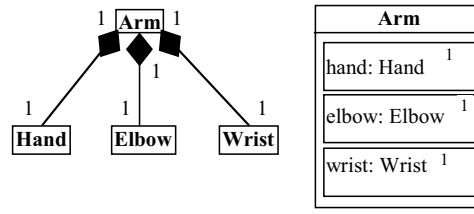


Figure 2.38 — Autres variations sur l'expression de la composition

2.3.7 Héritage ou généralisation/spécialisation

L'héritage (triangle blanc en figure 2.39 avec trait plein, à ne pas confondre avec la relation de réalisation : triangle blanc et trait pointillé) est une relation ordinaire et de base en UML. Dans la figure 2.39, l'absence de stéréotypes pour *Vehicle* et *Car* assure que ce sont des classes (et non des types ni des interfaces). Parmi les trois sens usuels de l'héritage, en l'occurrence *Subclassing*, *Subtyping* et *Is-a* [22], on est en UML plus proche de la sémantique de *Is-a* que de celles de *Subclassing* et *Subtyping*. Cela peut paraître de prime abord étonnant vu que l'on a des « classes », mais *Subclassing* n'est liée qu'à l'implémentation et correspond à l'héritage de structure en programmation objet. *Subtyping*, au contraire, incarne l'héritage de comportement basé sur la substituabilité. UML est bien insuffisamment fondé sur la théorie des types pour prétendre supporter le sous-typage. *Is-a* est l'inclusion ensembliste et considérer en

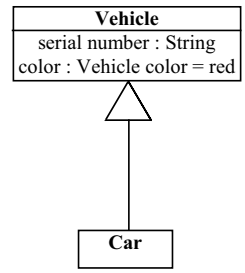


Figure 2.39 — Héritage simple.

UML que l'inclusion est la sémantique de l'héritage, est justifié par l'existence des contraintes prédéfinies utilisables avec cet héritage (voir les sections « Formes et contraintes pour l'héritage » et « Interprétation ensembliste de l'héritage, lien *Is-a* »). Ce n'est pas un hasard si le terme consacré en UML n'est pas héritage mais l'expression généralisation/spécialisation.

Classe abstraite

Une classe abstraite est en UML une classe dont le nom est en italique. Elle s'apparente à la notion de classe abstraite en programmation (*abstract class* en Java ou encore les classes offrant des fonctions virtuelles pures en C++).

Les figures 2.1, 2.3, 2.10, 2.29, et 2.31 comportent des classes abstraites au niveau métamodélisation particulièrement. Revenons sur le cas réel de la figure 2.29, *Organization* est une classe abstraite signifiant qu'elle ne peut être instanciée que via ses sous-types à la réserve près que ceux-ci ne doivent pas non plus être abstraits, ce qui est le cas de *Site* et de *Company*.

Un conseil utile est de faire apparaître quels sont les services, lorsqu'ils existent et sont décrits, qui rendent une classe abstraite : il faut les faire apparaître en italique (voir plus loin dans ce chapitre) comme cela est pratiqué dans UML 1.x. En Java par exemple, le *modifier abstract* préfixe un service, si celui-ci est abstrait.

Héritage multiple

L'héritage multiple est possible en UML et est historiquement utilisé dans le métamodèle lui-même. Par exemple, le métatype *AssociationClass* hérite de *Class* et *Association* à la fois. Le monde réel comme le prouve la figure 2.40 est lui aussi organisé sur la base de ce mécanisme. Autant donc en disposer dans UML, même si son utilisation est la plupart du temps mal maîtrisée, que ce soit en programmation (Eiffel, C++...) ou en conceptualisation. Dans ce dernier cas, la difficulté s'accroît par le fait que le support d'implémentation peut ne pas posséder ce mécanisme (le modèle objet-relationnel d'Oracle par exemple, le langage de programmation Smalltalk…) d'où la déstructuration inévitable des modèles à l'implémentation.

Les programmeurs Eiffel (voire certains programmeurs C++) qui lisent ces lignes savent néanmoins toute la puissance et donc tout l'intérêt de ce mécanisme. Dans la figure 2.40, il est donc bien utile de caractériser les ornithorynques comme à la fois des ovipares et des mammifères, puisqu'ils pondent des œufs et allaitent leurs petits.

Formes et contraintes pour l'héritage

Les contraintes pouvant s'attacher aux liens d'héritage sont :

- *{complete}* et son contraire *{incomplete}* ;
- *{disjoint}* et son contraire *{overlapping}*.

UML 2.x On rappelle ici, en toute cohérence avec l'assimilation de l'héritage à l'inclusion ensembliste, que *disjoint* signifie que l'intersection des sous-types est vide : par exemple, si on est un mammifère (*Mammal*), on ne peut pas être à la fois un chat (*Cat*) et

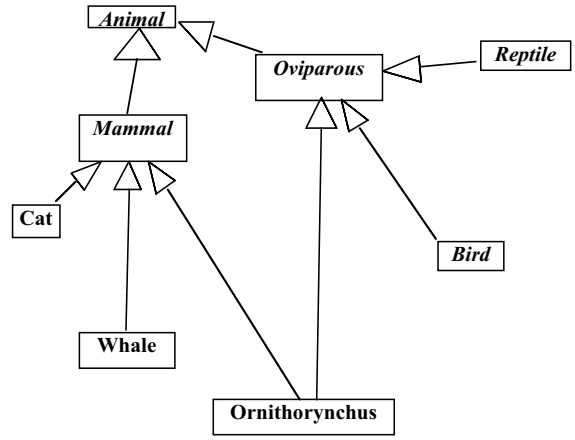


Figure 2.40 — Partie du règne animal décrit avec l'héritage multiple.

une baleine (*Whale*). Quant à *incomplete*, il signifie que la réunion des sous-types n'est pas égale au type. Si je réunis toutes les instances de *Mammal* et de *Oviparous*, je n'obtiens pas toutes les instances d'*Animal*.Les contraintes demeurent en UML 2.x, mais avec des présentations différentes. En l'occurrence, le modèle de la figure 2.41 n'est plus tout à fait valide, au « goût du jour » dirons-nous, car les quatre contraintes prédéfinies précédentes doivent s'utiliser deux à deux. Quatre couples sont bien entendu possibles, à savoir : *{incomplete, disjoint}*, *{complete, disjoint}*, *{incomplete, overlapping}* et *{complete, overlapping}*. De plus, la présentation des sous-types directs de *Oviparous*, comme celle de la figure 2.41, n'est plus acceptable dans sa forme : le symbole « … » disparaît en UML 2.x, ce qui est sain car il est redondant avec le fait de qualifier la liste des sous-types d'*incomplete*. En outre, le duo *{incomplete, disjoint}* étant le couple par défaut, il peut être omis.

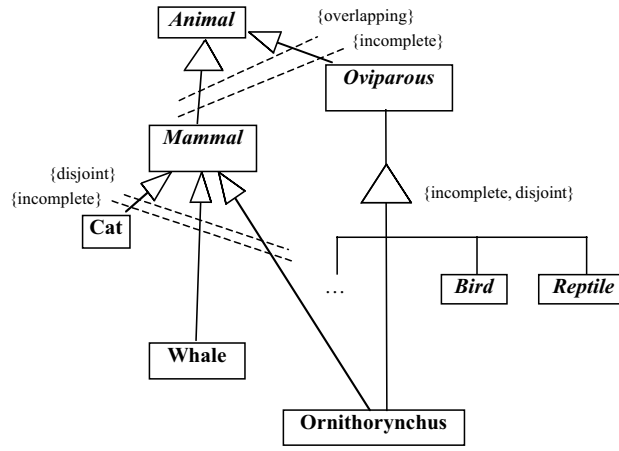


Figure 2.41 — Archétype de graphe d'héritage en UML 1.x.

Les spécialistes de la spécification auront reconnu ici les célèbres contraintes d'exclusion et de totalité entre sous-types vieilles comme le monde car présentes depuis bien longtemps dans d'autres méthodes, dont NIAM ou ORM abordées au chapitre 4.

La figure 2.42 corrige donc le tir du point de vue de l'aspect, sans faire de révolution. On supprime *{incomplete, disjoint}* puisque par défaut.

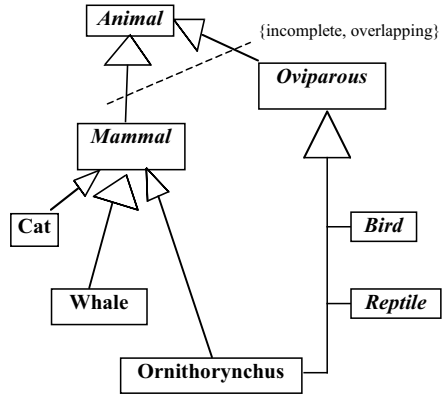


Figure 2.42 — Archétype de graphe d'héritage en UML 2.x.

Le danger, ce dont il faut être conscient

La notation des contraintes sur l'héritage n'est pas très pertinente, voire pratique, en UML. Essayons de représenter trois sous-types d'un type donné et décrivons trois disjonctions deux à deux ce qui par définition n'est pas équivalent à une disjonction à trois.

Solution : c'est graphiquement difficile et même si c'est possible, c'est esthétiquement gênant. Passons à quatre sous-types : c'est impossible à moins de faire des petits sauts comme en symbolique électronique. La figure 2.43 illustre le problème. Pour des raisons de surcharge du schéma, nous n'avons pas écrit le nom des contraintes entre parenthèses.

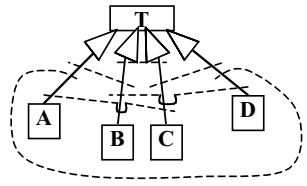


Figure 2.43 — Difficultés des contraintes d'héritage.

Cohérence en héritage multiple avec contraintes

Il advient vite des problèmes lorsque l'on mélange un peu tout. Partons du principe qu'il n'y a que deux sortes de comptes bancaires : les comptes chèques (*Checking account*) non rémunérés et les comptes épargne (*Savings account*) rémunérés[1], nous obtiendrons le modèle de la figure 2.44. La racine du graphe d'héritage *Account* est forcément abstraite, car la réunion des deux types de compte donne tous les comptes en raison de l'utilisation de *complete*. Si *Account* n'était pas abstraite, il serait possible de l'instancier sans préciser si c'est « chèque » ou « épargne ». Il y aurait donc des comptes qui n'appartiennent pas à la réunion des instances des deux sous-types : c'est incohérent au regard de la mise en œuvre de *complete*. Quant à l'usage de *disjoint*, il empêche l'héritage multiple. En l'occurrence, la classe X n'a aucun sens dans la figure 2.44 : une instance de X est à la fois « chèque » et « épargne », mais il est interdit d'être les deux à la fois. C'est l'usage de *disjoint* qui le dit.

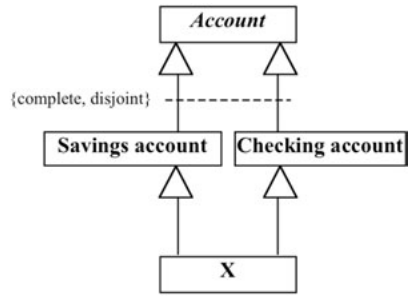


Figure 2.44 — Cas incorrect en utilisation conjointe de l'héritage multiple et de contraintes sur l'héritage.

Interprétation ensembliste de l'héritage, lien Is-a

Le développement d'un graphe d'héritage un tant soit peu compliqué peut rapidement tourner au cauchemar si l'on veut bien prendre le temps de réfléchir à ce que l'on est en train de spécifier. À droite de la figure 2.45, on utilise une représentation équivalente par diagrammes de Venn. Les parties noires sont par convention vides. Le carré *H* représente l'ensemble de toutes les instances du type d'objet *Helicopter*. Les cercles se trouvant à l'intérieur y sont par inclusion.

Dans le diagramme de Venn en haut et à droite de la figure 2.45, on montre par exemple qu'il peut y avoir des hélicoptères à la fois militaires (*Military helicopter - MH*) et armés (*Armed helicopter - AH*) : contrainte *overlapping*. Mais *Military helicopter* est une classe abstraite (en italique) et donc instanciable que par ses sous-types comme *Fighter helicopter*. Le deuxième diagramme de Venn au milieu et à droite de la figure 2.45 le montre.

1. Ce n'est pas tout à fait vrai dans la réalité. On ne fait ici l'hypothèse que pour les besoins de la démonstration.

En résumé, on décrit des types d'objet à propriétés réduites et à ce titre on est amené à ne plus rien maîtriser dans le graphe d'héritage. Le dernier diagramme de Venn en bas et à droite de la figure 2.45 démontre qu'il devient difficile de comprendre ce que l'on a formellement décrit. En d'autres termes, le diagramme de Venn est déjà difficile à interpréter, alors que le schéma UML ne reflète pas ou mal, les instances que l'on peut réellement créer à partir des types impliqués dans tout le graphe d'héritage.

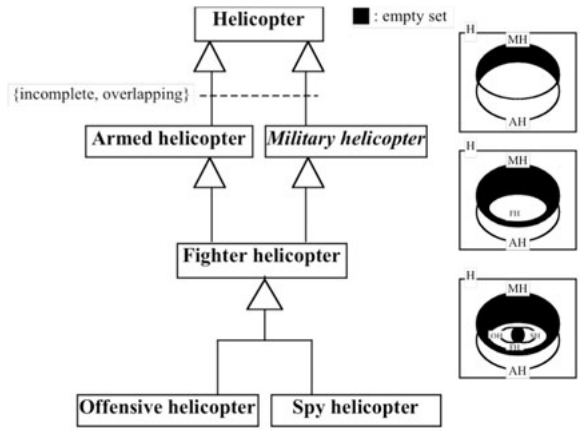


Figure 2.45 — Graphe d'héritage et interprétation ensembliste.

Discriminant en héritage

Le terme *discriminator* d'UML 1.x a disparu au profit de celui de *power type* en UML 2.x. La présentation originelle est celle de la figure 2.46. Des labels sont associés aux liens d'héritage pour donner le critère qui a fait obtenir le sous-type. Ce mécanisme n'est pas très sain en général, car il induit l'utilisation de l'héritage mul-

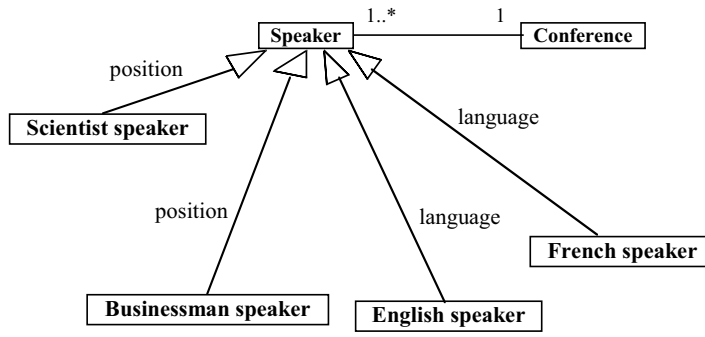


Figure 2.46 — Critère d'obtention des sous-types ou discriminant en héritage.

tiple si l'on veut ensuite décrire les conférenciers français (*French speaker*) qui sont aussi des scientifiques (*Scientist speaker*).

UML 2.x Le formalisme de la figure 2.46 reste. Des alternatives de notation existent néanmoins comme celles de la figure 2.47. De manière additionnelle et spécifique, le label étiquetant le lien d'héritage, s'il est préfixé d'un :, désigne un *power type*. L'idée est de dire qu'une espèce d'arbre (*Tree species*), toujours connue et unique (cardinalité *1*), donne de façon non exhaustive (contrainte *incomplete*) deux sous-types : les chênes (*Oak*) et les sapins (*Fir*).

Le danger, ce dont il faut être conscient

Ce type de modélisation est à notre sens fortement redondant. *Tree species* donne par essence l'information sur l'espèce via une navigation partant de *Tree*. Avoir une instance d'*Oak* par exemple, aussi instance d'arbre, fait doublon. L'espèce de cette instance est une information intrinsèque de sa classe (*Oak*). C'est aussi une information calculable par navigation. En résumé, *power type* est une notion plutôt irrationnelle et mal assise.

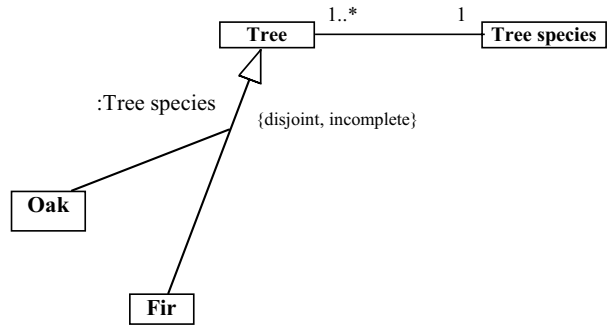


Figure 2.47 — Autre notation pour les *power type* en UML 2.x.

2.3.8 Exemple de synthèse

Nous donnons ici un exemple académique et de synthèse (figure 2.48). Le but est de montrer que les modèles UML sont en général graphiquement concis mais qu'une spécification rigoureuse engendre beaucoup de contraintes formelles et que donc, OCL est un outil incontournable.

Modèle initial

Expliquer la figure 2.48 revient à expliquer la figure 2.49. Un croisement de segment (*Segment crossing*) est parfaitement défini par au moins deux segments qui se croisent, voire plus (cardinalité *2..**). Les segments et les croisements de segment sont nommés (voir figure 2.48).

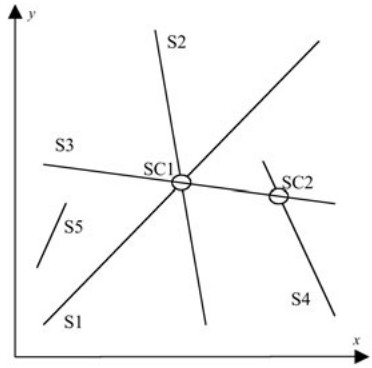


Figure 2.48 — Notion de segment et de croisement de segment dans un repère cartésien.

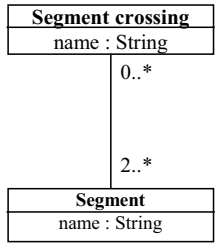


Figure 2.49 — Expression primitive du besoin.

Validation par Object Diagram

L'intérêt des *Object Diagram* est de représenter plus intuitivement des situations. La figure 2.50 est ainsi intéressante à double titre, comme instanciation de la figure 2.49 et comme une vision objet d'une réalité, celle de la figure 2.48. On voit par exemple que *S5* ne concourt à la définition d'aucun croisement ou encore que *SC1* est caractérisé par le croisement de *S1*, *S2* et *S3*.

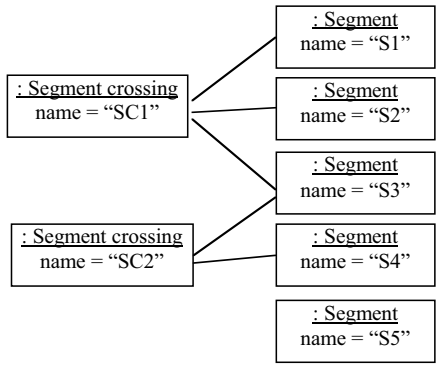


Figure 2.50 — *Object Diagram* incarnant la réalité de la figure 2.48.

Modèle abouti

Nous allons maintenant intégrer la notion de repère cartésien et donc la notion de coordonnée, de point ou de toute notion géométrique élémentaire (figure 2.51). La spécificité du modèle de la figure 2.51 est que seules les deux associations entre *Segment* et *Point* sont informatives, les autres étant dérivées et exprimées en tant que telles par des contraintes OCL. Nous avons conservé une formalisation de type UML 1.x, car ce sont les noms des associations (plutôt que les noms de rôles) qui sont préfixées d'un /.

Une observation pertinente est donc qu'on peut se limiter à un modèle plutôt fluet et préciser les contraintes OCL, ou, comme nous l'avons fait, être redondant mais maîtriser cette redondance en marquant bien les associations comme dérivées. Le texte de commentaire des assertions OCL donne tous les éléments de compréhension.

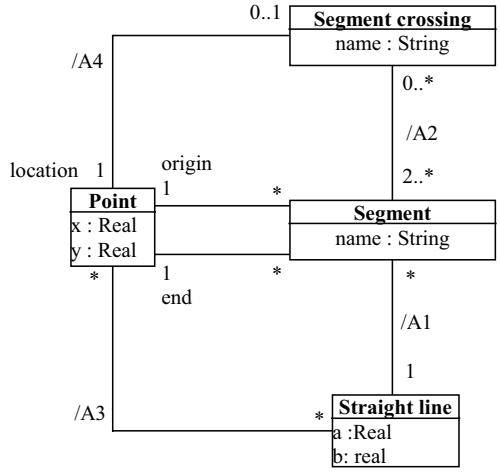


Figure 2.51 — Schéma final de l'exemple de synthèse

Un segment est tel que son origine et son extrémité sont différentes :

```
context Segment inv:
   origin.x = end.x implies origin.y <> end.y
```

Ensuite, la droite (*Straight line*) qui supporte un segment et qui est unique a son coefficient *a* et son déplacement *b* calculés comme suit :

```
context Segment inv A1:
   straight line = Straight line.allInstances→select(sl | sl.a =
      ➥ (end.y - origin.y) / (end.x - origin.x) and sl.b = end.y - sl.a * end.x)
```

De plus, la position d'un croisement de segment peut se calculer :

```
context Segment crossing inv A2:
   segment = Segment.allInstances→select(s | location.y = s.straight line.a
     ➥ * location.x + s.straight line.b)
```

L'appartenance d'un point à une droite a une formulation bien connue :

```
context Point inv A3:
   straight line→forAll(sl | y = sl.a * x + sl.b)
```

Un croisement de segment a une position (*location*) unique qui est un point. Si l'on prend tous les points sur les droites sur lesquelles se trouvent les segments qui concourent à sa définition, on sait que sa position appartient à cet ensemble (attention à l'opérateur OCL *includes* qui veut dire « appartient à » et non « inclus dans » comme on pourrait le penser).

```
context Segment crossing inv:
   segment.straight line.point→includes(location)
```

L'essentiel, ce qu'il faut retenir

Une approche par spécification formelle demanderait de déterminer et démontrer quel est l'ensemble minimal, nécessaire et suffisant d'assertions, non redondantes et non contradictoires, qui représenterait le système de la figure 2.48 de manière optimale et rationnelle, finalité qui n'a jamais été celle d'UML.

2.3.9 Package

Les packages n'ont qu'une finalité : l'organisation concise et logique des modèles. La figure 2.52 donne une représentation générale avec (en bas, à gauche de la figure) la possibilité de préfixer un élément de modèle par le nom du package auquel il appartient (réutilisation de l'opérateur de portée C++ : *::*). Sinon, les deux schémas de droite sont équivalents, le + cerclé désignant le statut d'appartenance.

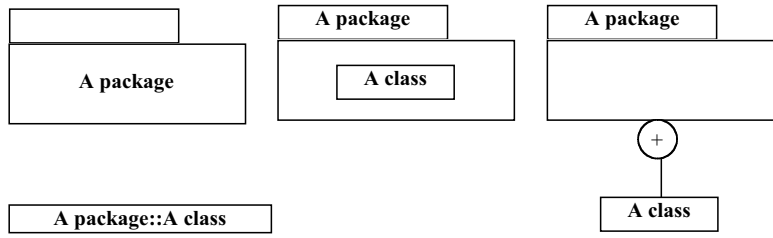


Figure 2.52 — Notations pour les packages en UML 1.x.

Il est particulièrement intéressant d'étudier comment relier les packages notamment tel que cela a été prévu en UML 2.x. Il faut bien distinguer une relation entre un package et une entité qui n'est pas un package (*A class* par exemple dans la figure 2.52) d'une relation entre packages.

UML 2.x Le formalisme 1 x de la figure 2.52 est maintenu en 2.x avec d'autres apports. Ainsi, *Package* hérite de *Namespace* dans le métamodèle. En clair, d'autres éléments de modèles disposent de possibilités de nommage pour des sous-éléments aussi appelés « membres » dans la documentation.

Les stéréotypes propres aux dépendances entre packages sont *«merge»*, *«access»*, *«use»* et *«import»*. *«merge»* étiquette une relation de dépendance où les membres du package pointé s'étoffent des membres du package « source » (*i.e.* le package d'où part la flèche symbolisant la relation de dépendance). Tout le métamodèle est fortement architecturé sur ce *«merge»*. Cela permet de spécialiser les concepts du métamodèle qui se dotent de métarelations et métapropriétés supplémentaires ou redéfinies, dans le package pointé. Ainsi le métaélément *Class* a de nombreuses spécialisations et/ou adaptations selon le package où il intervient, sa définition initiale étant dans *Kernel* (voir aussi figure 2.3).

A contrario, *«access»*, *«use»* et *«import»* sont plus dédiés aux modèles utilisateur. Les trois stéréotypes signifient globalement la même chose sauf qu'ils donnent des indications de visibilité différente : privé pour *«access»* et *«use»*, public pour *«import»*.

Dans la figure 2.53, l'extrémité des dépendances touche les classes mais elles peuvent toucher les packages si l'on veut tout importer. La non-importation de *P2::A* oblige à référencer cette dernière explicitement (opérateur *::*) dans *P1*, ce qui est le contraire pour *B* impliquée dans une association avec *X* et venant d'un autre package que *P1*. *P3* importe (*«use»*) toutes les classes membres de *P1* plus les classes préimportées, en l'occurrence *B* mais pas *C*, car la relation est *«access»* empêchant toute transitivité. Quelle est alors la différence entre *«access»* et *«use»* ? Il ne semble pas y en avoir en fait car la documentation [17] semble se contredire entre les pages 38 et 130. Page 38, *«access»* est la notation associée à l'import privé, alors que page 130, c'est à *«use»* qu'est dévolu ce rôle.

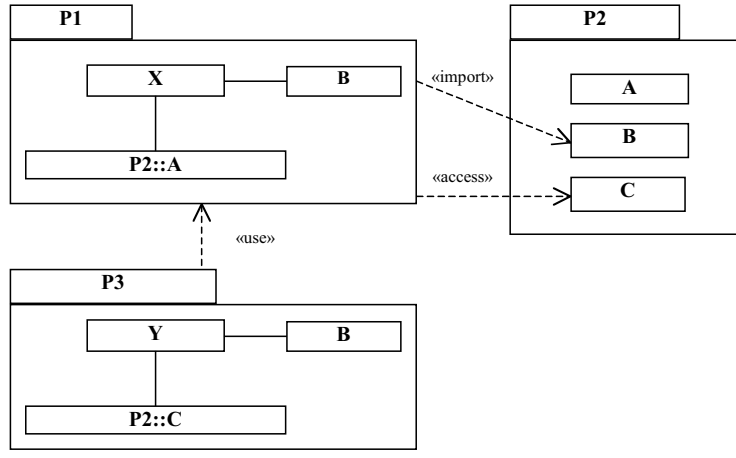


Figure 2.53 — Dépendances entre packages en UML 2.x.

L'essentiel, ce qu'il faut retenir

Une perspective intéressante d'utilisation des packages est de fabriquer des micro-modèles sur lesquels on s'applique à dégager un fort taux de réutilisation. En effet, soit les packages découpent les applications en « sous-systèmes » de manière à maîtriser la complexité, soit ils groupent des types/composants fortement cohérents entre eux via l'identification de leurs collaborations récurrentes et standard : c'est somme toute la réutilisation au niveau spécification.

2.3.10 Opération

La notion d'opération, fonction, méthode ou encore service est centrale en orienté objet. La figure 2.54 montre un type élémentaire (stéréotype «*datatype*») avec ses opérations. Comme ce type n'a pas d'attribut, on ne voit pas en lecture rapide (le deuxième compartiment disparaît) que les opérations sont notées dans le troisième et plus bas compartiment d'une boîte. C'est notre convention, certains outils UML faisant toujours apparaître le deuxième compartiment vide même s'il n'y a pas d'attribut.

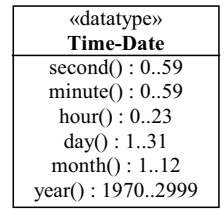


Figure 2.54 — Exemple de type primitif avec opérations.

Comme nous l'avons déjà dit, les opérations sont suffixées par des parenthèses avec à l'intérieur, leurs arguments ou rien comme c'est le cas pour toutes les opérations de la figure 2.54. Les types retournés sont ici des intervalles. Visuellement, un attribut se distingue donc facilement d'une opération sans paramètre, et ce en raison de la présence ou de l'absence de parenthèses. Les compartiments n'ont donc qu'un rôle de clarté de présentation.

Préconditions et post-conditions

Il est bien évident que la première attente relative aux opérations est la description de leur comportement. Quels sont leurs contenus ? Quels calculs assurent-elles ? Il n'est pas logique de faire cette description dans la partie *Structure* d'UML mais dans *Behavior* (voir chapitre 3). Cependant, le métatype *Operation* peut se voir attribuer des préconditions et post-conditions pour exprimer de manière déclarative les contrats des opérations. La figure 2.55 revient sur le cas des segments et des points de la

figure 2.51 et ajoute deux opérations permettant de translater et de retourner des segments.

Les préconditions et post-conditions peuvent se noter au sein même de la boîte, à la suite de l'opération concernée, ou en cas d'empêchement, via un commentaire incluant toutes les contraintes du type en général. Dans ce dernier cas, la notation OCL *pre:* et *post:* fait la distinction.

Dans la figure 2.55, on dit que l'attribut *name* de *Segment* doit être composé d'au moins un caractère. Les préconditions et post-conditions sur *rotate* et *translate* donnent les conditions d'évolution d'une instance de l'objet lui-même, mais aussi de ses relations de structure, comme par exemple les valeurs des navigations *origin* et *end* (extrémités des segments) qui vont probablement changer après l'invocation de *rotate* et *translate*.

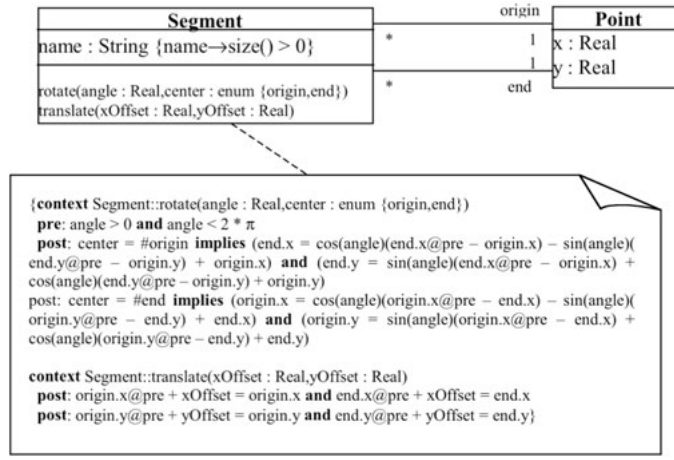


Figure 2.55 — Description de préconditions et post-conditions sur opérations.

Invariant et opération

La notion d'invariant n'est pas propre aux opérations mais il est intéressant de l'introduire ici, car elle est complémentaire de celles de précondition et de post-condition. En effet, en donnant un invariant de classe, on décrit par essence une partie du comportement des opérations : les altérations de valeurs (attributs, associations…) qui leur sont interdites.

L'idée simple dans la figure 2.56 est d'introduire le concept d'année bissextile via la fonction booléenne *leap year()*.

UML 2.x Nous attirons l'attention du lecteur sur le fait qu'UML 2.x a plutôt accentué l'influence de la notion d'invariant, ce qui est une bonne chose. L'invariant exprimé en figure 2.56 est en fait un invariant de classe basé sur des propriétés qui sont des opérations. D'autres invariants, inexistants en UML 1.x, sont maintenant possibles notamment ceux d'état (voir chapitre 3).

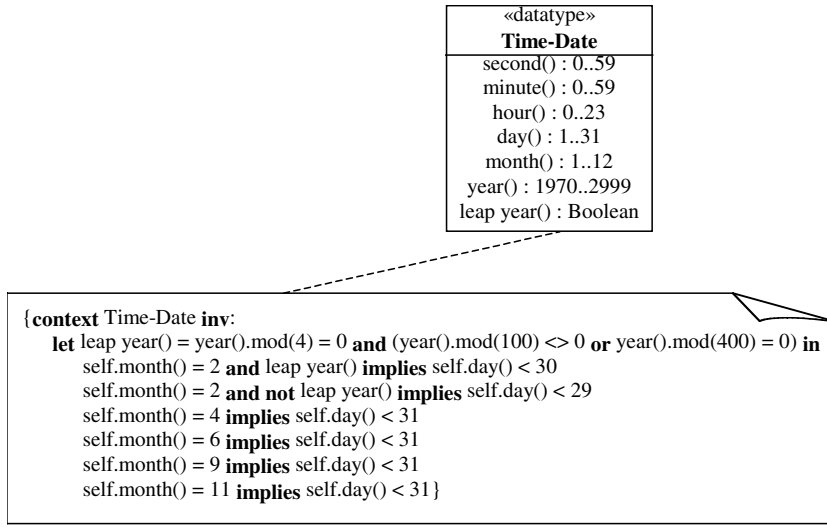


Figure 2.56 — Invariant d'opération.

Stéréotypes pour les opérations

Les stéréotypes *«constructor»*, *«query»* et *«update»* d'UML 1.x (figure 2.57) semblent avoir disparu d'UML 2.x (ils ne sont jamais utilisés dans la documentation). Dans notre discours, « semblent » est hésitant mais encore une fois, si l'on jette un coup d'œil à la liste des stéréotypes obsolètes, *«constructor»*, *«query»* et *«update»* n'y sont pas, il est donc difficile de trancher.

L'idée dans la figure 2.57 est de simuler les types abstraits de données (TAD, *Abstract Data Type*) en classant les opérations en opérations de construction, d'interrogation (pas d'effet de bord) et de mise à jour, qui sous l'angle de l'implémentation d'un TAD, est un changement éventuel d'état de l'instance à l'issue de l'invocation.

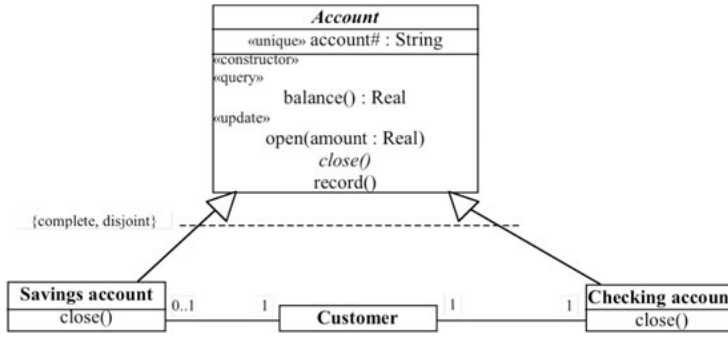


Figure 2.57 — Stéréotypes dédiés aux opérations.

Profitons-en pour décrire une opération abstraite (*close*) rendant sa classe de définition abstraite (*Account* en italique dans la figure 2.57). Les classes héritant deviennent concrètes par le fait que *close* (non écrit en italique dans *Savings account* et *Checking account*) se voit doter d'un comportement ou un corps en programmation.

Sinon, dans l'état actuel de la documentation, seuls *«create»* et *«destroy»* sont des stéréotypes prédéfinis des opérations. On pourrait donc remplacer *«constructor»* par *«create»* dans la figure 2.57, et utiliser *«destroy»* si l'on voulait décrire des opérations de destruction d'instances, ce qui paraît d'usage limité. On peut ainsi imaginer utiliser *«destroy»* pour stéréotyper les opérations de finalisation de Java (méthode *finalize()* d'*Object*) ou encore les fonctions de destruction de composants logiciels comme l'opération *ejbRemove()* dans la technologie EJB.

2.3.11 De l'analyse à la conception, encapsulation

L'usage concomitant d'attributs et d'opérations demande quelques précautions s'il existe des dépendances de calcul. En l'occurrence, dans la figure 2.58, l'attribut *age* est dérivé (préfixé d'un /). La contrainte OCL liée à la classe *Person* dit que *age* est l'instant courant moins la date de naissance (*birthdate*). Pour ce faire, nous introduisons une opération de classe *Now* (mot-clef *static* en C++ ou Java) soulignée (notation UML) ainsi que *minus* qui retourne une durée : type *Time-Date span*.

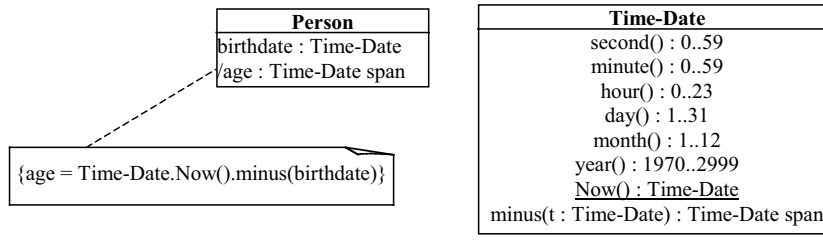


Figure 2.58 — Principe de référence uniforme en UML.

Pourquoi *age* est-il un attribut et non une opération, puisque c'est un calcul ? C'est en fait une fausse question, cela n'a pas d'importance. Le défaut d'UML est de ne pas supporter le principe de référence uniforme. Ainsi en spécification, dire qu'une propriété est un attribut (un champ en programmation) ou une opération, est un choix trop en avance, *i.e.* on aborde trop tôt des considérations d'implémentation. De ce point de vue, le choix fait en figure 2.58 est élégant et plus conceptuel. A *contrario*, une approche plus « programmation » donne le modèle de la figure 2.59. *age* devient une opération publique (+) alors que *birthdate* devient un champ privé (-). Outre le fait que l'on donne des indications de conception plus coercitives, on remarque que la date de naissance n'est plus une propriété questionnable car privée. À notre sens, des modèles comme celui de la figure 2.59 devraient se limiter à la conception objet (par opposition à l'analyse objet).

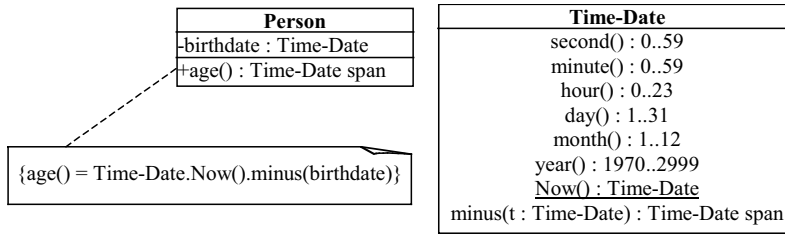


Figure 2.59 — Non application du principe de référence uniforme.

De manière plus générale, les indicateurs de visibilité +/*public*, -/*private* et #/*protected* sont plutôt liés à l'implémentation proprement dite. Dans la figure 2.60, on les utilise ouvertement pour rendre en particulier la propriété *birthdate* accessible aux clients du type *Person*. Le type *Time-Date* subit aussi l'application de ces indicateurs de visibilité : on introduit l'attribut caché *how it is internally recorded* qui est de type *time_t*, type bien connu du langage C pour coder un instant de manière binaire.

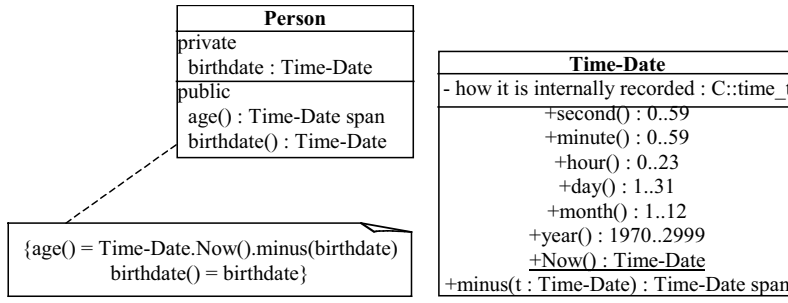


Figure 2.60 — Visibilité en UML.

Raffinement de types

Dans cette section, nous allons vous proposer une certaine manière de manipuler l'encapsulation en UML et donc expliquer comment gérer simultanément des types d'analyse et de conception. Pour cela, le stéréotype *«refine»* sur une relation de dépendance est en UML 2.x, une relation dite d'abstraction. L'extrait de la documentation [17, p. 14] montre bien à quoi ce stéréotype est destiné : « *A relationship that represents a fuller specification of something that has already been specified at a certain level of detail. For example, a design class is a refinement of an analysis class.* »

Ainsi préconisons-nous d'utiliser les packages pour scinder modèles d'analyse et modèles de conception. À ce titre, la figure 2.61 décrit une classe température, concept des sciences physiques s'il en est, permettant en particulier des conversions entre les unités de mesure les plus connues : Celsius, Fahrenheit et Kelvin.

La figure 2.62 est le modèle raffiné de conception. Cela apparaît clairement à travers le stéréotype *«refine»*. Le zéro Kelvin absolu (-273,15°C) est par exemple intro-

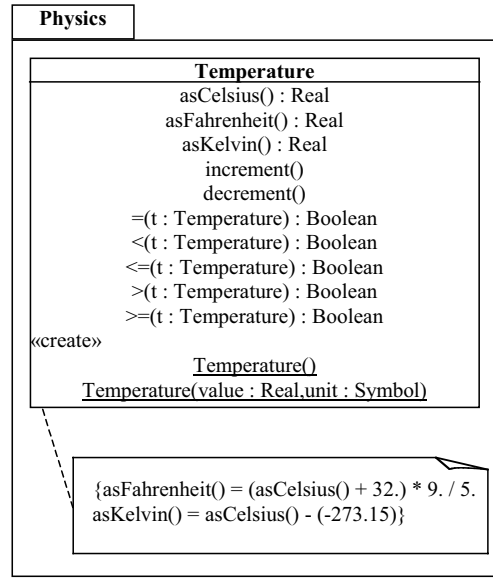


Figure 2.61 — Modèle d'analyse de *Temperature*.

duit comme une constante (*«const»*) publique (+) qui va pouvoir s'utiliser dans les calculs de conversion (c'est aussi une variable de classe car elle est soulignée). Une information importante dans la figure 2.62 est l'apparition de l'attribut privé *_value* qui va servir à coder la température dans la mémoire. Par ailleurs, les opérations logiques de comparaison changent de nom (*e.g.* = devient *equals*). Ce changement est tracé par la contrainte *{redefines =}* qui suffixe l'opération *equals* dans la figure 2.62.

En poursuivant le raffinement des modèles, il est important de savoir s'arrêter à temps afin de ne pas atteindre un niveau de détail redondant avec le code. Ainsi, à notre sens, le modèle de la figure 2.62 est suffisamment « mûr » pour passer au codage. En Java, cela peut donner :

```
public final class Temperature { // hériter de Temperature n'a pas de sens
                                 // d'où final
   public static final byte Celsius = 0;
   public static final byte Fahrenheit = 1;
   public static final byte Kelvin = 2;
   private static String[] _Literals = new String[3];
   static {
      Literals[Celsius] = new String("°C");
      Literals[Fahrenheit] = new String("°F");
      Literals[Kelvin] = new String("°K");
   }
   public final static float Min = -273.15F; // en Celsius
   private float _value; // en Celsius
   private float _step;
   // create
   public Temperature() {
      value = 0.F;
```

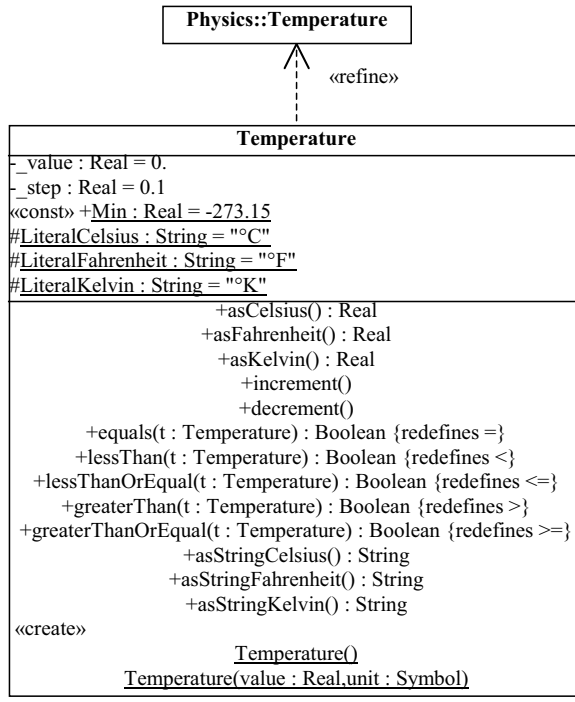


Figure 2.62 — Modèle de conception de *Temperature*.

```
        step = 0.0001F;
    }
  public Temperature(float value,byte unit) throws Invalid_temperature_exception
{
        this();
      …
    }
      …
}
```

Visibilité d'association

L'encapsulation dépasse le simple cas des classes elles-mêmes et s'applique aux associations (voire les agrégations et les compositions) qui peuvent être privées, protégées ou publiques. La figure 2.63 montre donc deux instances de *Temperature* (*ambient temperature* et *target temperature*) embarquées (ou encapsulées) dans la classe *Programmable thermostat*.

Tout à fait logiquement, les deux associations de la figure 2.63 sont orientées là où l'indicateur privé (-) est utilisé. En d'autres termes, les instances de *Temperature* ne voient, *i.e.* ne connaissent pas, l'instance de *Programmable thermostat* qui les manipule.

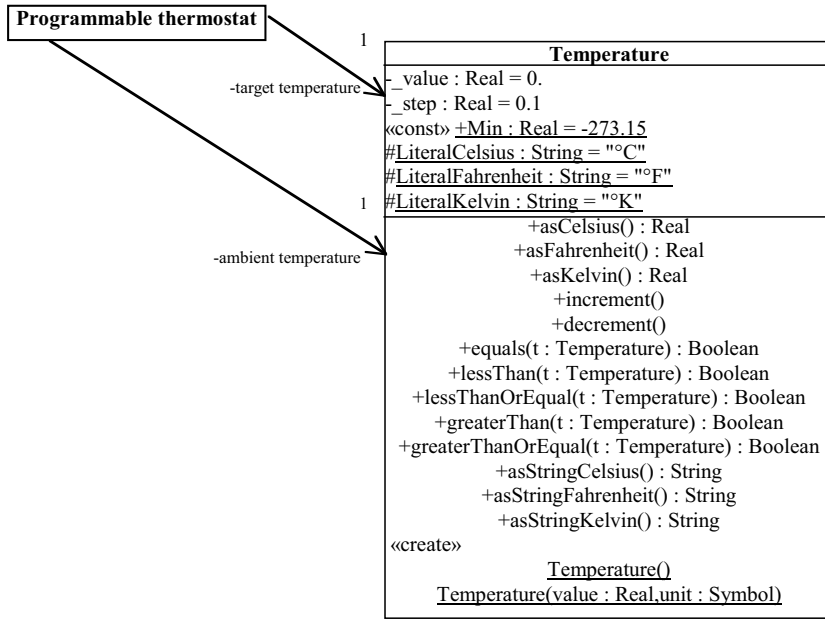


Figure 2.63 — Visibilité sur les associations.

Couplage faible contre couplage fort

Cette discussion nous amène au problème de couplage en conception. Simplement, l'idée est d'utiliser avec parcimonie les orientations en analyse. Comme nous l'avons dit, en analyse, orienter une association revient pour l'essentiel à empêcher des navigations qui du point de vue du métier, du domaine, ou encore des besoins, n'ont pas de sens ou constituent de l'information superflue.

En conception, les orientations sont des guides architecturaux. En clair, elles donnent le couplage qui doit être choisi le plus faible possible. Sans démonstration aucune, rappelons seulement que du point de vue de la maintenance, moins les types se connaissent, plus ils sont aptes à évoluer. En d'autres termes, modifier l'interface ou l'implémentation d'une classe requiert un investissement intellectuel sur la classe[1], mais aussi son environnement proche : les fonctions des types qu'elle appelle, les types retournés par ces fonctions…

La figure 2.64 caricature le problème en incitant à l'abandon de micro-architectures qui conduiraient à systématiser en conception les doubles orientations. Malheureusement, ce double pointage est parfois nécessaire (voir les chapitres 4, 5, et 6). En C++, cela pourrait donner :

```
class X;
class Y {
   X* _x; // pointeur obligatoire (i.e. X _x; impossible)
};
```

1. Attention au fait que la maintenance n'est pas toujours assurée par le concepteur initial de la classe !

```
class X {
   Y* _y; // le pointeur n'est pas obligatoire ici,
          // i.e. Y _y; possible et mieux si cardinalité 1..1
};
```

L'hypothèse faite est que les cardinalités sont *0..1* ou *1..1* dans le modèle de gauche de la figure 2.64.

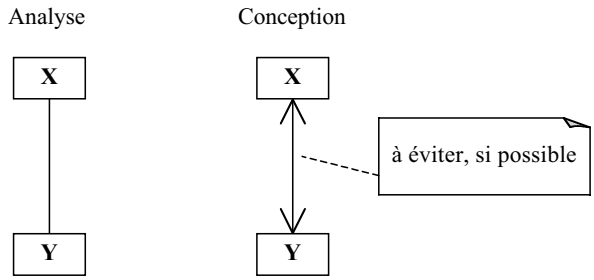


Figure 2.64 — Description du couplage via l'orientation des associations.

2.3.12 En direction de l'implémentation

En allant plus loin avec UML, on s'approche des notations de plus en plus spécifiques et fortement liées à de l'implémentation pure et dure.

Généricité

Le cas de la généricité (au sens d'Ada), également connue via l'expression *template class* en C++ ou via le mot *generics* en Java, fait dériver vers des notations de portée limitée. Par exemple Smalltalk, ou encore Java jusqu'à peu, ne supportant pas la généricité, sont de fait évincés comme langage cible, si l'on a à implanter des modèles comme celui de la figure 2.65.

> **Le danger, ce dont il faut être conscient**
>
> On peut s'interroger sur l'intérêt de telles constructions de modélisation : représenter graphiquement du code existant (*reengineering*) ? Guider scrupuleusement l'implémentation ? Oui mais dans ce dernier cas, avec quelle valeur ajoutée et quel coût ? En effet, l'expérience montre que les équipes de développement souffrent avec UML du manque d'aspects concrets. La production de modèles doit donc être au plus courte dans le temps, même si ces derniers ne sont pas bien finalisés, pour rendre attractif le langage UML lui-même.

Pour illustrer la généricité, la figure 2.65 propose une instance du métatype *TemplateableElement* appelée *Association* (qui n'a rien à voir par ailleurs avec le métatype *Association* du noyau d'UML) et paramétrée par les types formels *Black* et *White*. C'est en fait une classe C++ prédéfinie sous la forme : *template <typename Black,type-*

name White> class Association {…};. Cette classe sert à automatiser l'implantation des associations et fait partie de la bibliothèque *cOlOr* [2].

UML 2.x La notation de la figure 2.65 est celle d'UML 2.x, qui a vraiment changé au regard de celle de 1.x. Le lecteur doit en particulier se méfier de la flèche entre la classe générique *Association* et la classe inférée sur la base de cette classe générique, en l'occurrence la classe appelée *Company-Site Association*.

Dans cet exemple, il s'agit d'implémenter la relation d'association entre *Site* et *Company* qui vient de l'analyse, par cette classe *Company-Site Association* (classe de conception par essence) ainsi qu'un lien de *Site* vers elle et un lien de *Company* vers elle. En bilan, les instances de *Site* et de *Company* ont accès par navigation aux instances de l'autre auxquelles elles sont contractuellement attachées, cela résultant de l'association simple entre *Site* et *Company* en haut, à gauche de la figure 2.65.

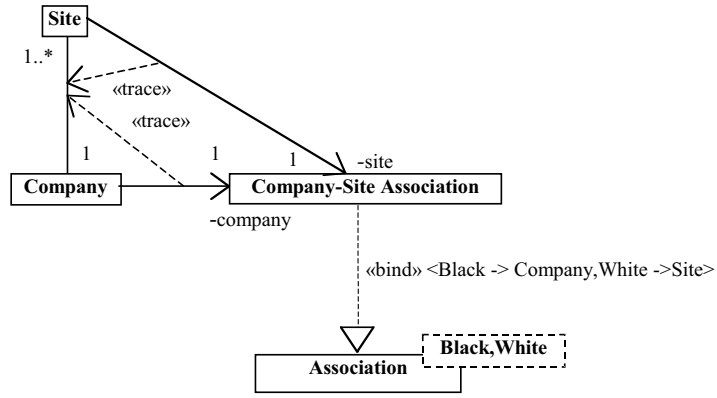


Figure 2.65 — Généricité en UML et stéréotype *«trace»*.

Le stéréotype *«trace»* est comme *«refine»* dédié à la relation dite d'abstraction et montre là que les associations nouvelles (aussi de conception) entre *Company-Site Association* et *Site* ainsi qu'entre *Company-Site Association* et *Company*, tracent l'implantation de l'association initiale entre *Site* et *Company*.

La figure 2.66 complète juste les notations possibles. La dépendance de type *«bind»* n'est pas utilisée. La classe résultante de l'utilisation d'*Association<Black,White>* est inférée en disant que le type formel *Black* est remplacé par le type réel *Speaker* et le type formel *White* est remplacé par le type réel *Conference*.

Modèles techniques

Il est tentant d'utiliser UML pour décrire des bibliothèques de classes préexistantes. Par exemple, une partie des classes et interfaces servant de support à MVC (Model/Vue/Contrôle) dans Java (figure 2.67), permet au programmeur de ne pas plonger dans le code ou la documentation Java standard (pages HTML) pour tout d'abord comprendre comment fonctionne MVC. Ensuite, il peut chercher à relier des modè-

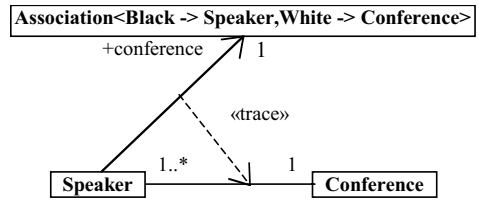


Figure 2.66 — Autre notation de la généricité en UML.

les « proches du terrain » comme celui de la figure 2.67, à des modèles plus conceptuels (analyse) ou d'architecture (conception). Nous nous y appliquons dans le chapitre 5 en mettant en œuvre l'interface *PropertyChangeListener* de la figure 2.67.

Comme le montre la figure 2.67, les actions de la bibliothèque *Swing* qui sert en Java à concevoir les interfaces utilisateur sont des sortes (liens d'héritage) d'écouteurs d'événements (*java::util::EventListener*)[1]. Les événements sont eux-mêmes typés en partant de l'interface généraliste *java::util::EventObject*.

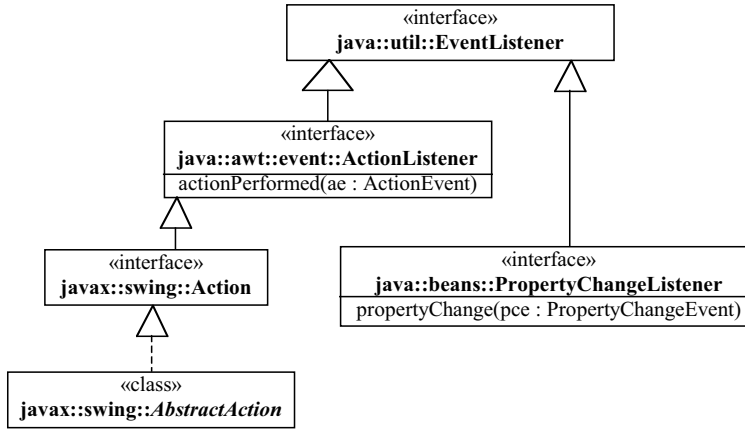


Figure 2.67 — Dépendances des deux notions centrales de *JavaBeans* : *PropertyChangeEvent* et *PropertyChangeListener*.

La démarche pragmatique qui veut que les modèles UML « collent » au code est favorisée par les ateliers de génie logiciel qui peuvent aisément remonter des modèles UML à partir du code, mais avec le risque élevé que d'un outil à l'autre, les modèles établis soient différents alors que le code est le même (interprétation antagoniste de la sémantique UML par les concepteurs d'AGL).

1. Les packages sont séparés par des points en Java mais nous utilisons ici la notation UML pour décrire l'emboîtement des packages, à savoir le symbole ::.

Le danger, ce dont il faut être conscient

Dans ce contexte, une dérive importante de l'usage d'UML dans l'industrie est de réduire UML à la cartographie de code à cause de la difficulté, encore une fois, à différencier analyse, conception et implémentation. Il existe aussi une impasse découlant de la quantité de classes qui peuvent exister : imaginons la description UML de toutes les classes et interfaces Java de J2SE, voire mêmes celles de J2EE, J2ME… à laquelle peut s'ajouter la description de classes et interfaces « maison ». Comment gérer avec efficacité et de façon rationnelle de tels modèles UML qui graphiquement seraient immenses ? « Modéliser le code[a] » doit se faire avec parcimonie.

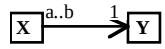
a. L'expression est un peu stupide en soi, mais reste parlante par son caractère pragmatique.

Implémentation des associations et compositions en C++ : la bibliothèque cOlOr

Dans cette section, nous allons illustrer l'idée générale d'« anticipation », et ce au regard du développement d'un outillage dédié à l'implantation directe et rapide de modèles UML. Soit on dépend fortement d'un AGL et donc du type de code qu'il génère, soit on utilise UML plus librement et on veut dans ce cas mieux contrôler et/ou personnaliser le code produit. C'est un dilemme industriel important, car il est difficile de travailler sans outil, mais souvent la nécessité de contourner un AGL est aussi importante.

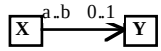
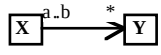
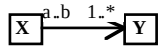
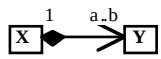
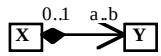
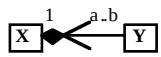
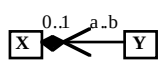
La bibliothèque C++ *cOlOr* [2] a pour but l'automatisation de l'implémentation des associations. Elle est adossée à la bibliothèque STL évoquée au chapitre 1. *cOlOr* prolonge donc toute la puissance de STL dont ses algorithmes génériques. Le tableau qui suit est un guide méthodologique. À partir d'une association (colonne de gauche), il donne le code C++ (colonne de droite) : plusieurs possibilités sont ainsi numérotées. En fonction des directions retenues, des cardinalités et des noms de rôles pour l'essentiel, le code joue sur les pointeurs (* en C++), les références (*&* en C++) et sur la classe *template* C++ prédéfinie *Association<Black,White>*[1] (voir figure 2.65 où cette classe générique est dépeinte en UML). Le tableau donne aussi l'écriture d'une partie des constructeurs des types qui vont embarquer l'association.

Tableau 2.2 – Guide méthodologique d'associations UML en code C++

UML	C++, déclaration dans la classe X ou dans la classe Y
	1. Y _y; 2. Y& _y; 3. Y*& _y; 4. Association<X,Y> _y; constructor: _y(Association<X,Y>(a,b,1,1))

1. Il existe aussi la classe prédéfinie *Composition<Black,White>* mise en œuvre dans le tableau et dédiée au losange noir. Pour le losange blanc, sa sémantique lâche fait qu'il faut utiliser *Association<Black,White>*.

Tableau 2.2 – Guide méthodologique d'associations UML en code C++

	1. Y* _y; 2. const Y* _y; 3. Association<X,Y> _y; constructor: _y(Association<X,Y>(a,b,0,1))
	Association<X,Y> _y; constructor: _y(Association<X,Y>(a,b,0,INT_MAX))
	Association<X,Y> _y; constructor: _y(Association<X,Y>(a,b,1,INT_MAX))
	1. Composition<X,Y> _y; constructor: _y(Composition<X,Y>(1,1,a,b)) 2. // a= 0, b = 1 Y *const _y; const Y *const _y; 3. // a= 1, b = 1 const Y _y; const Y& _y; const Y*& _y;
	1. Composition<X,Y> _y; constructor: _y(Composition<X,Y>(0,1,a,b)) 2. // a= 0, b = 1 Y *const _y; const Y *const _y; 3. // a= 1, b = 1 const Y _y; const Y& _y; const Y*& _y;
	Composition<X,Y> _x; constructor: _x(Composition<X,Y>(1,1,a,b))
	Composition<X,Y> _x; constructor: _x(Composition<X,Y>(0,1,a,b))

La figure 2.68 donne une illustration de la manière de réutiliser *cOlOr*. À gauche figure le modèle d'analyse où aucun mode particulier de référencement n'est retenu. Le modèle de conception crée au contraire un couplage fort (double orientation de l'association) qui permet à toute instance de *X* de connaître les instances de *Y* qui lui

sont liées, et vice-versa. Le champ *y* qui représente dans *X* l'association est public alors que le champ *_x* dans *Y* est privé. Les constructeurs sont formés de telle sorte que la relation reste cohérente. En clair, les valeurs des champs *y* et *_x* font d'une certaine manière doublon. Ce choix reste cependant respectable, car il peut améliorer la performance alors que comme nous l'avons déjà dit, le couplage fort est en général à éviter. Ainsi, on ne peut créer une instance de *Y* qu'en exécutant *Y y1(x1);* engendrant la mise à jour automatique du champ *y* dans *x1*.

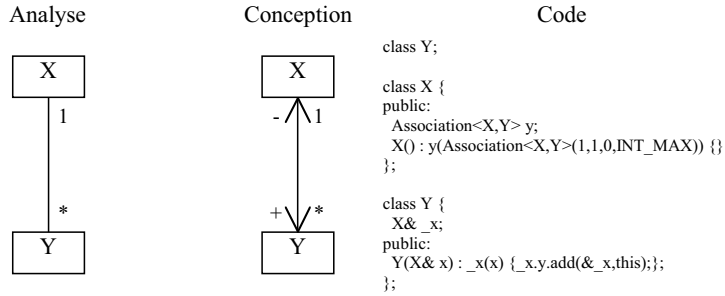


Figure 2.68 — Exemple de mise en œuvre de la bibliothèque C++ *cOlOr*.

2.4 COMPOSANTS LOGICIELS ET DÉPLOIEMENT

Les *Component Diagram*, *Composite Structure Diagram*[1] (absents d'UML 1.x) et *Deployment Diagram* appartiennent à *Structure* dans UML 2.x. Nous les présentons succinctement ici pour y revenir via un exemple concret dans le chapitre 6, notamment. Rappelons avec insistance que les *Component Diagram* et *Deployment Diagram* étaient considérés dans UML 1.x comme des diagrammes d'implémentation (voir section 2.2 de ce chapitre).

Le stéréotype *«component»* est utilisé pour qualifier une *Class* (le métatype du package *StructuredClasses* précisément, figure 2.3) de composant logiciel (figure 2.69). Les *provided interface* (boule blanche à l'extrémité du trait partant du composant) sont des sous-ensembles de l'ensemble des services offerts. Dans la figure 2.69, *My functional interface*, *My testing interface* et *My configuration interface* sont par exemple de type *provided interface*.

UML 2.x En UML 1.x, le concept de *required interface* est absent. Dans la figure 2.69 et en UML 2.x, il possède une symbolique propre qui est le demi-cercle intitulé *My required interface* en bas de la figure. Ce demi-cercle n'a pas été choisi de manière quelconque : un cercle représentant une *provided interface* s'insère naturellement

1. Notons que ces types de diagramme sont « transversaux » aux autres types de diagramme. Ils sont à ce titre aussi étudiés au chapitre 3 tout à fait indépendamment des composants logiciels, alors qu'ils sont utilisés en totale conjonction des *Component Diagram* au chapitre 6.

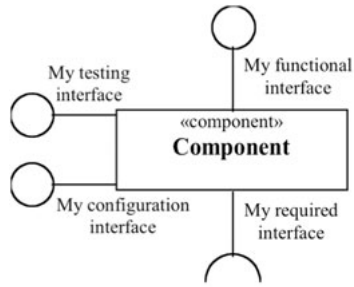


Figure 2.69 — Présentation canonique d'un composant logiciel en UML 2.x.

dans un demi-cercle représentant lui une *required interface*, cela signifie alors l'assemblage de deux composants.

2.4.1 Modèles de composants et profils

L'existence de modèles de composants technologiques dont les plus connus sont CCM (*CORBA Component Model*) de l'OMG, EJB de Sun ainsi que COM/DCOM et .NET de Microsoft a amené la création de profils, aménagements et extensions du métamodèle d'UML 2.x. La figure 2.70 donne celui d'EJB avec des corrections et adjonctions qui nous sont propres (en grisé). La partie droite de la figure 2.70 est notre propre création et montre la différence entre d'une part, créer de nouveaux stéréotypes assignables aux métatypes d'objet initiaux d'UML, et, d'autre part, étendre ces métatypes par de nouveaux. Par exemple, une nouvelle métaclasse *EJBHome interface* est vue comme un descendant direct de la métaclasse *Interface* qui est native en UML. Cela est notre propre vision et ne fait pas partie d'UML 2.x à ce jour. En revanche, la partie gauche non grisée est le profil *J2EE/EJB*. Nous avons rajouté tout en bas (en grisé) les stéréotypes *EJBStatelessSessionBean* et *EJBStatefulSessionBean* car ils sont absents[1] d'UML 2.x ce qui est gênant dans les modèles utilisateur (voir notamment le chapitre 4).

La fabrication d'un modèle utilisateur à partir de ce qui est offert en figure 2.70 donne la figure 2.71 : un EJB de la catégorie *Stateless Session Bean* est composé (losange noir) d'une classe d'implémentation, d'une *Home Interface* et d'une *Remote Interface* (voir chapitre 4 pour un modèle plus générique et complet).

On modélise en figure 2.71, un contrôleur d'autorisation bancaire (voir aussi chapitre 6) qui assure un service métier (stéréotype prédéfini dans UML 2.x : *«EJB business»*) et possède des propriétés système : son nom de localisation sur le réseau (*JNDI name*) qui est une variable de classe (soulignement) et une méthode de création sans paramètre, retournant une *Remote Interface* et disponible sur la *Home Interface*.

1. Dans les faits, ils sont quand même gérés via des valeurs marquées (*tagged values*), ce qui reste de notre point de vue insuffisant pour fabriquer des modèles utilisateur.

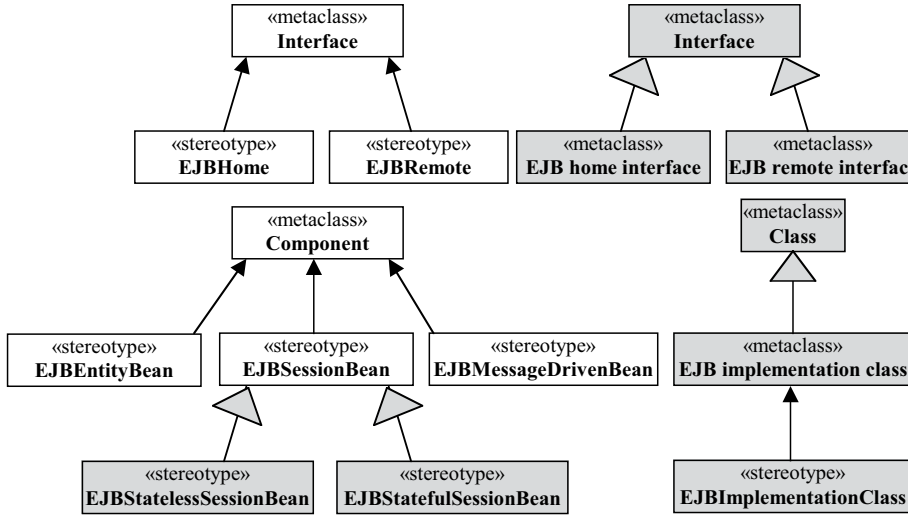


Figure 2.70 — Relation d'extension (gauche) pour créer de nouveaux stéréotypes contre création de nouveaux métatypes d'objet (droite) via l'héritage.

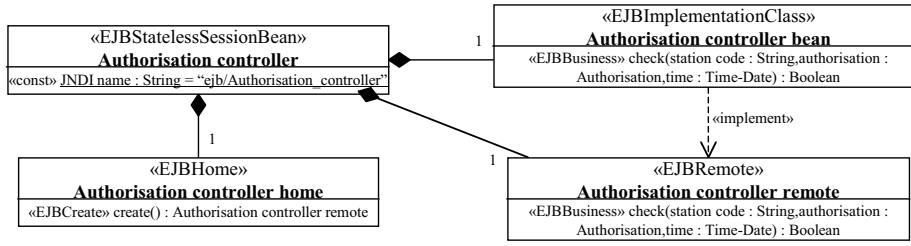


Figure 2.71 — Mise en œuvre des stéréotypes dédiés aux composants logiciels de type EJB.

Attention à la dépendance labellisée *«implement»* dans la figure 2.71 qui n'est vraiment que conceptuelle. Dans l'esprit des EJB, on doit impérativement créer une classe concrète Java (c'est le serveur d'EJB qui en l'occurrence le fait) qui supporte exactement et scrupuleusement tous les services métier listés dans la *Remote Interface*. Toutefois, dans le code en dur, il n'y a pas de lien entre les deux entités. Si l'on code l'exemple en l'occurrence, la construction Java *implements* entre la classe *Authorisation_controller_bean* et l'interface *Authorisation_controller_remote* ne doit donc pas être mise en œuvre dans l'esprit des EJB.

2.4.2 Déploiement

Les diagrammes de déploiement restent assez exotiques et marginales quant à la portée de ce qu'ils modélisent. Dans la figure 2.72 par exemple, dire qu'un fichier Java *Authorisation_controller.jar* (stéréotype *«artifact»*) sert de manifeste au composant

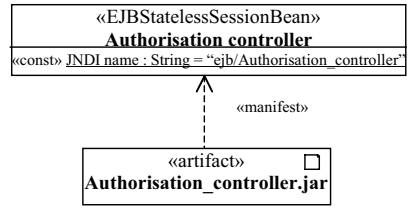


Figure 2.72 — Mise en œuvre du stéréotype *«artifact»*.

logiciel *Authorisation controller* sur son nœud de déploiement, relève plus de l'information que de la véritable spécification.

Au-delà, une partie *«artifacts»* est présélectionnée dans les boîtes décrivant les composants logiciels. Pour information, le petit dessin (deux rectangles chevauchant partiellement un troisième) en haut à droite de la figure 2.73 est une alternative au stéréotype *«component»*. Attention donc au modèle de la figure 2.73 qui est un *Component Diagram* et non un *Deployment Diagram* : c'est une alternative de modélisation à la figure 2.71 avec en plus des informations de déploiement. Mais alors pourquoi utiliser les *Component Diagram* pour faire figurer des informations de déploiement alors que justement on a inventé les *Deployment Diagram* ?

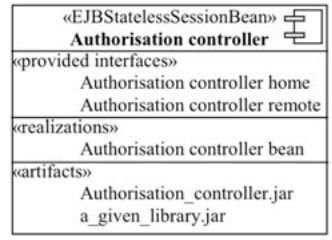


Figure 2.73 — Vision duale de l'artefact comme moyen d'implémentation.

D'autres stéréotypes et/ou formalismes prédéfinis permettent dans les modèles de déploiement de décrire des dépendances comme le besoin d'une bibliothèque résidente sur un nœud (*ojdbc14.jar*) ou l'adjonction d'un fichier XML pour faire office de spécification de déploiement dans une instance de serveur J2EE (figure 2.74).

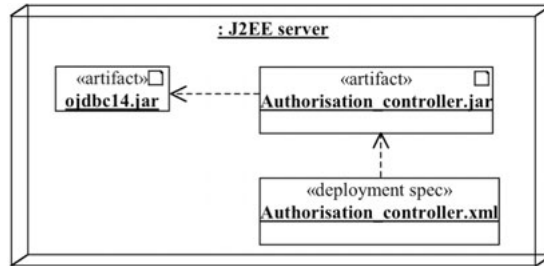


Figure 2.74 — Nœuds de déploiement au niveau instance.

Le métatype spécification de déploiement (*DeploymentSpecification*, stéréotype *«deployment spec»*) hérite de celui d'artefact dans le métamodèle UML 2.x. La distinction se fait cependant graphiquement. Dans la figure 2.74, le symbole figurant dans les boîtes correspondant à /ojdbc14.jar/ et à /Authorisation_controller.jar/ (en haut à droite de chaque boîte) n'est réservé qu'aux instances du métatype /Artifact/. Tout soulignement fait que l'on est au niveau instance (*cf. J2EE server* dans la figure 2.74 qui est une instance du métatype *Node* héritant lui-même de *Class*). Dans la figure 2.75, on est au niveau type.

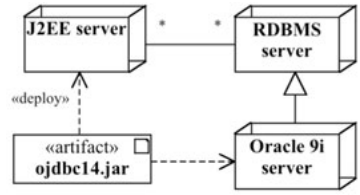


Figure 2.75 — Nœuds de déploiement au niveau type.

2.5 ÉTUDES DE CAS AVEC EXERCICES

Nous proposons deux études de cas simples dont le but est de décrire deux modèles entité/relation à la mode UML. Le premier (cas GPAO, gestion de production assistée par ordinateur) est une entreprise de fabrication de camions jouets en plastique. Les données sont les articles, leurs attributs et leurs valeurs (tableau 2.3 de la section 2.5.1).

2.5.1 Cas GPAO

Les données du cas GPAO sont les suivantes :

Tableau 2.3 – Données du cas GPAO

Réf.	Désignation	Type fabrication ou achat	Unité achat ou stock	Délai (sem.)	Prix standard	Lot de réappro.	Stock mini.	Stock maxi.
CD100	Camion déménagement bleu	Fab. par lot	À l'unité	2		200		600
CC201	Camion citerne rouge	Fab. à la commande	À l'unité	2		150		600
CA000	Cabine montée bleue	Fab. par lot	À l'unité	2		250		750
CA001	Cabine montée rouge	Fab. à la commande	À l'unité	2		150		

Tableau 2.3 – Données du cas GPAO

CH005	Chassis monté	Fab. par lot	À l'unité	1		300		900
C000	Cabine bleue	Fab. par lot	À l'unité	1		250		750
C001	Cabine rouge	Fab. à la commande	À l'unité	1		150		
C004	Chassis	Achat à la commande	À l'unité	2	4		300	
ES000	Essieu monté	Fab. par lot	À l'unité	2		500	750	1 500
H000	Conteneur bleu	Fab. par lot	À l'unité	1		150	350	800
H001	Conteneur bleu spécial	Fab. à la commande	À l'unité	1		150	350	
M000	Moteur bleu	Fab. par lot	À l'unité	1		250	150	800
M001	Moteur rouge	Fab. à la commande	À l'unité	1		150	150	
P004	Pare-chocs	Achat par lot	À l'unité	2	1	500	300	1 500
P005	Phare normal	Achat par lot	À l'unité	2	0,75	500	750	1 500
P006	Phare à iode	Achat à la commande	À l'unité	2	1,20		300	
ROUE50	Roue de camion	Achat par lot	À l'unité	6	1,50	500	500	2 000
T001	Citerne rouge	Fab. à la commande	À l'unité	1		150	300	
V004	Pare-brise	Achat à la commande	À l'unité	2	1,45		500	
[a]ABS501	Plastique rouge	Achat à la commande	Au kilo	2	18		100	
[a]ABS502	Plastique bleu	Achat à la commande	Au kilo	2	18		100	
V005	Verre de portière	Achat par lot	À l'unité	2	0,50	500	300	1 500
V006	Pare-brise teinté	Achat à la commande	À l'unité	3	1,20		300	
[a]BAR103	Rond d'acier	Achat par lot	Au mètre	3	5	250	250	800

a. Donner un % de perte de 5 % sur ces matières.

Le tableau 2.3 se complète par une représentation arborescente de la nomenclature de fabrication (figure 2.76). Les chiffres sur les liens de nomenclature sont les quantités nécessitées à la fabrication.

La compréhension et l'évaluation du modèle solution (voir section 2.5.2) sont laissées à l'appréciation du lecteur avec un petit exercice OCL pour se forcer à améliorer le modèle. Concernant la seconde étude de cas (voir section 2.5.3), à la diffé-

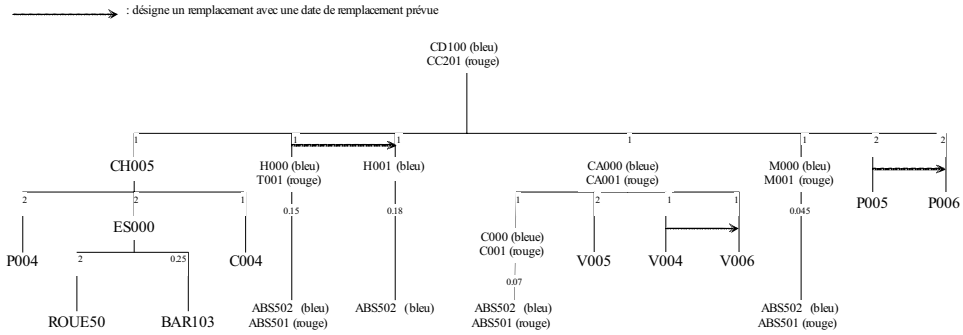


Figure 2.76 — Cas GPAO, nomenclature da fabrication.

rence de la première, les contraintes OCL sont fournies en sus du modèle solution (voir section 2.5.4).

2.5.2 Solution du cas GPAO

En UML, la solution de l'exercice donne la figure 2.77.

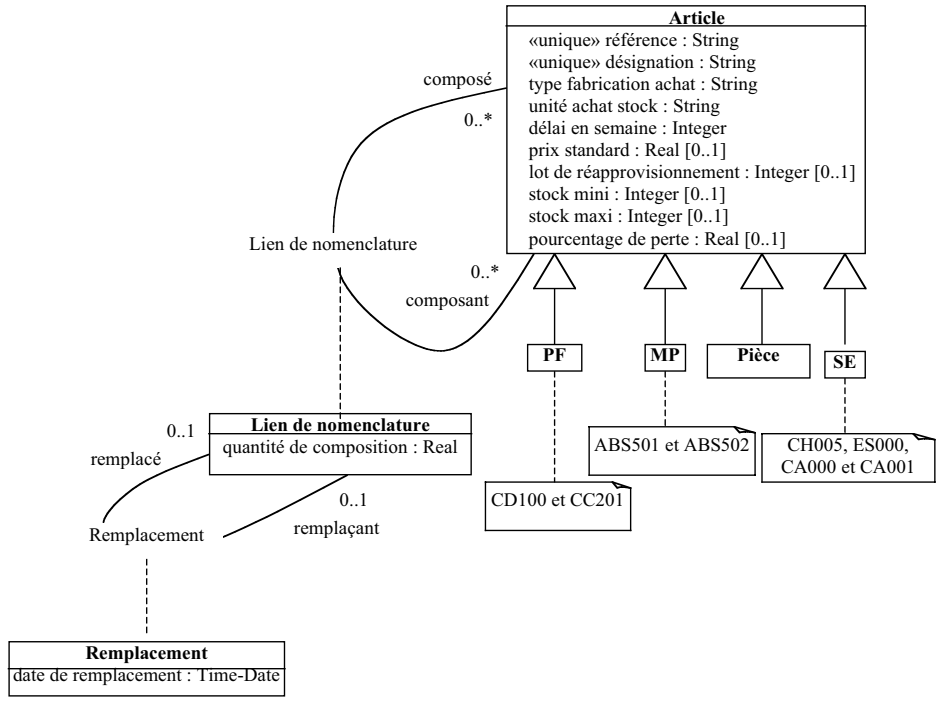


Figure 2.77 — Cas GPAO, *Class Diagram* relatif aux articles et à la nomenclature de fabrication.

Exprimer en OCL les contraintes suivantes :

- le composé et le composant d'un lien de nomenclature ne peuvent pas être le même article ;
- le remplacé et le remplaçant d'un remplacement ne peuvent pas être le même lien de nomenclature ;
- et il ne peut y avoir de liens de nomenclature qu'entre *PF* et *SE*, *PF* et *Pièce*, *SE* et *Pièce* ainsi qu'entre *Pièce* et *MP*.

2.5.3 Cas QUALIF

Le service des achats de la QUALIF, petite PME, souhaite étudier les informations relatives à la qualification des fournisseurs. Pour ce faire, cette entreprise établit une liste d'informations pertinentes :

- l'adresse postale d'un fournisseur ;
- la capacité de production d'un produit fabriqué et vendu par un fournisseur ;
- le code d'un produit fabriqué et vendu par un fournisseur (identifiant d'un produit fabriqué et vendu par un fournisseur) ;
- la date de qualification d'un fournisseur par un service d'étude ;
- le domaine d'activité d'un service d'étude ;
- le nom commercial d'un produit fabriqué et vendu par un fournisseur ;
- le nom du responsable d'un service d'étude ;
- le nombre de salariés d'un fournisseur ;
- le numéro de qualification d'un fournisseur par un service d'étude ;
- le numéro de SIRET d'un fournisseur (identifiant d'un fournisseur) ;
- le numéro de télécopie d'un fournisseur ;
- le numéro d'un service d'étude (identifiant d'un service d'étude) ;
- le pourcentage de participation d'un fournisseur chez un autre fournisseur ;
- le prix d'achat HT d'un produit acheté, déterminé par un service d'étude ;
- le prix de vente HT d'un produit fixé par un fournisseur ;
- la quantité prévisionnelle d'achat d'un produit acheté, déterminée par un service d'étude ;
- le code d'un produit acheté, déterminé par un service d'étude (identifiant d'un produit acheté par un service d'étude) ;
- le résultat avant impôt d'un fournisseur sur les trois dernières années ;
- le résultat après impôt d'un fournisseur sur les trois dernières années ;
- l'unité de mesure d'un produit acheté, déterminée par un service d'étude.

Les règles de gestion de l'entreprise décrivant la technique de qualification sont :

- un service d'étude peut qualifier autant de fournisseurs qu'il le souhaite ;

- un fournisseur effectuant une demande de qualification auprès d'un service d'étude se voit qualifier ou non ;
- un même fournisseur peut être qualifié par un service d'étude et ne pas l'être par un autre ;
- un même produit X fabriqué et vendu par un fournisseur peut correspondre à plusieurs produits achetés, en fait à autant de produits achetés qu'il y a de services d'étude qui ont qualifié ce fournisseur en vue de lui acheter en particulier ce produit X ;
- après qualification d'un fournisseur, un service d'étude établit la liste des produits achetés auprès de ce fournisseur ainsi que le prix d'achat HT et la quantité prévisionnelle d'achat ;
- un fournisseur peut avoir une prise de participation chez plusieurs autres fournisseurs.

2.5.4 Solution du cas QUALIF

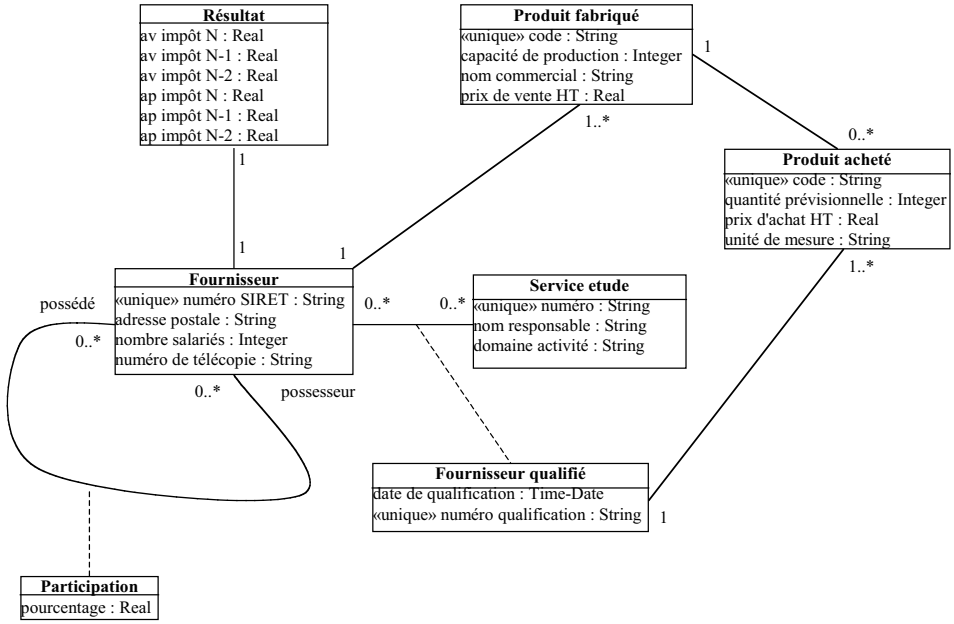


Figure 2.78 — Cas QUALIF.

La contrainte OCL qui s'ajoute au modèle de la figure 2.78 est que les produits fabriqués par un fournisseur donné sont un surensemble des produits pour lesquels il s'est fait qualifié (*i.e.* les produits dits achetés) :

```
context Fournisseur inv:
   produit fabriqué→includesAll(fournisseur qualifié.produit
      ➥ acheté.produit fabriqué)
```

2.6 CONCLUSION

Ce chapitre est indéniablement une vision critique d'UML pour la modélisation des aspects statiques des systèmes. Il est difficile parfois d'exploiter la documentation, qui est touffue, ambiguë et contradictoire. Une présentation impartiale demande une certaine exhaustivité mais la liberté de penser n'empêche pas de pointer les grands bogues difficilement admissibles, à un niveau d'usage professionnel surtout.

Nous avons toujours été étrangement surpris par la tolérance des utilisateurs à l'égard d'UML. Ceux qui échouent dans sa mise en œuvre croient que c'est par manque de connaissances en spécification et/ou de compétences sur la technologie objet. Pour les spécialistes UML, bon nombre lisent et relisent la documentation, croisent les définitions, les comparent, les composent, et, avec un peu de pratique et d'expérience, arrivent à manipuler sans trop d'effets de bord une construction de modélisation donnée. Ce qu'ils oublient peut être, c'est qu'inconsciemment, ils opèrent un défrichage pour arriver à une stabilisation de la sémantique qui n'aurait pas lieu d'être si le mode d'élaboration d'UML à l'OMG était rigoureux.

Peu savent que le processus de création d'UML est une bataille de chiffonniers où le but est de maximiser l'impact et l'influence de sa propre contribution (appel d'offres de contribution de deux ordres : *Request for Information*, *Request for Proposals*) au détriment de la concision et de la cohérence. C'est de bonne guerre économique : si vous arrivez à imposer dans une norme ce qui est votre propre création, ce que vous maîtrisez parfaitement ou que vous avez déjà implanté dans un atelier de génie logiciel, vous êtes plus crédible par rapport à la clientèle. J'ai personnellement participé à ce petit jeu pour la fabrication de la version 2.0 [8-9][1] avec des résultats médiocres de par le faible poids du consortium DSTC avec qui j'ai travaillé (voir la section 2.8 « Webographie »). Beaucoup d'organisations à but lucratif qui ont participé à la définition des différentes versions d'UML ont donc préféré viser le « business » plutôt que l'établissement d'un langage de modélisation « au-dessus de tous soupçons ». De façon alternative, le groupe de recherche pUML (voir aussi la section 2.8) par exemple, s'est monté pour corriger la tendance qui fait qu'UML est un capharnaüm. Dans le monde scientifique, c'est justement ce capharnaüm qui sert d'excitateur et d'émulateur à la recherche, aujourd'hui toujours vigoureuse autour d'UML.

2.7 BIBLIOGRAPHIE

1. André, P, and Vailly, A. : *Spécification des logiciels – deux exemples de pratiques récentes : Z et UML*, Ellipses (2001)
2. Barbier, F., and Henderson-Sellers, B. : "The Whole-Part Relationship in Object Modeling: A Definition in cOlOr", *Information and Software Technology*, 43(1), (2001), 19-39

1. Les deux documents précités sont sur le site de l'OMG mais avec l'accès *member*.

3. Barbier, F., Le Parc-Lacayrelle, A., and Bruel J.-M. : « Agrégation et composition dans UML – Révision fondée sur la théorie Tout-Partie », *Technique et Science Informatiques*, 21(10), (2002) 1343-1370
4. Booch, G., and Rumbaugh, J. : *Unified Method, technical report, version 0.8*, Rational Software Corporation (1995)
5. Booch, G., Rumbaugh, J., and Jacobson, I. : *The Unified Modeling Language for Object-Oriented Development, technical report, version 0.9*, Rational Software Corporation (1996)
6. Booch, G., Rumbaugh, J., and Jacobson, I. : *Unified Modeling Language, technical report, version 1.0,* Rational Software Corporation (1997)
7. Cook, S., and Daniels, J. : *Designing Object Systems – Object-Oriented Modelling with Syntropy*, Prentice Hall (1994)
8. DSTC Pty Ltd : *UML 2.0 Infrastructure Revised Submission*, OMG document ad/2000-09-02 (2002)
9. DSTC Pty Ltd : *UML 2.0 Superstructure Revised Submission*, OMG document ad/2000-09-06 (2002)
10. Henderson-Sellers, B., and Barbier, F. : "Black and White Diamonds", *proceedings of «UML'99» – Beyond the Standard*, Lecture Notes in Computer Science #1723, Springer, (1999) 550-565
11. Leniewski, S. : *Sur les Fondements de la Mathématique – Fragments*, Hermès (1989)
12. Mellor, S. : "Whiter UML ?", *Keynote in Technology of Object-Oriented Languages and Systems Pacific'2000*, Sydney, Australia, November 20-23 (2000)
13. Object Management Group : *UML Summary, Semantics and Notation Guide*, version 1.1 (1997)
14. Object Management Group : *OMG Unified Modeling Language Specification*, version 1.3 (1999)
15. Object Management Group : *OMG Unified Modeling Language Specification*, version 1.5 (2003)
16. Object Management Group : *UML 2.0 Infrastructure Specification* (2003)
17. Object Management Group : *UML 2.0 Superstructure Specification* (2003)
18. Object Management Group : *UML 2.0 Superstructure Specification* (2004)
19. Object Management Group : *UML 2.0 OCL Specification* (2003)
20. Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., and Lorensen, W. : *Object-Oriented Modeling and Design*, Prentice Hall (1991)
21. Shlaer, S., and Mellor, S. : *Object-Oriented Systems Analysis – Modeling the World in Data*, Prentice Hall (1988)
22. Taivalsaari, A. : "On the Notion of Inheritance", *ACM Computing Surveys*, 28(3), (1996) 439-479
23. Tardieu, H., Rochfeld, A., and Colletti, R. : *La méthode MERISE – tome I – Principes et outils*, Nouvelle édition, Les Editions d'Organisation (1989)
24. Warmer, J., and Kleppe, A. : *The Object Constraint Language*, Addison-Wesley (1998)

2.8 WEBOGRAPHIE

Distributed Systems Technology Centre (DSTC) : www.dstc.edu.au
Precise UML group (pUML) : www.puml.org
Unified Modeling Language (UML) : www.uml.org
UML à IBM/Rational : www-306.ibm.com/software/rational/uml
UML Resources Center : umlcenter.visual-paradigm.com
The UML Bibliography : www.db.informatik.uni-bremen.de/umlbib

2.9 ANNEXE : PRÉSENTATION CONCISE D'OCL

La navigabilité dépend en OCL de modèles UML comme celui de la figure 2.79 :

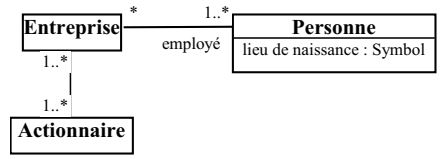


Figure 2.79 — Modèle UML sans contrainte OCL.

L'expression qui suit correspond à l'ensemble de tous les employés d'une entreprise donnée (*self*) :

```
context Entreprise inv:[1]
self.employé
```

L'expression qui suit correspond à l'ensemble de tous les actionnaires des entreprises où est employée une personne (*self* peut être omis en général). L'absence de rôles (« employé » pour « Personne » par exemple) à côté de « Entreprise » et de « Actionnaire » impose l'utilisation des termes « entreprise » et « actionnaire » :

```
context Personne inv:
entreprise.actionnaire
```

Les termes logiques de base sont : *not*, *true*, *false*, *and*, *or*, *implies* (*i.e.* $\Rightarrow$)

La quantification est : *forAll* ($\forall$), *exists* ($\exists$)

Les opérateurs ensemblistes (préfixés par) sont : *select* (sous-ensemble vérifiant une propriété), *isEmpty*, *notEmpty*, *size* (cardinalité), *includes* ($\in$), *includesAll* ($\subseteq$), *union* ($\cup$), *intersection* ($\cap$), *including* (surensemble obtenu par ajout d'éléments)…

Une utilisation des opérateurs ensemblistes est :

1. *context Entreprise inv:* s'interprète comme $\forall$ *self: Entreprise*.

```
context Actionnaire inv:
entrepriseforAll(e | e.employéselect(lieu de naissance = #Paris)notEmpty())
```

Cette contrainte hétéroclite précise que pour un actionnaire donné, la liste des employés nés à Paris (# représente une valeur possible du type OCL prédéfini *Symbol*) et travaillant dans chaque entreprise pour lequel il est actionnaire, n'est pas vide.

Les opérateurs dans la spécification des opérations sont :

- *oclIsNew* (dans les post-conditions uniquement : révèle la naissance d'un objet)
- *@pre* (dans les post-conditions uniquement : révèle la valeur d'un objet avant le début de l'opération)

Voici un exemple de précondition et de post-condition :

```
context Entreprise::nouveauEmployé(p : Personne) : Boolean[1]
pre: not employé Æ $$includes(p)
post: employé = employé@preincluding(p)
```

Les opérateurs utilisant les aspects typage sont :

- *oclIsTypeOf* (teste le type direct d'un objet)
- *oclIsKindOf* (teste si un objet se conforme à un type qu'il soit son type direct ou un ancêtre de son type direct au sens de l'héritage)

D'autres opérateurs intéressants sont :

- *oclIsUndefined* (depuis OCL 2 seulement, pour déterminer la dissociation d'un symbole/terme d'une expression OCL d'un objet : l'objet n'existe plus)
- *oclInState* (teste l'état d'un objet et permet donc de faire le lien avec les *State Machine Diagram*)

Enfin, l'accès au métamodèle UML et l'utilisation des navigations prédéfinies du métamodèle peuvent donner (expression incongrue mais néanmoins correcte) :

```
context Personne inv:
self.oclIsType(Personne) = true and Personne.oclIsType(Classifier) = true
    ➥and Personne.oclIsKindOf(ModelElement) = true
```

1. Forme générale des contraintes d'une opération d'un type (symbole « :: ») ainsi que préconditions (« *pre* ») et post-conditions (« *post* ») associées.

3

UML Behavior

3.1 INTRODUCTION

Nous traitons dans ce chapitre de tout ce qui relève de la modélisation du « comportement » (appelé *Behavior* dans UML 2.x ; attention néanmoins à ce terme « comportement » et à ce qu'il recouvre). Lorsque nous décrivons les opérations dans les boîtes incarnant les classes (voir le chapitre 2), ne représentons-nous pas en partie le comportement ? Séparer modèles de structure et modèles de comportement a l'avantage de mieux organiser les vues du système modélisé, mais le besoin de cohérence reste élevé. Par exemple, il est à notre sens fondamental de retrouver les opérations éventuellement décrites dans la partie basse des boîtes présentes dans les *Class Diagram*, dans les modèles de comportement : quels sont les contextes dans lesquels elles sont lancées ? Quelles sont leurs conséquences ? Etc.

Dans UML 2.x, la scission entre *Structure* et *Behavior* est claire et surtout saine au sens où les types de diagramme sont soit relatifs à la description de la structure, soit relatifs à la description du comportement du système modélisé. Les *Use Case Diagram* en particulier sont maintenant insérés dans *Behavior* alors qu'en UML 1.x, ils jouaient un rôle particulier. Néanmoins, *Behavior* reste une forêt amazonienne et il est donc difficile de trier et d'extraire parmi toutes les notations celles qui sont les plus pertinentes. Comme nous l'avons dit au début du chapitre 2, notre faveur va aux *State Machine Diagram*. Ce chapitre est une investigation sur ce thème et dépasse les approches naïves habituelles d'UML où les machines à états sont plus des modèles informatifs que de véritables supports pour dériver du code. En effet, en approche objet et composant il est vital de bien isoler chaque entité en la pourvoyant d'un comportement intelligible par autrui pour en faciliter sa réutilisation. Les autres diagrammes (*Sequence Diagram*, *Activity Diagram*…) jouent alors plus un rôle de synthèse pour savoir comment les architectures et les applications peuvent se formater via l'interaction d'objets et de composants souvent prédéfinis.

Dans ce chapitre, mais aussi dans les suivants traitant d'exemples concrets et à grande échelle, nous nous intéressons particulièrement à la cohérence entre les deux classes de modèles que sont *Structure* et *Behavior*. Aucun progrès en UML 2.x sur ce sujet n'est à noter. Comment corriger alors cette tendance persistante héritée d'UML 1.x ? L'idée est de référencer dans les modèles de comportement les propriétés des modèles de structure, et vice-versa. Par exemple, l'arrivée dans un nouvel état d'un objet peut être sujette à une contrainte OCL indiquant qu'une collection se voit ajouter un nouvel élément. Dans un *Class Diagram*, une instance d'association varie alors en termes de valeur et on peut vérifier par exemple que la borne haute de la cardinalité n'est pas dépassée. Nous privilégions cette méthode de modélisation fortement axée sur OCL, qui est à notre sens essentielle à une bonne modélisation objet.

3.2 COHÉRENCE ENTRE TYPES DE DIAGRAMME DYNAMIQUES EN UML 2.X

Il y a dans UML 2.x un grand trouble quant à la cohabitation (recouvrement des concepts utilisés voire une redondance certaine impliquant une quasi-compétition) entre tous les types de diagramme comportemental. Il nous semble que, dans UML 2.x, les *Activity Diagram* sont devenus les diagrammes dominants (au sein de *Behavior*) par la place qu'ils tiennent dans la documentation et la richesse de leur notation propre.

Dans les différentes « couches » de la modélisation objet (chapitre 1, modélisation objet dite « stratégique » en particulier), l'ingénierie des *business process* et des *workflow process* concerne plus volontiers la modélisation des systèmes que la modélisation des logiciels. Il apparaît que les *Activity Diagram* se situent dans cette problématique et arrivent donc en amont (de même que les *Use Case Diagram*) de diagrammes comme les *Statecharts* de Harel [4] ou encore les scénarios, plus dédiés à la formalisation de comportements individuels de composants logiciels (*State Machine Diagram*) et d'assemblages/interactions entre eux (*Sequence Diagram*), c'est-à-dire des architectures logicielles et des applications finies. Cette première observation conduit à discerner et sélectionner un type de diagramme plutôt qu'un autre en fonction de ses objectifs. Comme dit en introduction, notre intérêt porté à la conception concrète de logiciels rend les *Activity Diagram* peu séduisants. Un second constat est qu'UML perd de plus en plus sa nature objet car les *Activity Diagram* surtout, sont souvent[1] des modèles transverses à plusieurs types d'objet sans faire apparaître, comme dans les *Sequence Diagram* en particulier, les responsabilités de chaque type (services fournis) et les collaborations avec d'autres pour réaliser un calcul donné.

Ces remarques faites, UML 2.x fait partager aux différents types de diagramme des métatypes semblables. L'étude des principaux est donc indispensable.

1. Un usage précis et méthodique peut inverser cette tendance.

3.2.1 Niveau métamodèle

Il existe ainsi deux métatypes centraux qui sont *Activity* et *Action*. *Activity* appartient au package *BasicBehaviors* et fait l'objet de différentes spécialisations dont *Activity* du package *BasicActivities*. La figure 3.1comporte un extrait du métamodèle où l'héritage entre les deux notions d'*Activity* et leur liaison avec celle d'*Action* sont représentés : une activité est composée (losange noir) d'actions ordonnées (contrainte *{ordered}*).

Cette relation de composition donne donc la philosophie attachée à ces deux notions phares d'*Action* et d'*Activity*. Une action est une unité d'exécution atomique. Il en existe de différents types, selon en particulier leur usage dans un type de diagramme donné. Par exemple, le type *SendSignalAction* a une association avec le type *Signal* et est par ailleurs un sous-type d'*InvocationAction*, lui-même sous-type du type *Action* de la figure 3.1. *SendSignalAction* semble donc naturellement prédestiné aux *Sequence Diagram* mais peut être aussi associé aux *State Machine Diagram* pour modéliser l'envoi d'un signal d'un automate à un autre et donc symboliser leur interaction. Dans un autre esprit, *CallOperationAction* a lui une association avec le type *Operation* et hérite de *CallAction* (classe abstraite) qui hérite aussi d'*InvocationAction*. *CallOperationAction* est donc dédié à l'expression de la mise en œuvre des opérations (partie basse des boîtes) décrites dans les *Class Diagram*.

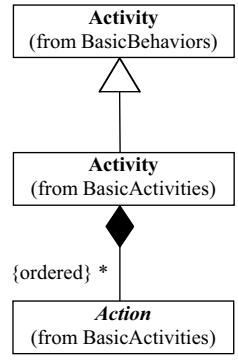


Figure 3.1 — Architecture noyau du métamodèle d'UML 2.x pour les concepts d'activité et d'action.

Le danger, ce dont il faut être conscient

Le lien entre les spécialisations avancées des métatypes *Action* et *Activity* avec d'autres métatypes intéressants comme *Operation*, *Event*, *Signal*, *Request* ou encore *Message* est irrationnel et difficilement compréhensible. On constate ici l'application du dicton « pourquoi faire simple quand on peut faire compliquer ». Au regard du socle intellectuel commun aux sciences informatiques, la distinction entre *Signal* et *Event* par exemple, même si elle peut se justifier dans d'autres domaines qu'UML, n'a pas lieu d'être. Il est de manière générale regrettable que le passage d'UML 1.x à UML 2.x ait été un alourdissement plutôt qu'un allégement.

3.2.2 Notation

L'instanciation des métatypes amène à produire des modèles utilisateur. Il n'y a pas toujours de guide clair quant aux règles rigoureuses régissant cette instanciation. Par exemple, *CallOperationAction* est instancié dans un *Activity Diagram* comme indiqué dans la figure 3.2 pour décrire la mise en œuvre de l'opération *decrement()* du type *Temperature* du chapitre précédent.

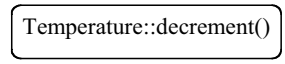


Figure 3.2 – Exemples d'instanciation du métatype *CallOperationAction*.

Dans un *State Machine Diagram*, celui de la classe *Temperature* précisément, une alternative de modélisation existe via l'opérateur *do/* qui signifie le lancement de l'opération dans l'état *A given state* (figure 3.3). On voit ici distinctement que c'est plutôt par goût et non par l'application d'une démarche méthodique que l'on modélise selon l'approche de la figure 3.2 ou celle de la figure 3.3.

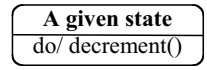


Figure 3.3 – Alternative de modélisation à la figure 3.2.

3.3 DIAGRAMMES D'ACTIVITÉ (ACTIVITY DIAGRAM)

La primauté des *Activity Diagram* en UML 2.x résulte de leur adéquation à la description de processus de gestion ou de calcul de taille plus moins imposante mais aussi d'organisations et de systèmes d'information.

À petite échelle par exemple, la figure 3.4 montre que l'activité *A given activity* est composée de l'action *decrement()* de *Temperature* précédant l'action *Another action*. *Step* est un paramètre d'entrée de l'activité également caractérisé comme une instance de *InputPin* dans le métamodèle. C'est normalement une donnée élémentaire ou une instance d'objet, ici en l'occurrence une valeur décimale qui est le pas de décrémentation de la température.

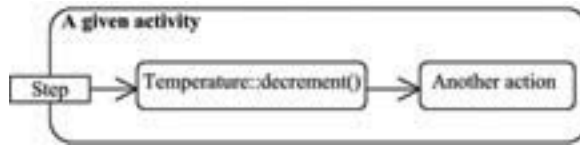


Figure 3.4 – Exemples d'instanciation du métatype *Activity* (un de ses sous-types).

Le formalisme des *Activity Diagram* s'étend à discrétion comme dans la figure 3.5 où l'on peut exprimer des récupérateurs d'erreurs, en l'occurrence des actions de sauvetage comme l'opération *increment()* de *Temperature* si l'on dépasse le zéro absolu (–273,15°C) en décrémentant de façon incorrecte.

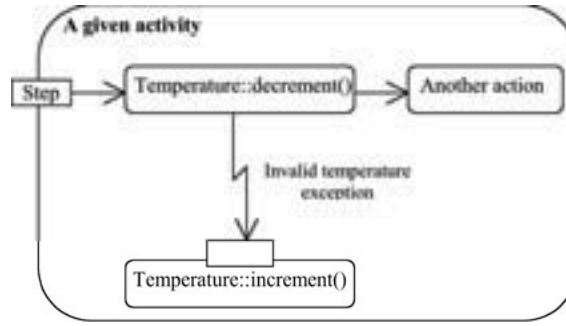


Figure 3.5 – Gestion des exceptions dans les *Activity Diagram*.

Pour les modèles de la figure 3.4 et de la figure 3.5, nous privilégions les *State Machine Diagram* au vu de l'étude comparée avec les *Activity Diagram* de la suite de ce chapitre, qui montre que le pouvoir d'interprétation des premiers est plus grand. Reste que pour des systèmes à grande échelle et dans des phases de dégrossissage, des modèles comme celui de la figure 3.6 donnent des vues intuitives de fonctionnement d'organisations complexes et de systèmes d'information d'entreprise.

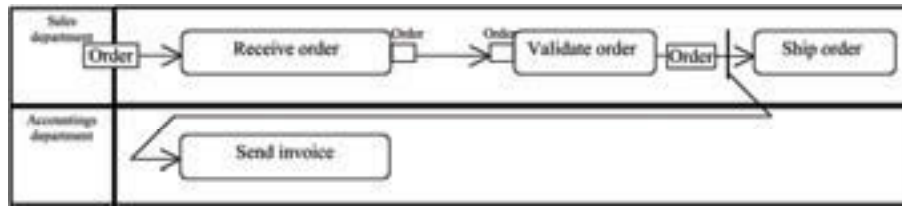


Figure 3.6 – Exemple de *business process* modélisé par *Activity Diagram*.

Dans la figure 3.6, le partitionnement permet de distinguer deux services d'une entreprise qui sont les ventes (*Sales*) et la comptabilité (*Accountings*) avec des actions propres et des flux de données (*Order* pour commande) et de contrôle. C'est la petite barre verticale noire à droite de la figure 3.6 : déclenchement de l'expédition (*Ship order*) et de la facturation (*Send invoice*), toutes deux liées à la fin de la validation de la commande (*Validate order*). Un des défauts des *Activity Diagram* est l'offre de constructions de modélisation redondantes. Par exemple en figure 3.6, les rectangles accolés aux actions *Receive order* et *Validate order* désignant une instance d'objet *Order* ont le même sens que l'unique rectangle entre les actions *Validate order* et *Ship order*. Quel est l'intérêt ?

3.4 MACHINE À ÉTATS (STATE MACHINE DIAGRAM)

La notion de machine à états est à l'informatique ce que le moteur à explosion est à l'automobile. L'inspiration initiale d'UML (qui fut aussi celle d'OMT [16]) est l'intégration d'une technique particulière de modélisation par machine à états : les *Statecharts* de Harel [4]. Cet emprunt semble un peu occulté dans la documentation UML 2.x mais les bases restent.

Les *Statecharts* de Harel ont évolué au cours du temps jusqu'à des versions « plus objet » dont Harel lui-même est l'auteur [6]. Notre présentation s'en tient à l'adaptation de cette technique dans son ensemble et telle qu'elle existe en UML 2.x. En ce sens, un principe fondateur des *Statecharts* en OMT et en UML est qu'un type dans un *Class Diagram* (*X* à gauche dans l'exemple de la figure 3.7) a ou n'a pas son comportement spécifié par un *statechart*. En effet, certains types d'objet ne sont que des données, de l'information, sans comportement tangible. Par exemple, *Order line* (ligne de commande) n'a probablement pas différents états possibles alors que la commande qu'elle compose en a probablement plusieurs, décrivant son cycle de gestion : soldée, non soldée…

UML 2.x La présentation dans la figure 3.7 est celle d'UML 2.x et non d'UML 1.x. En UML 2.x, il y a maintenant un formalisme approprié pour spécifier le type d'intérêt (*X* en figure 3.7) : en haut à gauche de la machine à états dans un compartiment dédié. Dans la figure 3.7, *State 1*, *State 2*, *State 3* et *State 4* sont des états nommés offrant une discrétisation du comportement du type X. À gauche de la figure 3.7, on suggère que la classe *X* dans un *Class Diagram* a ou non (*at most one*) *State Machine Diagram* associé.

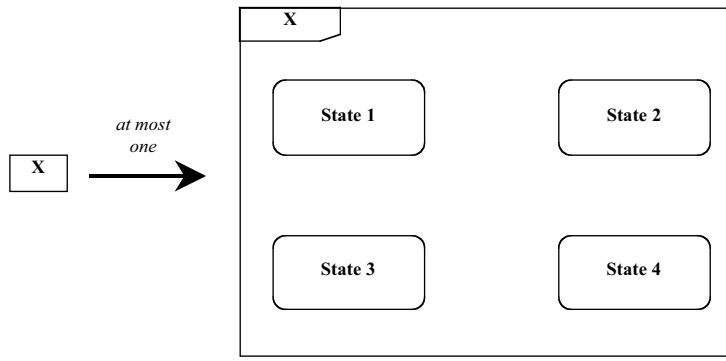


Figure 3.7 – Machine à états en UML 2.x.

L'essentiel, ce qu'il faut retenir

L'idée d'avoir au plus un automate par type d'objet n'est pas anecdotique. Ce principe énoncé et appliqué formellement à l'origine dans la méthode de Shlaer et Mellor [17] puis dans Syntropy [2], était mis à mal dans OMT : pas de règle bien

précise. La question est : pourrait-on avoir une machine à états qui décrive non pas un comportement individuel mais une collaboration entre plusieurs entités ? Si de tels automates sont conçus « au niveau système », *i.e.* non vus comme des types d'objet à part entière, les services qu'ils assurent (souvent uniquement de coordination) doivent alors être assignés à des composants logiciels à définir ou à trouver.

En pratique, certains types d'objet ne supportent aucune fonction de calcul, c'est-à-dire de transformation de données, au profit uniquement de tâches de gestion de la coopération : ces types d'objet sont souvent appelés contrôleurs et s'identifiaient en tant que tels dans UML 1.x sous l'influence d'Objectory/OOSE [8]. Dans UML 2.x, le stéréotype *«controller»* semble avoir disparu.

Ainsi, si la spécification des tâches de coopération est critique, des contrôleurs doivent être identifiés et classés d'après un regroupement logique des fonctions de gestion de coordination. Pour chaque type de contrôleur identifié, un *statechart* peut préciser alors la communication avec des types d'objet de calcul coopérant et échangeant via ces contrôleurs. Une autre façon d'expliquer les choses est de faire référence à Merise et à ses modèles conceptuels de traitement (MCT). Fondés sur les réseaux de Petri, ces MCT avaient justement le défaut de ne pas scinder et attribuer à des entités à forte cohésion le comportement global du système. Les chapitres suivants comportent des exemples de contrôleurs (*ATM* par exemple dans le chapitre 6).

3.4.1 Activité

Il faut être extrêmement prudent sur le terme « activité » en UML 2.x. La métaclasse *Activity* du package *BasicBehaviors* est celle utilisée dans les *State Machine Diagram* qui nous intéressent ici. Comme nous le verrons dans la suite de ce chapitre, cette métaclasse ainsi que celle d'*Action* sont définies et/ou spécialisées dans d'autres packages pour un usage plus spécifique aux *Activity Diagram*.

Concernant les *State Machine Diagram*, les « opérations » ou « fonctions » ou encore « services », appelons-les comme on veut, qui apparaissent dans les machines d'états, sont des instances de la métaclasse *Activity* du package *BasicBehaviors*, et rien d'autre. En pratique, les expressions *entry/*, *exit/* et *do/* d'UML 1.x sont toujours là et préfixent maintenant des activités. En UML 1.x, *do/* s'associait à une activité alors que *entry/* et *exit/* s'associaient à des « actions », concept différent des « activités », tout cela étant un legs d'OMT. Par la suite et dans la logique d'UML 2.x, activité et action sont des concepts assez interchangeables, à la réserve près qu'une activité est une opération décomposable et qu'une action est atomique.

Activité d'état

Si l'on prend un distributeur automatique bancaire, le cycle opératoire peut de manière simplifiée être vu comme la séquence : état opératoire (*Operating*), faire

éjecter la carte (*make card ejected*), faire distribuer l'argent (*make cash dispensed*) et faire imprimer le ticket (*make receipt printed*). La figure 3.8 illustre cela sans événement particulier attaché aux transitions.

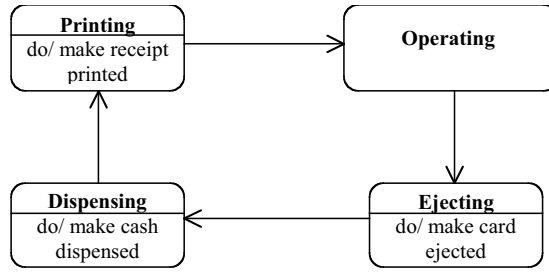


Figure 3.8 – Exemple d'activités d'état.

Le danger, ce dont il faut être conscient

Les questions soumises au lecteur et sur lesquelles nous revenons dans ce chapitre sont : comment transite-t-on dans l'automate ? La détection de la fin du déroulement d'une activité fait-elle passer à l'état suivant ? Automatiquement, *i.e.* sans intervention extérieure ? S'il y a des événements étiquetant les transitions, une occurrence d'événement signifie-t-elle l'interruption de l'activité ou l'attente de sa fin avant interprétation de l'événement ? S'il y a interruption et que nous revenions dans l'état peu après, l'activité sera-t-elle relancée à son point d'arrêt ou reprendra-t-elle au départ ?

Si toutes ces questions surviennent, c'est parce que lors du traitement des cas réels dans les chapitres qui suivent, la précision de la modélisation dépendra de la façon dont UML répond à ces questions, ou souvent n'y répond pas, d'où la nécessité de définir notre propre sémantique. Jeu dangereux à grande échelle, car dans une équipe de développement, l'interprétation des modèles doit être unique au risque d'erreurs dans l'application logicielle conçue. De manière générale, nous essayons ici de dépasser le simple cadre de la notation pour comprendre rigoureusement ce que sous-entendent les *State Machine Diagram*.

UML 2.x Par curiosité et par souci de comparaison, si l'on doit représenter le modèle de la figure 3.8 sous forme d'*Activity Diagram* de type UML 2.x, on aboutit au modèle de la figure 3.9. Il n'y a pas grande différence.

Les deux cercles nommés *An activity edge connector* sont des connexions entre activités. Bien que visuellement distincts (il aurait peut-être été préférable de n'en dessiner qu'un), ils symbolisent la même entité et précisent ici que l'activité *Make card ejected* suit séquentiellement celle intitulée *Make receipt printed*. On les utilise parce l'état *Operating* en figure 3.8 n'a pas de *do/* et donc n'a pas d'activité en propre. On ne peut donc pas *a priori* le transformer comme une activité si l'on traduit un *State Machine Diagram* en un *Activity Diagram*.

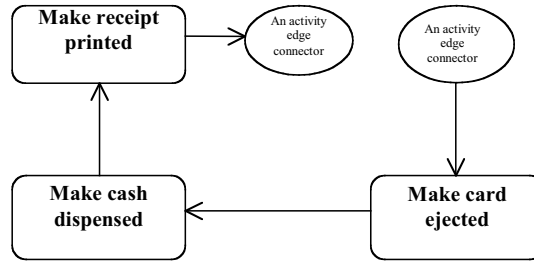


Figure 3.9 – *Activity Diagram* équivalent au modèle de la figure 3.8.

Activité déclenchée par événement

On s'intéresse maintenant à un être humain dont l'automate en figure 3.10 donne un événement « enterrement » (*burial*) qui déclenche l'activité « enterrer » (*inter*).

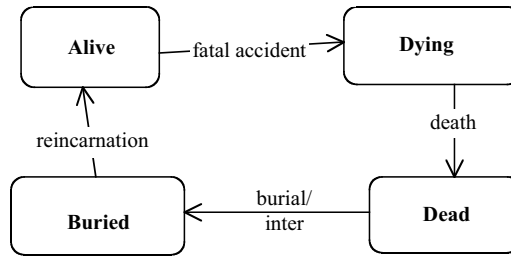


Figure 3.10 – Exemple d'activité déclenchée par événement (au sens d'UML 2.x).

Tout cela n'est pas très réjouissant certes mais la réincarnation ne se profile-t-elle pas ? Plus sérieusement, une occurrence d'événement par nature sans durée signifie-t-elle que la durée de l'activité *inter* est logiquement aussi sans durée ? Alors, les états ayant par définition des durées n'imposent-ils pas que les activités préfixées par *do/* aient elles aussi des durées ? Ce sont d'autres problèmes traités dans ce chapitre dans la section sur l'exécutabilité.

Activités d'entrée et de sortie d'état

La figure 3.11 met en opposition des activités lancées au moment où l'on rentre dans un état (mot-clef *entry*) avec des activités se déroulant *tout le temps* où l'on est dans un état (mot-clef *do*, figure 3.8). En fait, les activités d'entrée d'état ont vocation à simplifier les modèles. Dans la figure 3.11, un système d'ouverture de porte de garage fait qu'il y a deux façons d'entrer dans l'état « en phase de fermeture de la porte » (*Closing*). Si la porte est en phase d'ouverture (*Opening*) et l'on presse sur le bouton de la télécommande (*depress*), le mouvement de la porte s'inverse. Si la porte est déjà ouverte (*Open*), le même événement amène au mouvement de fermeture. Dans les deux cas, il faut commander le moteur avec un actionnement vers le bas (activité *motor down*). Au lieu d'associer l'activité *motor down* à chaque transition entrant dans *Closing*, cela par un / suffixant l'événement étiquetant la transition, on factorise

la réaction dans l'état. La règle générale est : quelle que soit la manière faisant entrer dans *Closing*, il faut lancer *motor down*. Ce cas de figure justifie l'intérêt des activités qualifiées *entry*.

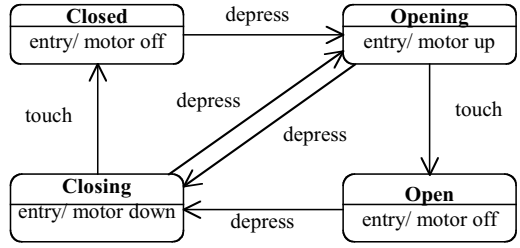


Figure 3.11 – Exemples d'activités préfixées par *entry/*.

Le même raisonnement s'applique pour les activités lancées à la sortie d'un état (mot-clef *exit*). Il faut aussi et finalement comprendre que les activités d'entrée et de sortie d'état ont le même « pouvoir » que les activités déclenchées par événements : l'instantanéité.

L'essentiel, ce qu'il faut retenir

Restons-en ici à une règle simple, les activités associées à des *do/* ont une durée et celles figurant derrière les événements sur les transitions (ou actions internes, section suivante), et celles associées à *entry/* et *exit/*, n'ont pas de durée. Cette approche est relative à l'échelle de temps de référence : que signifie avoir une durée ? Rumbaugh *et al.* développent une discussion sur le sujet dans [16]. D'un point de vue formel, considérer les activités associées aux états comme interruptibles et celles associées aux événements comme ininterruptibles est un critère de modélisation plus sûr.

Action interne associée à un événement (par opposition à activité associée à un état)

Les actions internes aux états n'étant pas préfixées par l'expression *do/* mais par le nom d'un événement puis un / (événement *event a* inscrit dans l'état *S* en figure 3.12), sont considérées instantanées. Ces actions servent à court-circuiter les clauses *entry/* et *exit/*.

Une simulation du comportement décrit en figure 3.12 donne donc :

- Si l'on est dans l'état *S* et qu'il y a apparition d'une occurrence de *event a* alors *x* est activée ;
- Si l'on est dans l'état *S* et qu'il y a apparition d'une occurrence de *event b* alors *y* (on sort d'abord de S) puis *z* (déclenchée car suffixant *event b*) et enfin *w* (on rerentre dans S) sont activées.

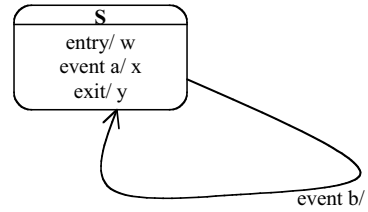


Figure 3.12 – Action interne.

3.4.2 Emboîtement

L'emboîtement est un fondement des *Statecharts* de Harel. Un sous-état ou état emboîté a son contour placé au sein d'un autre état. La relation entre l'état contenu et l'état contenant est conceptuellement apparentée à l'héritage. Par exemple, soit une boîte de vitesses (*Gearbox*) ayant trois états de base (figure 3.13) : point mort (*Neutral*), en marche avant (*Forward*) et en marche arrière (*Backward*), on peut dire qu'être en première vitesse (*First gear*) « est une sorte de » être en marche avant.

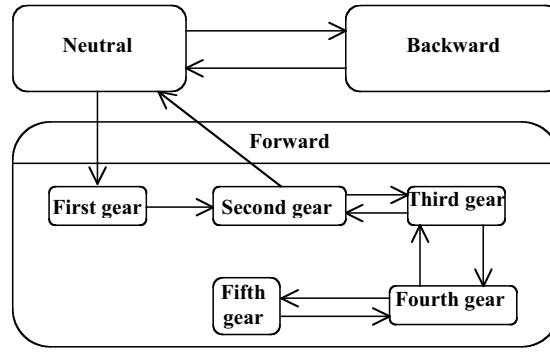


Figure 3.13 – *Statechart* avec emboîtement décrivant une boîte de vitesses.

Il en résulte que *Forward* est un état macroscopique auquel il est possible d'associer des comportements par défaut. Si l'on a besoin de détailler, on s'intéresse à ses sous-états pour, par exemple, préciser que l'arrêt d'une voiture consiste à passer de la deuxième au point mort (bien que certains conducteurs, en rétrogradant, passe la première avant le point mort, cela pour la bonne santé mécanique de la voiture !).

Dans la figure 3.13, la transition de *Second gear* à *Neutral* franchit le contour de *Forward* selon la notation originelle des *Statecharts*. Une transition de *Forward* à *Neutral* est ainsi dans l'absolu « correcte » mais imprécise, d'où la nécessité de s'intéresser à l'intérieur de *Forward* : *Second gear* ici.

3.4.3 États d'entrée et de sortie, pseudo-états

Les états d'entrée (point noir) et de sortie (point noir cerclé) déjà présents en UML 1.x demeurent, mais de nouvelles possibilités existent comme celles mises en œuvre dans la figure 3.14 (à droite). Parallèlement, la nouvelle notation d'UML 2.x autorisant le nommage de la machine à états elle-même (figure 3.7 ou droite de la figure 3.14) donne une schématisation plus claire. La notation adoptée dans le modèle à gauche de la figure 3.14 est standard pour les états d'entrée et de sortie. Au lieu d'avoir une transition directe de *Neutral* à *First gear*, la transition s'arrête sur le contour de *Forward*. Pour éviter l'indéterminisme, le point noir état d'entrée dit que toute entrée par défaut dans *Forward* fait aller à l'état *First Gear*. Idem pour la sortie de *Forward*, l'automate est sans équivoque car la seule transition touchant le contour de *Forward* et quittant cet état va sur *Neutral*. Tout cela est équivalent à ce qui est spécifié dans la figure 3.13.

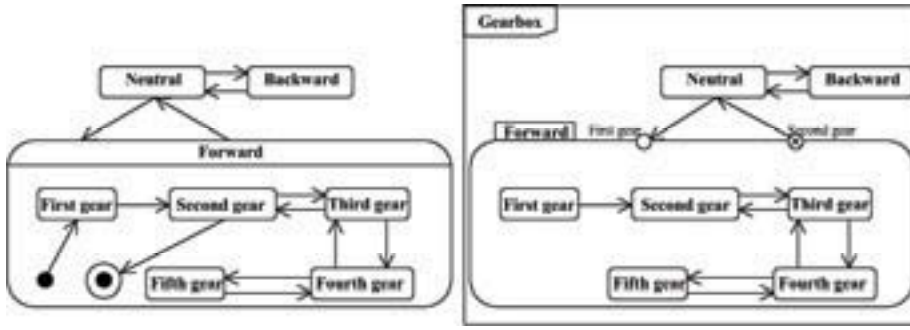


Figure 3.14 – Pseudo-états.

UML 2.x La nouveauté apportée par UML 2.x se situe dans le modèle à droite de la figure 3.14. Elle est faible et n'a pas besoin d'explication supplémentaire : quel est l'intérêt d'une telle notation ? On se le demande.

Pour information, notons que les états d'entrée point noir et de sortie point noir cerclé appartiennent aussi aux *Activity Diagram*. Nous cherchons en fait tout au long de ce chapitre à sensibiliser le lecteur sur l'appariement évident et reconnu entre *State Machine Diagram* et *Activity Diagram*. Pratiquement, c'est un problème de goût que d'utiliser l'un ou l'autre car ils se recouvrent de manière importante. Rappelons-nous l'extrait de texte de la version 1.3 d'UML cité dans le chapitre 2 qui dit que les *Activity Diagram* sont des sous-types des *State Machine Diagram*.

3.4.4 Invariants

La notion d'invariant bien qu'essentielle n'a pas fait l'objet d'une attention particulière en UML 1.x alors qu'elle le fut en Syntropy [2]. Comme nous l'avons signalé au chapitre 2, les invariants reviennent au premier plan en UML 2.x et il est utile de rappeler d'où vient cette influence : Syntropy.

Dans la figure 3.15, c'est exceptionnellement la notation de Syntropy et non celle d'UML qui est utilisée. Les classes sur la droite barrées en haut à gauche de leur boîte sont des « classes état »[1]. Autrement dit, elles constituent une autre vue des états et sont liées au type par des liens d'héritage. En clair, le type *Bottle* (bouteille) a un état de premier niveau « en cours » (*In progress*) ayant lui-même un sous-état plein (*Full*), etc.

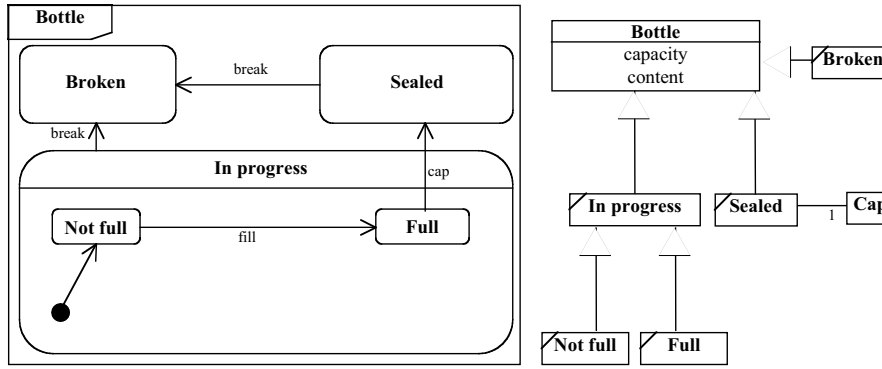


Figure 3.15 – Invariant et classe état en Syntropy.

L'intérêt ici est de pouvoir dire qu'une bouteille fermée (*Sealed*) a un bouchon (*Cap*) : association entre la classe état *Sealed* et le type *Cap* au lieu de dire qu'une bouteille (indépendamment de son état) a un lien avec un bouchon de cardinalité *0..1*. Le lecteur fera le lien avec la contrainte *{redefines}* du chapitre précédent apparue en UML 2.x, qui, dans le même esprit, a pour but de contextuellement redéfinir des cardinalités mais pour des sous-types n'étant pas des états.

L'idée est de montrer comment opérer une cohérence entre modèles statiques du chapitre 2 et modèles dynamiques de ce chapitre. Ne disposant pas en UML 1.x du formalisme utilisé à droite de la figure 3.15, l'usage d'OCL devient contingent à celui d'invariant pour préciser par exemple que l'état non plein (*Not full*) est le fait que le contenu (*content*) est strictement inférieur à la capacité (*capacity*). On obtient par exemple :

```
context Bottle inv Not full:
   content < capacity
context Bottle inv Full:
   content = capacity
```

Pour traiter le problème évoqué en début de cette section, comme il n'existe pas de *state class* en UML 1.x, on doit écrire :

```
context Bottle inv:
   --on suppose une association entre Bottle et Cap de cardinalité 0..1 vers Cap
   self.oclInState(Sealed) implies self.cap→size() = 1
```

1. Le terme anglais est *state class* dans une version 2 officieuse d'OMT jamais publiée.

UML 2.x Puisque la notion d'invariant prend une part plus importante en UML 2.x, il est intéressant de se rendre compte que, au contraire d'UML 1.x et en harmonie avec Syntropy, les invariants réintègrent le graphisme. En d'autres termes, les contraintes OCL se placent dans la machine d'états elle-même selon la présentation de la figure 3.16.

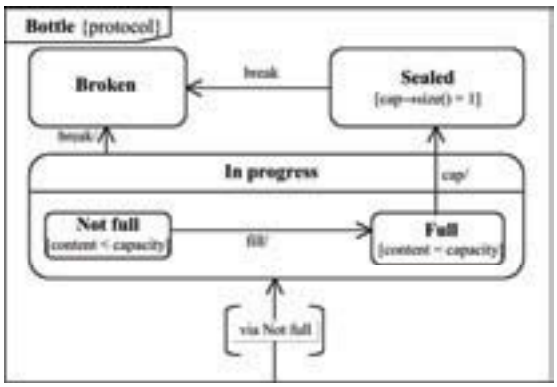


Figure 3.16 – Invariant en UML 2.x.

On en profite en figure 3.16 pour s'attacher à de nouveaux détails de notation comme *{protocol}* qui désigne des machines à états où les expressions clés *entry/*, *exit/* et *do/* sont par définition prohibées. En clair, les machines à états *{protocol}* sont déclaratives et non pas impératives au sens où il n'y a pas d'actions et d'activités mentionnées. Il y a plus volontiers des invariants. De plus, les transitions sont étiquetées selon le modèle suivant : [précondition] événement/ [post-condition] sans possibilité de déclarer des activités déclenchées par les événements.

Le lecteur attentif et connaissant bien UML 1.x aura remarqué la nécessité de suffixer le nom de l'événement par un / pour ne pas confondre garde (voir section suivante) et post-condition. Dans la figure 3.16, même s'il n'y a aucune post-condition, les événements sont immédiatement suivis d'un /. Quelle astuce ! On se demande bien où les concepteurs d'UML vont chercher tout cela.

Remarque : *via Not full* en figure 3.16 est encore une autre alternative aux pseudo-états.

3.4.5 Garde, précondition et post-condition

Les notions de garde, précondition et post-condition ont été « reformatées » dans UML 2.x, notamment les deux dernières qui maintenant bénéficient d'une notation dédiée (section précédente).

Garde

La figure 3.17 décrit un carrefour où alternent des situations où des voitures roulent nord-sud et vont tout droit (*N/S may go straight*) ou veulent tourner à gauche, ce qui

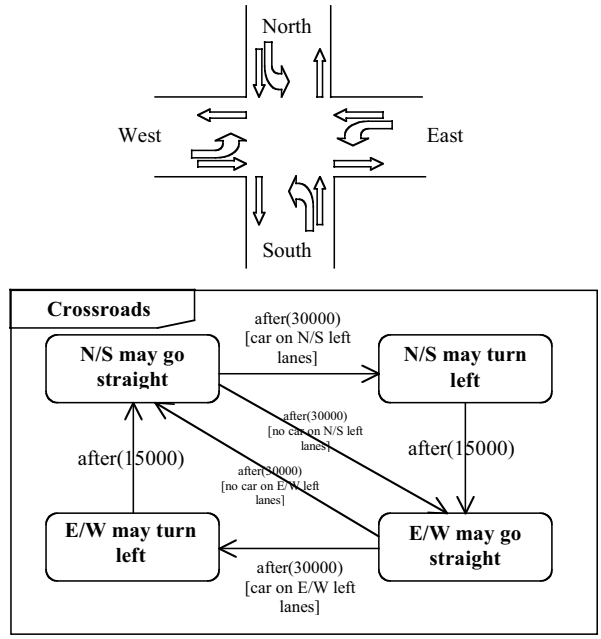


Figure 3.17 – Garde.

nécessite une configuration spécifique du carrefour routier : l'état *N/S may turn left*. La transition dans ce dernier état est conditionnée par la présence de candidats souhaitant tourner à gauche : *car on N/S left lanes*. C'est une garde dans l'automate qui pourrait être établie grâce à un capteur. De façon symétrique, une circulation est-ouest est régie par la même logique. L'automate résultant est en figure 3.17. L'événement *after(number)* qui repose sur le mot-clef *after* est plutôt propre à la notation UML 1.x. Il précise après combien de millisecondes (15000 ou 30000 dans la figure 3.17) une transition est déclenchée. On le remplace plus volontiers dans tout cet ouvrage par l'événement *time-out* avec une formalisation plus rigoureuse (voir chapitre 5).

Finalement et formellement, il faut écrire en OCL *car on N/S left lanes = not no car on N/S left lanes* et *car on E/W left lanes = not no car on E/W left lanes* pour que l'automate soit déterministe.

Précondition et post-condition

En imaginant des conséquences observables des transitions, on est alors amené à décrire des post-conditions éventuelles comme en figure 3.18.

La figure 3.18 introduit une post-condition : le feu passe à l'orange (*Amber*) pour les voitures circulant nord-sud si des voitures sur la même voie veulent tourner à gauche. Notons que la post-condition est bien incomplète si l'on veut décrire tous les états de tous les feux positionnés sur le carrefour : cela passerait aussi vraisemblablement par des invariants dans la machine à états de la classe *Crossroads* mais aussi et peut-être dans d'autres automates.

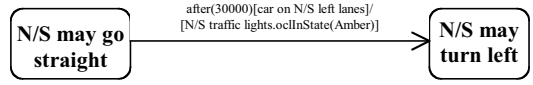


Figure 3.18 – Post-condition.

Le danger, ce dont il faut être conscient

Il y a là deux grands problèmes, un de forme, l'autre de fond. Pour le premier, la taille et la complexité des post-conditions (ou même des préconditions) vont rendre rédhibitoire leur insertion dans les automates eux-mêmes d'où le besoin, dans les autres chapitres d'études de cas notamment, de les exprimer hors des graphiques proprement dits (voir aussi la fin de ce chapitre où la même approche est suivie pour les *Sequence Diagram*). Pour le fond, le défaut de l'assertion *N/S traffic lights.oclInState(Amber)* est de faire référence à un état d'un autre objet. En l'occurrence *N/S traffic lights* est une expression de navigation vers un objet de type *Traffic lights* par exemple et *oclInState(Amber)* est l'opérateur OCL standard d'accès à l'état. La représentation de l'automate de *Traffic lights* engendrerait de fait la nécessité de mise en cohésion, ou mieux, de synchronisation des automates des deux types. Celui de *Traffic lights* posséderait au minimum et par définition l'état *Amber* plus d'autres propriétés « sémantiquement dépendantes » de *Crossroads*.
De manière plus générale, nous abordons un problème théorique bien connu dont une évocation est ce que l'on nomme « le produit synchronisé d'automates ». En UML, à un niveau beaucoup moins théorique, la question est de savoir par quelle manière (unique peut-être) les automates de différents types peuvent être rendus cohérents entre eux. À ce titre, une autre méthode de synchronisation est l'envoi d'événement, décrit soit dans les *State Machine Diagram*, soit dans les *Sequence Diagram* (voir ci-après).

Garde, aspects avancés

Il faut prendre garde (c'est le cas de le dire !) à la notation, dès lors que l'on profite de toute sa puissance. Sous un aspect anodin, la machine à états de la figure 3.19 est fortement contrainte par des dépendances logiques entre les gardes *g1*, *g2* et *g3*.

En fait, l'automate de la figure 3.19 ne fonctionne que si une occurrence de *e* apparaît. Une règle à nos yeux importante mais absolument pas émergente dans UML, est que la réactivité *e[g2]* masque *e[g1]*. Formellement, *e[g1]* donne une réaction « d'ordre général » que l'on soit dans *S1*, *S2* ou *S3* : on va dans (ou l'on boucle sur) *S1* si *g1* est vraie et *e* apparaît. *e[g2]* affine cette réaction générale pour *S1* sous la condition que *g2* soit vraie. Malheureusement, si *g1* implique *g2*, le modèle bien que restant « correct », devient « surfait ». Un automate plus concis et plus compréhensible aurait deux transitions étiquetées *e[g1]* allant de *S3* à *S1* et de *S2* à *S1*. Parallèlement, la transition actuelle démarrant du contour de *X* et arrivant sur *S1* disparaîtrait. Attention donc à la dépendance logique entre *g1*, *g2* et *g3*. Par exemple, il n'est ici pas nécessaire que $g2 \vee g3 = true$ mais que $g1 \vee g2 \vee g3 = true$ pour que l'automate soit déterministe : cette contrainte n'est pas évidente à établir vu la présentation du modèle de la figure 3.19.

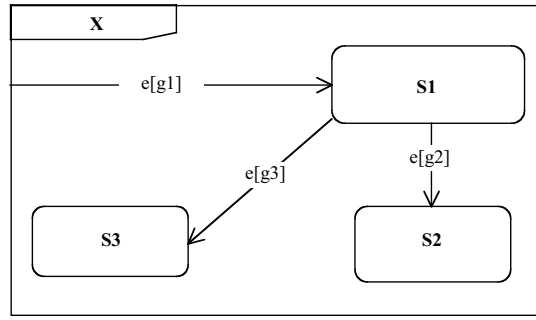


Figure 3.19 – Garde et emboîtement.

Le cas le plus extrême est qu'aucune des trois gardes ne soit vraie ($\neg g1 \wedge \neg g2 \wedge \neg g3$). Quel est l'effet produit par *e* ? Première hypothèse : aucun et *e* en tant que ressource consommable (jeton) disparaît du système : il n'aura en aucun cas d'effet retardé. Seconde hypothèse : il y a historisation des occurrences d'événements et des transitions peuvent être déclenchées *a posteriori* selon un processus d'interprétation (de type FIFO par exemple) des occurrences d'événements mémorisées.

3.4.6 Historique

UML 2.x dispose explicitement d'un mécanisme d'historisation calqué sur les *Statecharts* originaux de Harel. La documentation reste peu claire sur le sujet du fait d'absence d'exemple. La figure 3.20 est à notre sens un usage correct du mécanisme d'historisation (*H* cerclé). Imaginons que *S1* ait des sous-états, disons *S11* et *S12*, le *H* assure que toute nouvelle rentrée dans *S1* se fera dans « le » dernier sous-état actif, *i.e.* celui actif à la dernière sortie. Il serait alors inutile par exemple d'utiliser un pseudo-état d'entrée (point noir) dans *S1* lui-même, car on sait toujours que quand on rentre dans *S1*, on rentre dans *S11* ou dans *S12*. Lequel des deux ? Celui actif à la dernière sortie. Reste un problème, lequel des deux à la toute première entrée dans *S1* ? *S11* parce qu'il y a une transition du *H* cerclé vers *S11* justement. À notre connaissance, cette dernière transition n'est pas possible avec les *Statecharts* de Harel : le *H* cerclé n'est pas un sommet d'un graphe mais un indicateur de propriété d'historique pour le graphe lui-même.

Une notion d'historisation profonde (*H** cerclé) complète la précédente pour propager les règles au niveau le plus profond de l'automate (*deep history*) par opposition au *H* simple qui ne propage que d'un niveau (*shallow history*). En clair, en figure 3.20, l'effet mémoire ne fonctionne pas pour des sous-états éventuels de *S11* et *S12*.

3.4.7 Points de jonction, division et fusion de flux de contrôle

Dans UML 2.x, le concept de pseudo-état a pris une place plus importante qu'en UML 1.x, notamment via le métatype énuméré *PseudoStateKind* pouvant prendre les valeurs *entryPoint*, *exitPoint*, *initial*, *deepHistory*, *shallowHistory*, *join*, *fork*, *junction*, *terminate* et *choice*.

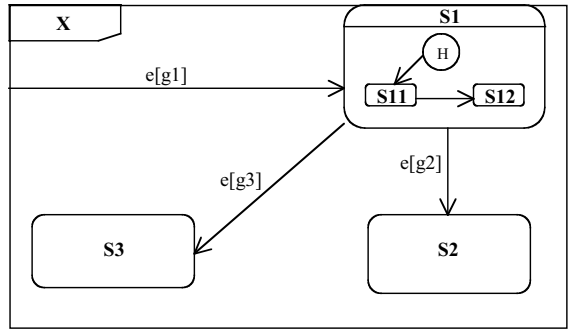


Figure 3.20 – Historisation.

Nous avons déjà vu ce que sont les états d'entrée, de sortie, initiaux, terminaux, d'historisation profonde et peu profonde (ou « à un niveau ») pour aborder maintenant les points de jonction (*junction*), de division (*fork*), de fusion (*join*) et de choix (*choice*). Nous allons exploiter ces concepts tant du point de vue des *State Machine Diagram* que des *Activity Diagram* où la ressemblance des modèles produits est étonnante, confirmant notre opinion de types de modèle plutôt concurrents dans UML.

Pour commencer, il est souhaitable de comprendre l'étude de cas abordée dans la figure 3.21 et la figure 3.22 : un conteneur mélangeur (*Mixing tank*) est associé à un mélangeur proprement dit (*Mixer*) qu'il commande en fonction de capteurs de remplissage (*Mixing tank full sensor*) et de vidage (*Mixing tank empty sensor*). Ce conteneur mélangeur possède aussi une valve d'injection (*Primary inlet valve*) des ingrédients à mélanger, le produit à fabriquer étant du shampooing.

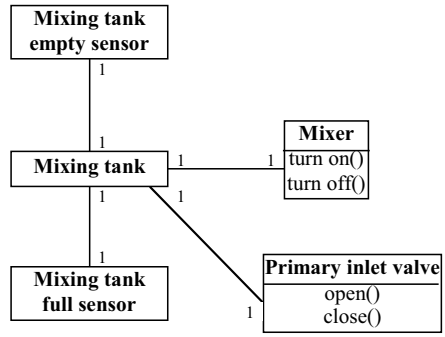


Figure 3.21 – *Class Diagram* d'un système de fabrication de shampooing d'après [17].

L'automate en haut de la figure 3.22 montre la division du flux de contrôle en deux à l'arrivée d'une occurrence de l'événement *filling stopped* (remplissage terminé). Les points (ou pseudo-états) de fusion et de division du flux de contrôle sont en effet les barres noires. La transition qu'étiquette *filling stopped* se sépare donc en deux puis on entre dans deux états parallèles (trait pointillé de séparation) et la sortie de chacun deux consiste alors à fusionner les deux transitions étiquetées par

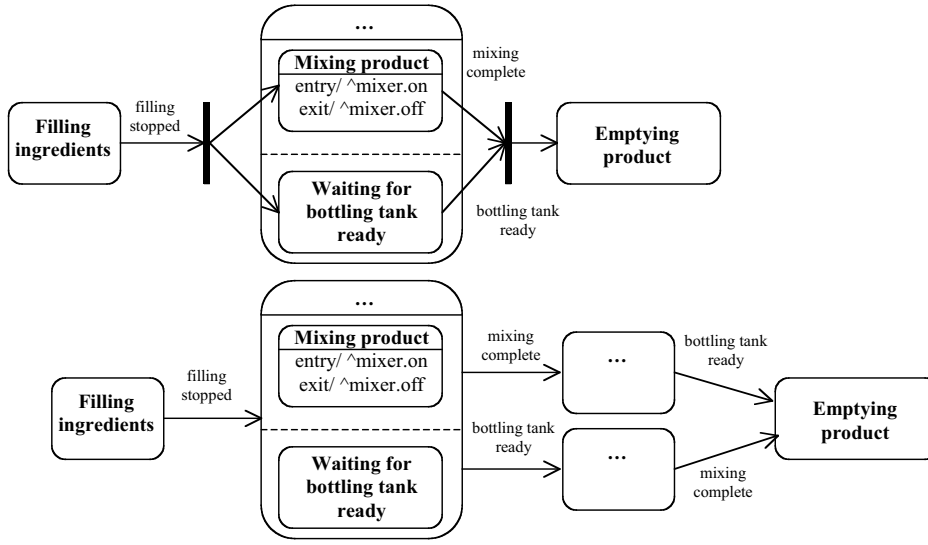


Figure 3.22 – Synchronisation par division et fusion (*State Machine Diagram*).

mixing complete (mélange terminé) et *bottling tank ready* (conteneur d'embouteillage prêt). Le shampooing est alors vidé du conteneur mélangeur : état *Emptying product*.

Les spécialistes de la modélisation objet connaissent bien ce cas paru dans [17, p. 59]. L'automate en bas de la figure 3.22 montre simplement le même système sans notation pour la division et pour la fusion. Deux états non nommés (le nom est remplacé par « … ») en bas à droite donnent la synchronisation : en clair, pour vider le shampooing, il faut que le processus de mélange soit fini et que le conteneur de réception (celui d'embouteillage) soit prêt à recueillir le produit. Ainsi, même si le modèle en haut de la figure 3.22 est plus compact via l'utilisation des barres noires mimant une transition d'un réseau de Petri, le modèle du bas est, pour ce qui est de la division du flux, équivalent et aussi simple. Concernant la fusion du flux en revanche, le modèle du bas est aussi équivalent mais plus « laborieux ». Il fait tout simplement apparaître la nécessité de prendre en compte l'ordre aléatoire d'arrivée des occurrences de *mixing complete* et de *bottling tank ready*.

Il est bien évident que les mécanismes de division et de fusion du flux de contrôle deviennent intéressants, en termes de notation spécialement, dès lors que de nombreuses transitions sont dégroupées, respectivement groupées. Le genre de modèle en bas de la figure 3.22 présente donc des limites au sens où il engendre une explosion combinatoire pour écrire la synchronisation. Sur le fond maintenant, Mellor dans [9] évoque la pertinence de modèles avec division/fusion pour les systèmes distribués où le temps est par essence relatif. L'indéterminisme quant à l'ordre d'arrivée de requêtes distantes pour un serveur donné par exemple, peut être géré sur la base de ces mécanismes de division et de fusion du flux de contrôle.

Les modèles de la figure 3.23 prennent le total contre-pied de ceux de la figure 3.22. Au lieu de réagir à des événements, par nature « phénomènes extérieurs », le

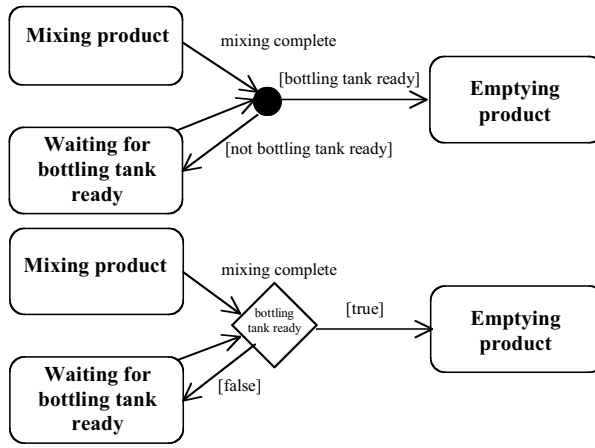


Figure 3.23 – Modèles connexes à ceux de la figure 3.22.

type d'objet *Mixing tank* dispose *a priori* d'informations sur son environnement pour mener à bien le même processus que celui représenté en figure 3.22. Il y a là un problème de fond extrêmement important sur lequel reviennent amplement les études de cas des chapitres à venir : quel concept UML (événement, garde…) associer à des propriétés, des phénomènes de la réalité ? Dans le cas du système de fabrication de shampooing, ne pourrait-on pas imaginer que l'événement *bottling tank ready* n'existe purement et simplement pas et qu'en contrepartie, le type d'objet *Mixing tank* peut toujours déterminer l'état du conteneur d'embouteillage via la variable locale booléenne *bottling tank ready* (figure 3.23, losange blanc) ?

Ainsi, les deux modèles de la figure 3.23 sont des *State Machine Diagram*. Le point noir est un point de jonction alors que le losange est un point de choix. Pas de différence notable dans leur usage : ils déroutent le flux en fonction de la valeur établie de *bottling tank ready*. La transition de l'état *Waiting for bottling tank ready* au point de jonction (modèle du haut), respectivement au point de choix (modèle du bas), n'est pas claire car elle n'est pas étiquetée par un événement. Il n'apparaît ainsi pas clairement quand et comment *bottling tank ready* est retestée.

Dans la figure 3.24, c'est un *Activity Diagram*. Il y a trois types de sommets : les activités (« rectangles à coins arrondis »), les nœuds objet comme *Mixing product report and anomalies* et finalement les signaux comme *Mixing complete*. Le sablier est un « sucre syntaxique » supplémentaire qui sous-entend ici que toutes les dix secondes, le statut du conteneur d'embouteillage est déterminé en vue de lancer l'activité *Empty product*. Les flèches correspondent autant à des flux de données qu'à des flux de contrôle, la distinction se faisant en fonction des types des nœuds. Par exemple, la flèche entre l'activité *Mix product* et le type d'objet *Mixing product report and anomalies* est un flux de données donnant naissance à un objet de ce dernier type. La notation prédéfinie *{weight=1}* mime une sorte de cardinalité, *i.e.* une seule instance est créée.

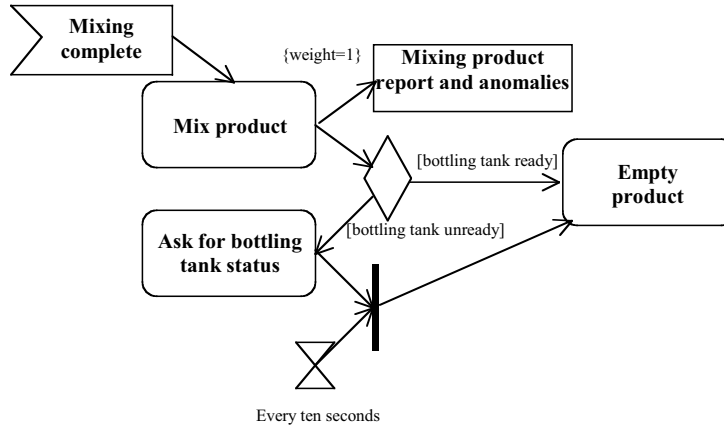


Figure 3.24 – *Activity Diagram* pour le système de fabrication de shampooing.

L'essentiel, ce qu'il faut retenir

La notation des *Activity Diagram* est plutôt intuitive et des régressions apparaissent : dans la bonne veille méthode SA-RT (*Structured Analysis – Real-Time*) comme dans OMT plus tard, les flux de données et de contrôle étaient graphiquement distingués : trait plein pour les premiers et trait pointillé pour les derniers. Ici, on revient à la confusion non seulement au sein même des *Activity Diagram* mais aussi dans leur faible démarcation des *State Machine Diagram*.

Remarque : la figure numérotée 374 de la documentation [14] qui traite de ce sujet est très probablement fausse. Il faut voir un losange au lieu d'un cercle.

3.4.8 Envoi d'événement

En UML 1.x, l'envoi d'événement ou de message[1] est symbolisé par le caractère ^. Le nom de l'objet adressé est suivi d'un point et du nom de l'événement envoyé. Dans la figure 3.22 par exemple, le démarrage et l'arrêt du moteur de mélange sont les mentions ^*mixer.on* en entrée, respectivement ^*mixer.off* en sortie, de l'état *Mixing product*.

UML 2.x Il semble que cette notation ait disparu en UML 2.x mais sans conséquence majeure, puisque le métatype *SendSignalAction*, comme nous l'avons vu en début de chapitre, incarne l'envoi d'événement. Nous conservons le caractère ^ tout au long de cet ouvrage car à la lecture de la documentation UML, seule une notation de ce métatype est prévue dans les *Activity Diagram* (figure 3.27) mais pas dans les *State Machine Diagram*.

1. En UML 2.x, *Message* et *Event* sont deux métatypes distincts. Nous y revenons mais nous pouvons d'ores et déjà dire qu'ils seront utilisés indifféremment, comme cela est fait par ailleurs dans OCL 2.

L'événement *filling stopped* reçu par la machine à états de *Mixing tank* (figure 3.22) est bien émis par « quelque chose ». Ce quelque chose est la machine à états de *Mixing tank full sensor* qui, une fois formalisée, donne les conditions et le contexte de l'émission (figure 3.25). On voit en particulier dans la figure 3.25 que l'ordre de fermeture de la valve d'injection des ingrédients du shampooing est donné par le capteur de gestion du remplissage. Il faut bien distinguer l'événement à traiter, ici *to be closed* (à fermer), de l'opération supportée par le type *Primary inlet valve* qui est *close()* (voir figure 3.21). En fait, on distingue l'envoi de l'ordre et son interprétation (franchissement de transition(s), changement(s) d'état le plus souvent) du traitement (calcul par déclenchement d'action) proprement dit que l'objet récepteur met en œuvre, en interne le plus souvent, pour assurer une conséquence à un phénomène par nature extérieur : l'envoi d'un événement, message ou encore requête d'un autre objet. La notation canonique correspond, dans une machine à états, au nom de l'événement reçu suffixé d'un / puis suivi du(des) nom(s) des services à réaliser et leurs paramètres éventuels.

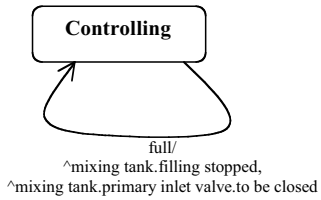


Figure 3.25 – Envoi d'événement en UML 1.x.

UML 1.x n'a jamais fait une distinction sémantique claire entre l'appel de fonction (par définition bloquant avec retour éventuel de valeur) et l'envoi de signal (asynchrone et sans retour associé) bien que la notation des *Sequence Diagram* s'attachait à le faire : trait plein pour les invocations de fonctions et trait pointillé pour les retours de telles invocations, les signaux étant matérialisés par une présentation spéciale de l'extrémité de la flèche, plutôt illisible à notre sens.

UML 2.x La dichotomie invocation d'opération/envoi de signal subsiste en UML 2.x et constitue même le noyau dur de la communication. Comment décrire maintenant la communication entre *Mixing tank* et *Mixing tank full sensor* ? Dans un *Sequence Diagram*, c'est immédiat (figure 3.26).

La notation de la figure 3.26 est utile mais bien insuffisante. Il faut pouvoir décrire la communication entre objets *dans* les *State Machine Diagram* (voir chapitres 5 et 6 notamment, signe ^) à cause du problème récurrent d'explosion combinatoire dans les *Sequence Diagram*. En UML 2.x, le métatype *SendSignalAction* incarne le moyen d'envoyer des signaux et est donc vu comme un type d'action spécifique. Il ne bénéficie pas de notation spéciale à l'exception de celle des *Activity Diagram* (figure 3.27). Pour conserver le signe ^ qui désigne l'envoi de message en UML 1.x, il faut maintenant mettre en œuvre le métatype *OclMessage* d'OCL 2 [15]. La portabilité reste donc globalement correcte.

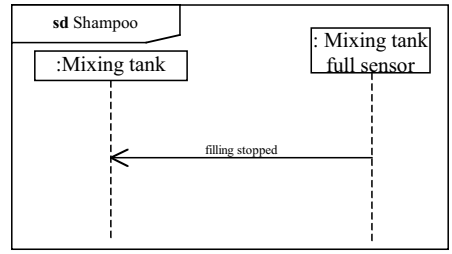


Figure 3.26 – Exemple de *Sequence Diagram* en UML 2.x.

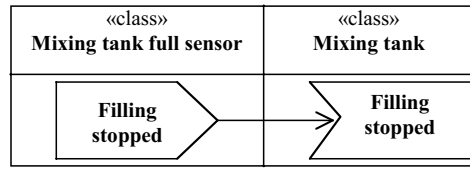


Figure 3.27 – Envoi et réception de signaux via les *Activity Diagram*.

En figure 3.27, on utilise le partitionnement, également appelé ligne de nage (*swimlane*) permettant comme dans la figure 3.26 de citer les objets concernés : on raisonne en termes d'instances dans la figure 3.26 (communication « un à un » entre une instance (préfixe :) de *Mixing tank* et une instance de *Mixing tank full sensor*). Dans la figure 3.27, ce sont des types qui sont décrits. On ne voit donc pas combien et quelles instances de *Mixing tank* et de *Mixing tank full sensor* sont censées collaborer : est-ce un envoi de message d'une instance à une autre (*unicast*), d'une instance à plusieurs (*multicast*) ou encore d'un ensemble d'instances à un autre ? Le formalisme reste flou.

Le danger, ce dont il faut être conscient

De façon plus générale, il apparaît de manière tangible l'extrême redondance dans les outils d'UML (*i.e.* les types de diagramme) d'où l'attention portée aux *State Machine Diagram, comme* annoncé en début de chapitre ; choix que les études de cas qui vont suivre valideront. Malheureusement, même dans l'hypothèse où ces *State Machine Diagram* sont privilégiés pour la partie *Behavior*, des problèmes persistent quant à leur sémantique formelle ou en terminologie moderne, leur exécutabilité. Malgré l'influence du groupe *Action Semantics* (anciennement xUML, voir la section « Webographie » en fin de chapitre) sur UML 2.x, il n'en résulte pas, à notre sens, des modèles réellement simulables, en particulier dans les ateliers de génie logiciel. Le dernier ouvrage de Mellor [10] est à cet égard décevant car il correspond plus à une adaptation des idées (de grande qualité certes) énoncées dans [17] qu'à une contribution significative.

Si l'on en revient au mécanisme primitif d'envoi et de réception d'événements, il convient de bien percevoir deux problèmes cruciaux. D'une part, le nom de l'objet adressé doit être résolu par calcul d'une instance précise d'un type d'un *Class Diagram* ou encore d'un ensemble d'instances. Dans la figure 3.25, l'expression *mixing tank* désigne par définition, l'instance unique du type *Mixing tank* liée à l'instance de *Mixing tank full sensor* qui émet l'occurrence de l'événement. Les navigations offertes par le *Class Diagram* (figure 3.21) jouent donc un rôle fondamental dans la détermination des objets ou ensembles d'objets concernés par la réception, l'interprétation et le traitement des événements. La navigation utilise aussi la liaison de *Mixing tank* à *Primary inlet valve*. On aurait pu imaginer deux communications au lieu d'une. Il aurait alors fallu décrire l'automate de *Mixing tank* pour d'abord le voir réceptionner *to be closed* et *filling stopped*, puis ensuite les propager à *Primary inlet valve*. Dans [2], la notion de filtre est introduite. Le filtre donne le mode de calcul de l'objet ou des objets récepteur(s). Soit donc l'automate d'un type d'objet R. Le premier paramètre d'un événement *e* identifie toujours l'objet récepteur avec la signature *e(r : R [r = self])*. Le filtre *[r = self]* est le filtre par défaut et peut être omis. Il en ressort que dans des applications un tant soit peu complexes, il faut disposer d'un moyen de fixer les objets récepteurs, ceux-ci étant souvent calculés dynamiquement : ils dépendent par exemple d'une valeur d'attribut d'un événement préalablement arrivé. À la fin de ce chapitre par exemple, on doit envoyer un signal à des « dispositifs » en panne, l'ensemble qu'ils constituent étant variable et dépendant du scénario (*Sequence Diagram*) où ils entrent en jeu. C'est assez difficile en UML 1.x comme en 2.x : les études de cas qui suivent se proposent de donner une solution.

L'autre grand problème est que toute occurrence d'événement porte comme attribut *implicite* l'identité de l'objet émetteur (mode d'exécution préconisé dans [17]). En conformité avec l'OMA (*Object Management Architecture*) [11] de l'OMG, l'hypothèse faite est que chaque objet a une identité unique. Le fait que l'identité de l'émetteur est connue du receveur autorise des réponses automatiques aux messages. Dans le cas inverse, le traitement d'une occurrence d'événement nécessitant de retourner quelque chose conduit le receveur à devenir l'émetteur d'un nouvel événement, et l'émetteur à devenir receveur pour récupérer la réponse à son stimulus. UML 1.x prévoit dans les *Sequence Diagram* et les *Collaboration Diagram* [12-13] une flèche en pointillé désignant un retour d'appel de procédure. Cette notation est proche de l'implémentation et nous la déconseillons fortement (voir aussi dans la fin de chapitre son pendant : l'appel de procédure) pour laisser la spécification flexible, c'est-à-dire ouverte à plusieurs possibilités d'implémentation.

Dans l'exemple de la figure 3.21, il y a dans le *Class Diagram* une navigation possible de *Mixing tank* à *Mixing tank full sensor* si une réponse s'avérait nécessaire. De plus, il n'y a qu'une seule association entre ces deux types avec à chaque extrémité des cardinalités *1..1*. Une solution est de signer les événements en incluant *explicitement* le type de l'objet émetteur comme premier attribut. On aurait ainsi dans l'exemple : *^mixing tank.primary inlet valve.to be closed(self)* et *^mixing tank.primary inlet valve.filing stopped(self)*. Malgré sa lourdeur, cette approche a l'avantage d'être plus didactique car dans les automates des receveurs, les émetteurs sont tangibles et

peuvent être manipulés tout à fait explicitement, pour répondre aux stimuli si besoin.

3.4.9 Parallélisme

Nous avons effleuré jusqu'à présent la notion de parallélisme d'états, plus volontiers appelée orthogonalité, voire concurrence. Dans les *Statecharts* de Harel, c'est un trait pointillé joignant le plus souvent deux points du contour d'un état ou deux points de la machine globale elle-même, c'est-à-dire le rectangle le plus englobant décrivant le comportement le plus macroscopique du type lui-même. Ainsi, une ligne pointillée rejoignant à chacune de ses extrémités un contour d'état, représente deux sous-états parallèles dudit état. Nous allons abondamment utiliser ce mécanisme dans toutes les études de cas de cet ouvrage.

En figure 3.28, l'état *S0* est tout d'abord présenté de manière macroscopique à droite en haut. Il y a dans l'ovale un petit symbole dédié (à l'aide de notation UML 2.x exclusivement) pour montrer que c'est un état. À gauche, la version expansée est présentée, avec des lignes pointillées séparant *S0* en trois sous-états parallèles : *S1*, *S2* et *S3*.

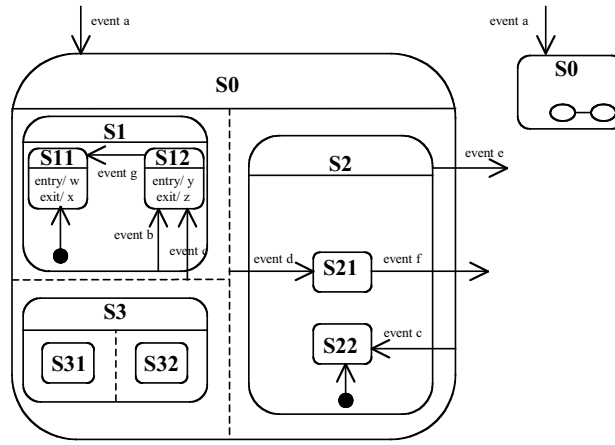


Figure 3.28 – Exemple d'états concurrents : *S1, S2* et *S3*.

Rappelons que l'emboîtement et la concurrence sont des éléments fondateurs des *Statecharts* de Harel. Même si de prime abord, le formalisme de la figure 3.28 est indigeste, une pratique volontariste (et longue) donne une autre opinion et montre la puissance de ce formalisme. Il est intéressant de vérifier « à la main » la cohérence du modèle en figure 3.28. L'événement *event a* est l'événement qui fait entrer dans *S0*. Après cela, si l'on analyse tous les scénarios d'apparition d'événement alors que l'on se trouve dans l'état *S11*, on obtient :

- S11 ∧ event b → x activée ∧ S12 ∧ y activée

- S11 ∧ event c → x activée ∧ S12 ∧ y activée
- S11 ∧ event d → (pas d'activation) S11 ∧ S21
- S11 ∧ event e → x activée ∧ ⊙ (*i.e.* on sort de *S0*)
- S11 ∧ event f ∧ S21 → x activée ∧ ⊙ (*i.e.* on sort de *S0*)
- S11 ∧ event f ∧ S22 → (pas d'activation) S11
- S11 ∧ event g → (pas d'activation) S11

Si l'on mène la même démarche pour *S12*, on s'aperçoit que l'automate est toujours déterministe et vivant :

- S12 ∧ event b → z;y activées ∧ S12
- S12 ∧ event c → z;y activées ∧ S12
- S12 ∧ event d → (pas d'activation) S12 ∧ S21
- S12 ∧ event e → z activée ∧ ⊙
- S12 ∧ event f ∧ S21 → z activée ∧ ⊙
- S12 ∧ event f ∧ S22 → (pas d'activation) S12
- S12 ∧ event g → z;w activées ∧ S11

En fait, il intéressant de se rendre compte que si l'on ne dispose pas de l'emboîtement, on produit des automates encore plus difficiles à interpréter. Dans la figure 3.29, on imagine une telle situation pour une partie réduite de l'automate en figure 3.28. En l'occurrence, toutes les factorisations (réactions aux événements sur les états les plus englobants) doivent être « descendues » sur les états plus intérieurs d'où une prolifération de transitions. De plus, des gardes doivent faire référence aux autres états comme *[**in** S21]* : on ne peut sortir de *S11* ou de *S12* à l'apparition d'*event f* que si l'on est conjointement dans *S21*. C'est ce que dit l'automate de la figure 3.28 mais de façon plus intelligible à notre goût.

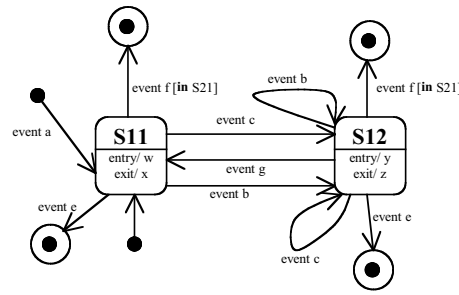


Figure 3.29 – Machine à états « à plat » imitant le comportement du sous-état *S1* de la figure 3.28.

Ainsi, rien que pour *S11* et *S12*, on voit la complication de l'automate dans la figure 3.29.

3.4.10 Exemple de synthèse

L'exemple présenté ici est un automate à usage général, en l'occurrence une liste d'éléments de type *T* partagée et donc utilisée de manière concurrente par des clients qui ajoutent (*push*) et suppriment (*pop*) des instances de *T*. Cette concurrence impose des mécanismes de verrouillage (*to be locked*) et de déverrouillage (*to be unlocked*). Le type dont on spécifie le comportement est un type générique *Locked list<T>* (notation UML 2.x idoine, voir chapitre précédent) scindé en deux grands états parallèles : le processus d'ajout/suppression d'éléments (sous-états *Empty*, *Not empty*, lui-même ayant les sous-états *Only one* et *More than one*, bas de la figure 3.31). Le processus de verrouillage/déverrouillage est caractérisé par les états *Unlocked* et *Locked*.

Les clients s'enregistrent et se retirent à l'aide des signaux *register* et *unregister*. Ils donnent leur identité (paramètre *client*) et le contexte (paramètre *context*). Ce dernier attribut des événements *register* et *unregister* est utilisé comme *qualifier* dans la figure 3.30. La navigation *clients* donne tous les clients enregistrés et la navigation via le *qualifier*, c'est-à-dire *client[context]*, donne ou non (cardinalité *0..1*), un client spécifique parfaitement caractérisé par la variable *context*. En clair, l'issue de l'enregistrement est d'ajouter le client à un dictionnaire dont *context* est la clé d'entrée. Pour cela, la post-condition de l'événement *register(client : Locked list client,context : Symbol)* (voir plus bas) est *client[context] = client*. Une conséquence intéressante est qu'un client peut s'enregistrer plusieurs fois ; il lui suffit pour cela d'utiliser des contextes différents. Au-delà, l'enregistrement permet de bénéficier des notifications *lock acquired* (« le » client demandeur de la ressource peut maintenant ajouter ou supprimer des éléments) et *lock released* (la liste vient de se libérer, « les » clients enregistrés, *i.e. clients*, peuvent en demander l'accès exclusif en réémettant l'événement *to be locked*) (voir le haut de la figure 3.31).

Dans la figure 3.30, la classe abstraite *Locked list client* (en italique) est introduite pour traiter de tout type d'objet client de *Locked list<T>*. Par ailleurs, ce dernier type d'objet hérite de la classe prédéfinie *Sequence(T)* d'OCL. On utilise pour cela le package lui aussi prédéfini *UML_OCL* pour l'atteindre.

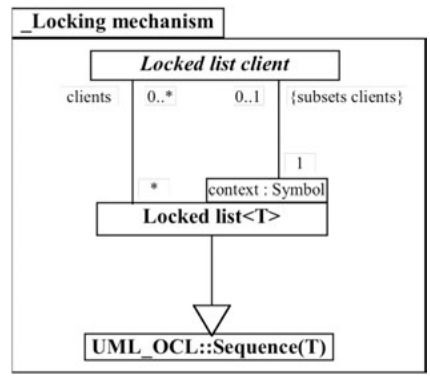


Figure 3.30 – Type générique *Locked list<T>* et son environnement.

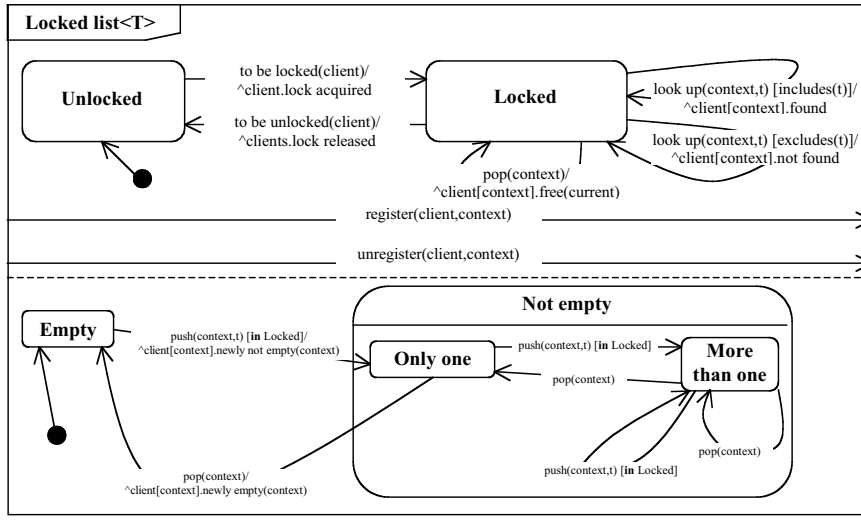


Figure 3.31 – Comportement de structure de données liste avec système de verrouillage/déverrouillage.

L'aspect le plus intéressant dans la figure 3.31 est la mise en cohésion des deux processus. En fait, le processus d'ajout/suppression via la garde *[**in** Locked]* détermine si la liste en tant que ressource partagée est bien bloquée (*Locked*). À titre de comparaison, un mécanisme de recherche d'élément (*lookup*) est aussi introduit. Soumis aux mêmes conditions (besoin de ressource verrouillée), il est spécifié sur la base d'une autotransition sur l'état *Locked* lui-même. C'est une alternative à *[**in** Locked]*.

L'essentiel, ce qu'il faut retenir

De manière plus générale, la nécessité de connaître d'autres états est récurrente dans les modèles. Ici, ce sont d'autres états du même objet mais cela peut être d'autres états d'autres objets (étude de cas du chapitre 5 par exemple). Faut-il alors banaliser la notation *in* (écrite en gras pour bien faire apparaître qu'elle appartient au formalisme originel des *Statecharts* de Harel) ? Ou alors, doit-on systématiser la représentation de l'interrogation d'état par un envoi d'événement « dans quel état es-tu ? » et d'une réponse « je suis dans l'état *E* » ? Au niveau conceptuel (modèles), cela n'induit pas d'inconvénient, à l'exception, sur la forme, de modèles plus « touffus » et donc plus difficiles à lire. En implémentation, la discussion est ouverte car il semble évident que des liens privilégiés entre composants logiciels doivent être tissés pour « laisser voir » les états, et donc d'une certaine manière casser, dans l'esprit, l'encapsulation. L'étude de cas du chapitre 5 se propose de traiter le problème de manière pragmatique.

Ainsi, la spécification de la figure 3.31 devient vite compliquée et sa vérification/validation n'est possible que via sa réutilisation dans un cas d'étude (voir chapitre 6) puisqu'UML ne dispose pas de système de preuve. Nous laissons au lecteur le soin

d'adapter/corriger cette spécification avec quelques conseils néanmoins. D'une part, la suppression d'élément ne permet pas de récupérer dans le contexte appelant l'élément supprimé. Ce n'est pas une erreur mais cela peut s'avérer utile, bien que non utilisé dans le chapitre 6. Une variable locale à l'automate appelée *current* est ainsi introduite dans les contraintes OCL ci-dessous (post-condition de *pop*) et correspond au dernier supprimé. Il manque le moyen de la lire/l'acquérir. D'autre part, la spécification ne contrôle pas au moment des *push* et *pop* si le client qui effectue les ajouts et/ou les suppressions, est celui qui a acquis le verrou. On considère que les clients sont disciplinés, c'est-à-dire qu'après s'être informés de l'acquisition du verrou (^*client.lock acquired*), ils envoient les signaux *push* et *pop* mais jamais à d'autres moments. Il est donc discutable du point de vue de la robustesse de laisser l'automate « si fragile ».

Nous terminons en donnant les contraintes OCL nécessaires à la formalisation précise du type *Locked list<T>*. Elles se divisent en trois grandes catégories.

Ajout/suppression :

```
context Locked list<T>::push(context : Symbol,t : T)
   pre: client[context]→notEmpty() --le client doit être préalablement enregistré
   post: self = self@pre→append(T) -- ajout de l'élément en fin de liste
context Locked list<T>::pop(context : Symbol)
   pre: client[context]→notEmpty() --le client doit être préalablement enregistré
   post: current = self@pre→first()
   post: self = self@pre→reject(t : T | t = self@pre→first()) -- suppression
                                          -- de l'élément en début de liste
```

Verrouillage/déverrouillage :

```
context Locked list<T>::to be locked(client : Locked list client)
context Locked list<T>::to be unlocked(client : Locked list client)
  pre: clients→includes(client) -- le client doit être préalablement enregistré
  pre: clients→includes(client) -- le client doit être préalablement enregistré
```

Enregistrement/désenregistrement :

```
context Locked list<T>::register(client : Locked list client,context : Symbol)
   post: client[context]→isEmpty() implies client[context] = client
context Locked list<T>::unregister(client : Locked list client,context : Symbol)
   post: client[context]→isEmpty()
```

3.4.11 Spécialisation de comportement

Nous plongeons ici dans les méandres théoriques de la modélisation objet, en l'occurrence les relations entre automates de types d'objet liés par l'héritage. De manière logique, considérant le *statechart* d'un type *X* et *Y* hérite de *X*, directement en l'occurrence, l'automate de *Y* n'est pas quelconque et peut être apparié à celui de *X*. De nombreux travaux scientifiques abordent ce sujet mais leur présentation est hors du cadre de cet ouvrage ; UML 2.x n'apporte toujours rien sur le sujet, malheureusement.

Dans le chapitre précédent, nous avions un compte bancaire (*Account*) qui était vu comme une classe abstraite dont les concrétisations sont les compte chèques (*Checking account*) et les comptes épargne (*Savings account*).

La figure 3.32 montre la singularité du comportement de chaque type de compte via en particulier l'ajout de la réaction à l'événement *pay back* (remboursement) pour un compte chèques. La fermeture d'un compte épargne s'opère par le versement du reliquat sur un compte chèques. L'événement *pay back* est donc émis par un compte épargne lors de sa fermeture (*end*) donnant aussi lieu à la fermeture proprement dite (opération *close* faisant partie de la liste des opérations de *Account*, voir chapitre 2). Au niveau de la réception, soit le compte chèques existe déjà (transition de *Open* à *Open*, en bas, à gauche de la figure 3.32), soit il est purement et simplement créé (transition du point d'entrée à l'état *Open* aussi en bas, à gauche de la figure 3.32).

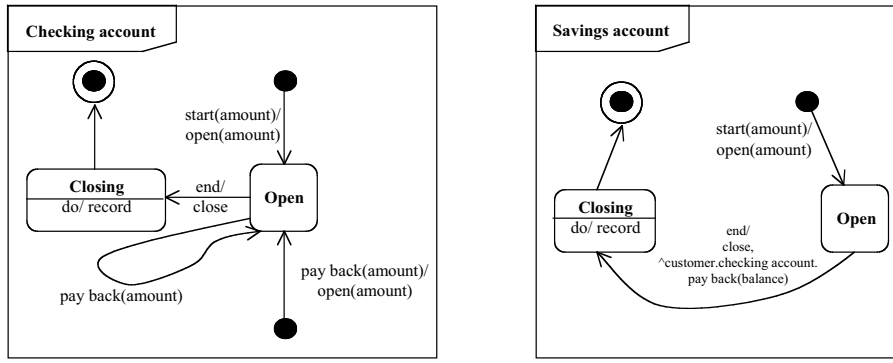


Figure 3.32 – Spécialisation du comportement à l'aide de *State Machine Diagram*.

En bilan, la réaction à l'événement *end* est particularisée pour les deux types alors que celle à *start* est commune, ce qui est cohérent avec le fait qu'ils héritent tous deux d'*Account* et donc qu'ils partagent des similitudes. Au-delà, une réflexion sur ce qu'est le polymorphisme (substituabilité) dans les *Statecharts* de Harel est complexe mais nécessaire dans l'esprit de la réutilisation. En effet, toute capitalisation des automates d'un type donné doit avoir pour bénéfice la représentation aisée, rapide et sûre du comportement des types qui le spécialisent.

3.4.12 Exécutabilité

La fabrication de « gros » automates et leur codage méthodique dans un langage de programmation mène vite vers une réflexion sur des règles de modélisation précises et rigoureuses. Prenons un exemple simple : si un état *E1* est doté d'un opérateur *do/* avec une activité *f1* et qu'il a un sous-état *E2* (de niveau immédiatement inférieur ou plus) lui aussi muni d'un opérateur *do/* avec une activité *f2*. Comment opèrent *f1* et *f2* ? En parallèle ? En séquence ? Si oui, laquelle des deux en premier, celle de

l'état le plus englobant ou celle de l'état le plus intérieur ? Il est possible de longuement chercher des réponses à ces questions dans la documentation UML…

Ainsi, si la spécification elle-même est ambiguë, comment alors ne pas penser que l'implémentation sera variable d'un programmeur à l'autre ? C'est intolérable. Le seul moyen de régler cela est de fixer soi-même ses propres règles. D'une part, les modèles deviennent exécutables, c'est-à-dire interprétables à l'aide d'outils logiciels par exemple. L'implémentation elle-même devient unique et mieux, une bibliothèque peut être développée en amont afin de systématiser le codage des *State Machine Diagram* sur la base de composants prédéfinis supportant la sémantique d'exécutabilité retenue. Cet ouvrage s'appuie à ce titre sur la bibliothèque MDE *PauWare.Statecharts* qui remplit ces conditions (voir le chapitre 5 en particulier).

Interprétation de statechart

L'idée toujours la même est de pouvoir interpréter n'importe quel automate. Par exemple, en figure 3.33, la mention *do/a* barrée (ce n'est pas de la notation UML) représente le fait que nous préconisons de déclarer des activités seulement pour les états terminaux (*i.e.* ceux n'ayant pas de sous-états) car sinon il y aurait ambiguïté concernant l'ordre d'enchaînement de l'activité *a* associée à l'état *V* et l'activité *b* associée à l'état *Y* sous-état de *V*. Par ailleurs, l'automate est mal formé car l'apparition d'une occurrence d'événement *e2* est associée contradictoirement à la sortie pure et simple de *V* (à gauche de la figure 3.33) et le lancement d'une activité interne *f* qui fait rester dans *W* sous-état lui aussi de *V*.

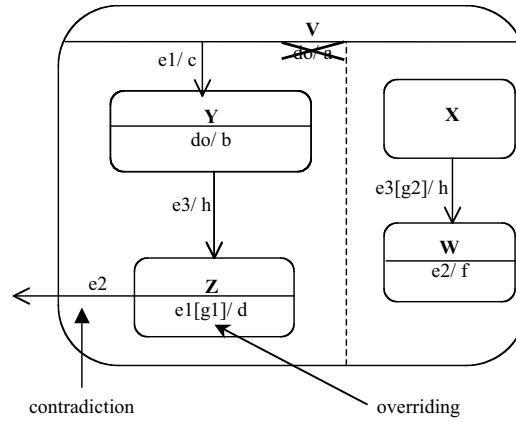


Figure 3.33 – Cas erroné de mise en œuvre des *State Machine Diagram*.

Bref, une bonne approche est l'établissement de règles optimales de modélisation rendant à coup sûr les *State Machine Diagram* intelligibles sans difficulté, ou au minimum, interprétables (*i.e.* simulables) via des ateliers de génie logiciel. Voici des scénarios d'interprétation :

- $Z \wedge e1 \wedge \text{not } g1 \rightarrow c \text{ activée} \wedge Y$

- X ∧ Y ∧ e3 ∧ g2 → h activée *deux fois*
- X ∧ Y ∧ e3 ∧ not g2 → e3 ignoré
- W ∧ Y ∧ e3 → e3 ignoré

Notons que de tels outils logiciels existent et que par définition, ils utilisent chacun leur propre sémantique d'exécution. Rhapsody ou TAU (voir la section « Webographie » en fin de chapitre) sont de bons exemples.

Notions avancées d'exécutabilité

Dans la communauté scientifique et industrielle qui gravite autour d'UML, l'engouement et l'accent mis sur MDA/MDE sont intimement corrélés, dans la littérature spécialement, à la notion de modèle exécutable [10]. Nous sommes néanmoins bien loin du but, et comme nous l'avons déjà souligné, nombre de notions d'exécutabilité restent à coucher sur le papier. Celle de ressource consommable en est une et s'avère vitale. En effet, nous raisonnons dans les *State Machine Diagram* en termes de types mais l'exécution concerne les instances et donc les occurrences d'événements. Simplement, une occurrence d'événement peut-elle être consommée par plus d'une machine à états d'instance ?

Soient ainsi le type *A* et le type *B* qui présentent tous deux une réaction au type d'événement *e* dans l'état *S1* pour *A* et l'état *S3* pour *B* (à gauche de la figure 3.34). Sous l'hypothèse qu'une occurrence d'événement est non partageable, l'apparition d'une occurrence de *e* fait évoluer la machine d'une instance de *A* (:A dans la figure 3.34) alors qu'une instance de *B* (:B) ne bouge pas. L'idée est que même s'il y avait d'autres instances de *A* dans l'état *S1*, *une et une seule* pourrait consommer l'occurrence de *e* qui disparaîtrait « du système » juste après. À titre de comparaison, le mode d'exécution de Syntropy [2] autorise que plusieurs machines d'instances changent d'état pour une et une seule occurrence d'événement présente.

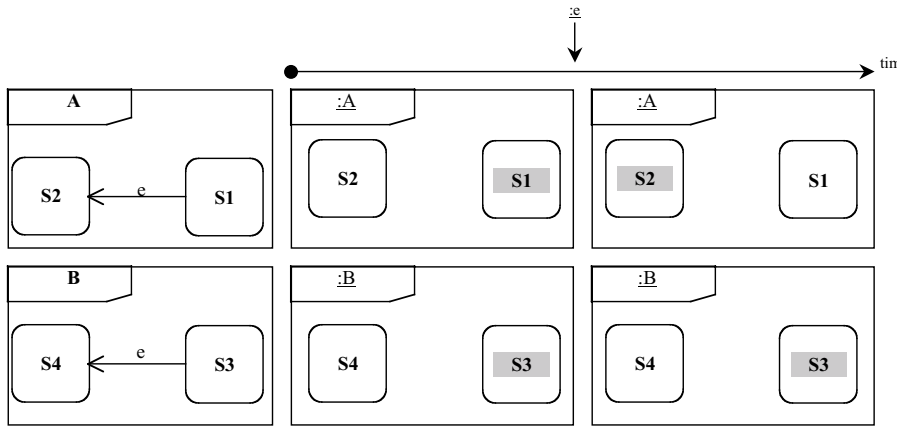


Figure 3.34 – Principe de consommation d'occurrence d'événement.

Il n'est donc pas possible de comprendre correctement les modèles tant qu'un mode d'exécution clair et rigoureux n'est pas défini. Au-delà, c'est la dépendance forte entre deux machines à états de deux types d'objet différents qui est intéressante à étudier.

Composition UML et machines à états, quelques perspectives

Dès lors que l'on a une relation « intime » entre deux types (la relation de composition étant de ce genre, cela au regard des dépendances de vie qu'elle sous-entend et de l'idée d'appartenance du composé au composite qu'elle recouvre), il est possible d'imaginer une sémantique particulière de l'exécutabilité. En figure 3.35, si *B* « appartient à » *A* alors *S4* n'est jamais qu'un état de *A*. Il en résulte que contrairement à ce qui est dit dans la figure 3.34, la même occurrence d'événement agit sur deux instances différentes de deux classes reliées par composition.

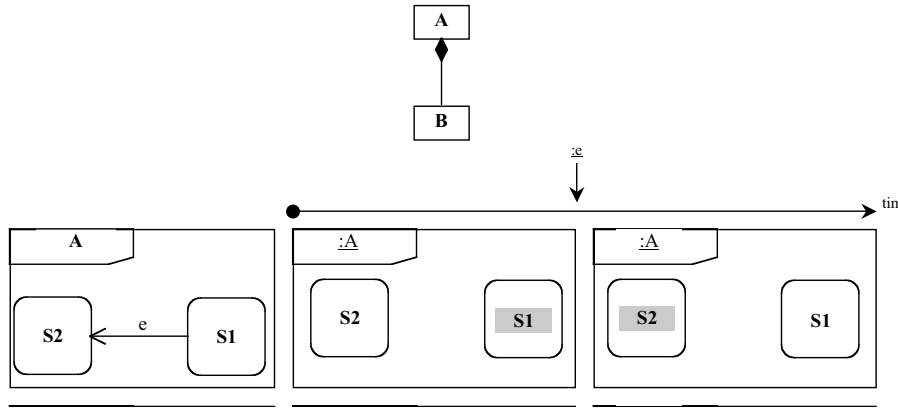


Figure 3.35 – Implication entre relation de composition et *State Machine Diagram*.

La question qui en découle est : le modèle à gauche de la figure 3.35 (*statechart* de *A* et *statechart* de *B*) n'est-il pas équivalent au *statechart* de *A* (à gauche de la figure 3.36) où *B* est vu comme un simple sous-état parallèle à *S1* et à *S2* ? D'autres questions fusent : sous quelles conditions utiliser l'opérateur *in* vu en figure 3.31 ? Autrement dit, *A* ne peut interroger l'état de *B* que si et seulement si *A* et *B* sont en relation de composition (appartenance totale, absence de partage) avec *A* composite et *B* composé. On pourrait imposer cette hypothèse.

L'essentiel, ce qu'il faut retenir

Toutes ces questions sont en suspens et y répondre serait élaborer son propre UML. Notre but est de sensibiliser le lecteur au fait qu'à un stade avancé de pratique, ces problèmes apparaîtront et qu'il vaut mieux apprendre UML sous un œil critique tel que nous le faisons ici, que de croire qu'une simple notation graphique résout toutes les difficultés.

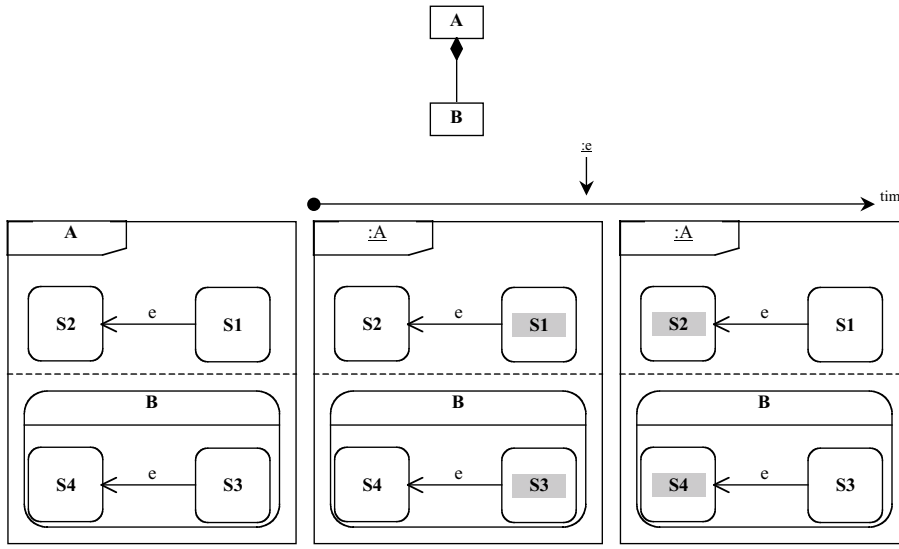


Figure 3.36 – Modélisation intuitive d'un composé par inclusion du *statechart* de son composant.

3.5 AUTRES FORMALISMES DE BEHAVIOR

Nous donnons diverses autres notations, d'UML 2.x plus spécialement, sans être réellement convaincu de leur utilité (pour ne pas dire plus). Concernant les *Activity Diagram*, le chapitre 6 présente une étude de cas les utilisant plus concrètement, et surtout les comparant aux *State Machine Diagram* sur un cas réel cette fois. Il reste alors à examiner dans cette fin de chapitre les *Sequence Diagram* (des compléments sont aussi donnés dans les chapitres 5 et 6), les *Collaboration Diagram*, les *Communication Diagram* et les *Use Case Diagram*.

Le lecteur se chargera de faire son propre tri et surtout sa propre évaluation, parce que par goût tout simplement, il n'adhérera pas à notre mise en exergue des *State Machine Diagram*. Il peut préférer les *Collaboration Diagram* par exemple, abordés dans cette section. En tout état de cause, le but de ce chapitre est de faire un tour d'horizon complet, d'où la nécessité de présenter les éléments essentiels de *Behavior*. Néanmoins, notons que la bonne compréhension des prochains chapitres ne nécessite pas d'investissement intellectuel soutenu sur ces « autres formalismes » de *Behavior*.

3.5.1 Représentation de notions dynamiques sous forme de classe

Les concepts phare de *Behavior* sont matérialisés dans le métamodèle par des métatypes. *Collaboration* est ainsi membre du métamodèle d'UML et hérite de *StructuredClassifier* et *BehavioredClassifier* (des spécialisations de *Classifier* appartenant au

noyau du métamodèle). La dérivation à partir de *Classifier* autorise alors la représentation d'une instance du métatype avec un stéréotype dédié et/ou un signe distinctif (*cf.* celui fait pour les composants, instances de *Component*, vu au chapitre 2). Cette représentation opère le plus souvent, logiquement, dans un *Class Diagram*.

Dans la figure 3.37, il est ainsi possible de représenter des instances du métatype *Signal* : « Brûlé par de l'eau » (*Scalded*). Celui-ci hérite d'une sorte de signal plus général : « Brûlé » (*Burnt*). Il en est de même pour le métatype *Use Case* par exemple, qui comme *Component*, dispose d'un signe distinctif : c'est une ellipse à plat (figure 3.38), le stéréotype *«use case»* bien qu'admissible devenant alors redondant.

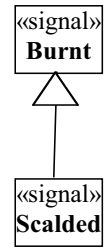


Figure 3.37 – Signaux UML vus comme des classes.

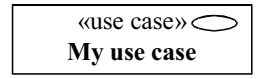


Figure 3.38 – *Use Case* vu comme une classe.

L'interrogation qui apparaît dans une telle démarche de modélisation est de savoir s'il est pertinent de relier des classes comme celles des figures 3.37 et 3.38 à des classes « plus classiques ». Par exemple, relier le signal *Burnt* à une classe *Individual*, pour dire que l'événement *Burnt* concerne un objet de type individu (cardinalité à définir). Cela semble aller dans l'esprit d'une mise en cohésion des modèles statiques et dynamiques (leitmotiv de ce début de chapitre) mais à grande échelle et en pratique, ce n'est pas sans créer un surcroît de travail et une certaine lourdeur.

3.5.2 Diagrammes de collaboration et de communication

Il est fondamental de noter ici que les *Collaboration Diagram* d'UML 1.x (toutes leurs constructions notationnelles en l'occurrence) se divisent maintenant dans UML 2.x en *Collaboration Diagram* et *Communication Diagram*, ces derniers étant inclus dans l'ensemble du formalisme des *Sequence Diagram*. Cette analyse qui nous est propre est formellement déduite de la figure 348, présente dans la documentation [14, p. 446]. Contradictoirement, le glossaire des termes et expressions [14, p. 10] dit que les *Communication Diagram*, les *Sequence Diagram* et les *Interaction Overview Diagram*

sont inclus dans la famille des *Interaction Diagram*, sachant que les *Interaction Overview Diagram* sont une variation des *Activity Diagram* (sic). Cette extrême confusion est regrettable, car la classification que nous donnons au début du chapitre 2 n'est pas non plus en phase avec le discours qui précède (elle est aussi cependant dans la documentation). Mais il s'agit d'UML (bref, c'est le cirque !).

Tentons un éclaircissement en prenant des exemples : la droite de la figure 3.39 est un diagramme de communication en UML 2.x alors que la gauche est un diagramme de collaboration en UML 1.x. On remarque que le modèle de droite est encadré par un rectangle dont l'identité en haut à gauche est : ***sd*** *Shampoo* (***sd*** signifiant *Sequence Diagram*). Cela n'est pas une erreur de notre part, nous ne faisons que nous conformer à la figure 348 de la documentation.

Sur le fond, nous décrivons en relation avec le modèle de la figure 3.21, l'ordre des opérations d'une partie d'un processus de fabrication de shampooing (voir aussi figures 3.22 et 3.23). Le phénomène numéroté *1* ou *1.a* (*filling stopped*) est l'avertissement du capteur de remplissage au container de fabrication de shampooing qui lui lance (opération *turn on()* numérotée *1.1* ou *1a.1*) le mélangeur (*Mixer*) puis sollicite la fermeture (*to be closed*) de la valve d'arrivée des ingrédients (*Primary inlet valve*).

On remarque tout d'abord en figure 3.39 que les deux modèles comportent bien des instances (pour être précis des instances d'*InstanceSpecification*), cela dans l'esprit des *Sequence Diagram*. Ensuite, les labels étiquetant les flux sont soit des événements/messages (*filling stopped* et *to be closed*), c'est-à-dire des unités de communication dont la forme à l'implémentation est maintenue abstraite, soit des opérations comme *turn on()* d'où les parenthèses (voir aussi la figure 3.21 où apparaît le service *turn on()* dans la partie basse de la classe *Mixer*). Les flux sur les modèles de la figure 3.39 mélangent donc événements/messages d'une part et opérations d'autre part, alors que dans les *State Machine Diagram* la distinction est plus naturelle. Dans l'esprit, même si les deux types de diagramme de la figure 3.39 se ressemblent fortement, celui d'UML 1.x est plus orienté implémentation alors que celui d'UML 2.x est d'un niveau conceptuel plus élevé.

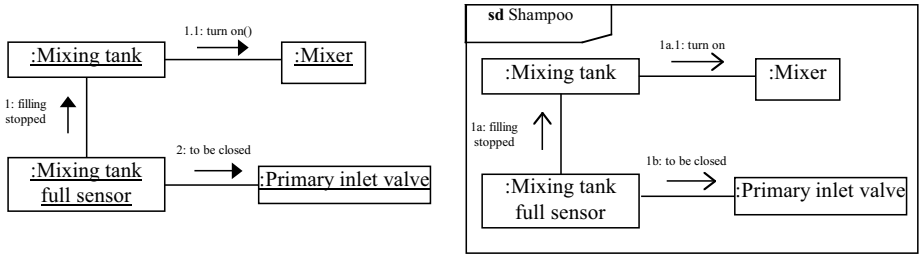


Figure 3.39 – Diagramme de collaboration 1.x contre diagramme de communication 2.x.

Retour sur la notion de collaboration en UML 1.x

En UML 1.5, la définition exacte est : « *A collaboration diagram presents either a Collaboration, which contains a set of roles to be played by Instances, as well as their required relationships given in a particular context, or it presents a CollaborationInstanceSet with a collection of Instances and their relationships. The diagram may also present an Interaction (InteractionInstanceSet), which defines a set of Messages (Stimuli) specifying the interaction between the Instances playing the roles within a Collaboration to achieve the desired result.* » [13]. Les *Collaboration Diagram* résultent plus généralement de la notion de *Mechanism* de Booch [1] également reprise dans la méthode Syntropy [2].

En regard de la définition qui précède, le modèle à gauche de la figure 3.39 présente une interaction alors que l'on peut avoir des diagrammes de collaboration sans interaction (figure 3.40). Le modèle de la figure 3.40 est normalement déduit d'un *Class Diagram* (figure 3.41) et est de niveau dit « spécification ». Il faut comprendre là que dans la figure 3.40, ce sont des rôles qui sont décrits. Le symbole / préfixe le nom du rôle alors que le symbole * est le fait de désigner un ensemble d'objets plutôt qu'un seul objet. Un *Collaboration Diagram* sans flux de données/contrôle (figure 3.40), flux

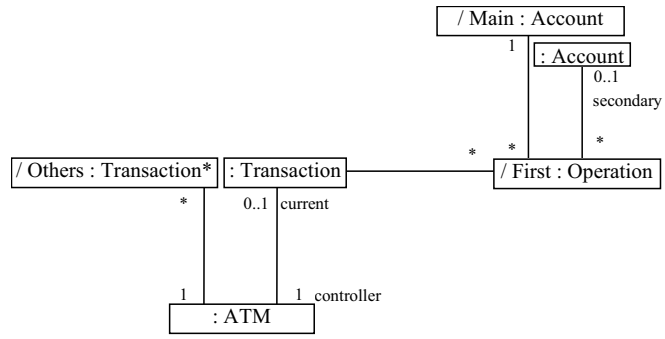


Figure 3.40 – *Collaboration Diagram* en UML 1.x « niveau spécification ».

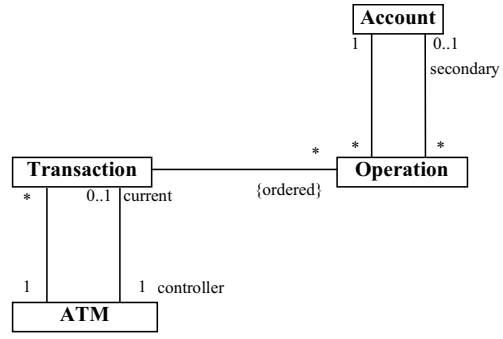


Figure 3.41 – *Class Diagram* permettant de dériver le *Collaboration Diagram* de la figure 3.40.

matérialisé par des flèches, donne un « contexte » mais pas une interaction proprement dite entre objets. Dans le schéma de la figure 3.40, une instance de la métaclasse *ClassifierRole* est introduite et nommée *Main* pour caractériser le compte principal d'un client où s'effectue ses opérations bancaires. Il en est de même pour les *ClassifierRole*, *First* et *Others*. Pour ce dernier, l'étoile dans le coin droit de la boîte indique que le rôle désigne un ensemble d'objets de type *Transaction*.

Par complémentarité, en UML 1.x, un *Collaboration Diagram* de niveau dit « instance » (figure 3.42) est censé être conforme au modèle de la figure 3.41. Il semble à la lecture de la documentation d'UML 1.x que la description de l'interaction n'est justement possible que pour les *Collaboration Diagram* de niveau instance. À ce titre, le modèle de gauche dans la figure 3.39 est donc de niveau instance. Un *Collaboration Diagram* de niveau instance augmente donc un *Collaboration Diagram* de niveau spécification avec des flux de données/contrôle dotés d'un pseudo-ordre via une numérotation des interactions élémentaires.

Ainsi, par exemple en figure 3.42, l'exécution de la fonction de création d'opération bancaire (*1*) entraîne la création d'une instance d'*Operation* (*1.1*). Les contraintes prédéfinies *{new}* montrent l'instanciation de l'opération jouant le rôle *First* ainsi que la création de deux liens : l'un sur le compte jouant le rôle *Main* ainsi qu'un autre, bien que le modèle de la figure 3.40 dise que le rôle *secondary* a une cardinalité fixée à *0..1*, c'est-à-dire qu'il n'existe pas forcément un autre compte bancaire que *Main*. Tout comme les scénarios, on voit donc bien que l'on ne peut pas gérer l'explosion combinatoire, c'est-à-dire décrire toutes les variantes d'interaction possibles. En fait, la création du second lien appelé *secondary* n'a lieu que pour les transferts bancaires car ils sont les seules opérations bancaires à impliquer deux comptes. Comment faire apparaître cela avec la notation offerte par UML 1.x ? Il faudrait peut-être dessiner deux *Collaboration Diagram*, un pour les transferts, un autre pour les opérations bancaires qui ne sont pas des transferts.

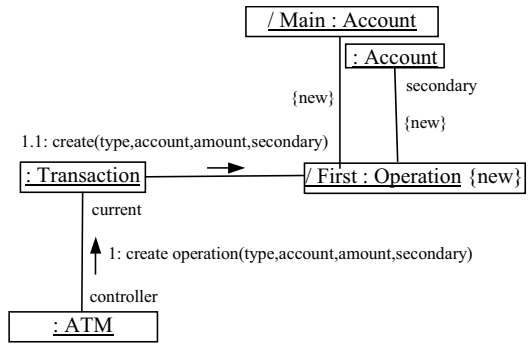


Figure 3.42 – *Collaboration Diagram* en UML 1.x « niveau instance ».

L'essentiel, ce qu'il faut retenir

Nous n'encourageons pas l'utilisation de ce formalisme parce que nous pouvons nous en passer. Dans le chapitre 6, nous reprenons ce cas bancaire relatif aux figures 3.40, 3.41 et 3.42 et montrons qu'avec les seuls *State Machine Diagram*, nous arrivons à une spécification de bonne qualité (meilleure ? au lecteur de juger). Par ailleurs, nous insistons de nouveau sur le fait que ces modèles sont devenus dans UML 2.x des modèles dit de communication et donc il y a lieu de repenser leur usage en conjonction avec d'autres types de modèle de *Behavior* ou en fonction du type d'application à traiter.

Collaboration Diagram, *notations communes entre 1.x et 2.x*

Revenons sur le redoutable imbroglio en UML 2.x concernant la classification des types de diagramme. Comme l'annonce le chapitre précédent, les *Collaboration Diagram* appartiennent à *Behavior*. Dans ce chapitre, nous montrons aussi qu'ils sont apparentés et/ou appariés à d'autres types de diagramme de *Behavior*. Dans la documentation d'UML 2.x, ils sont en toute contradiction présentés dans le chapitre *Composite Structure Diagram* et donc par voie de conséquence dans la partie *Structure*. Il faut déduire de cela qu'une collaboration est une structure composite.

Dans le schéma de la figure 2.43, l'ovale en pointillé est la notation canonique d'une collaboration. C'est un formalisme commun à UML 1.x et UML 2.x. Il existe aussi la possibilité de donner une vue interne qui correspond grosso modo à inclure dans l'ovale de la figure 2.43, le contenu de la figure 2.40.

Figure 3.43 – Vue macroscopique de la collaboration en figure 3.40.

Dans la figure 3.44, nous manipulons une collaboration nommée *Context* comme un *Classifier*. À cet égard, la relation de réalisation montre que *Context* réalise tout ou partie des fonctionnalités d'un type *Context manager*, ou plus précisément à droite, que *Context* implémente le service *analyze* offert par le type *Context manager*.

Cette notation est empruntée à UML 1.x et les exemples de collaboration sont si peu nombreux dans la documentation d'UML 2.x qu'il est difficile de dire si une telle schématisation perdure. Conceptuellement, considérer une collaboration comme une instance de *Classifier* reste valide en UML 2.x, et le modèle de la figure 3.44 conserve donc une certaine acuité en UML 2.x.

Selon le même raisonnement, la figure 3.45 reprend beaucoup de constructions de modélisation d'UML 1.x avec la présence explicite de relations. Ainsi, la relation de composition (losange noir) montre l'appartenance ensembliste des instances nommées *old* et *new* au *multiobject* (concept UML 1.x) identifiable par les rectangles

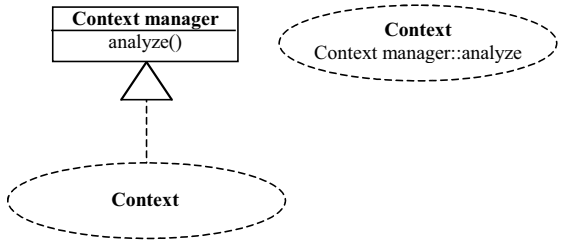


Figure 3.44 – Collaboration, variations de modélisation.

qui se superposent en bas à gauche dans le schéma. L'ordre des exécutions apparaît avec des annotations comme * (à la suite du numéro d'opération 1.2.2) qui signifie plusieurs appels (autant qu'il y a d'instances dans le *multiobject*). Les autres annotations sont la contrainte *{new}* vue auparavant et son pendant *{destroyed}*.

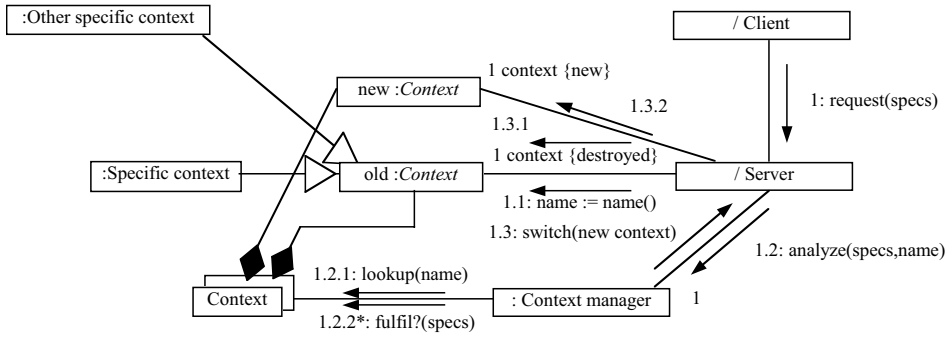


Figure 3.45 – Interaction (vue interne) de la collaboration *Context* de la figure 3.44.

Le danger, ce dont il faut être conscient

En conclusion intermédiaire, nous pouvons dire que les *Collaboration Diagram* sont proches de l'implémentation et sont utilisés en tant que tels, par exemple dans Syntropy [2, p. 152-192]. Dans UML 1.x lui-même, ils sont préconisés pour détailler le scénario d'exécution d'une opération importante d'un type d'objet nécessitant le plus souvent l'utilisation de services d'autres types. Le formalisme est extrêmement intuitif et donc hasardeux dans son usage. Personnellement, nous l'évitons et considérons qu'un pseudo-code est beaucoup plus efficace pour décrire les aspects relatifs à l'implémentation, s'il y a lieu. Rappelons que les *Collaboration Diagram* sont des modèles dérivés et donc d'une certaine manière sont subalternes. Ils correspondent bien sur le fond à l'esprit d'UML qui consiste à accroître exagérément le spectre des notations au détriment de la rationalité du formalisme.

3.5.3 Cas d'utilisation (*Use Case Diagram*)

Les cas d'utilisation ont intégré UML à partir de la version 0.9[1] avec l'arrivée de Jacobson comme coauteur d'UML et par là même sous l'influence d'Objectory/OOSE [8].

Le danger, ce dont il faut être conscient

Avant d'en venir aux considérations techniques, il nous est difficile de ne pas livrer ici nos sentiments profonds sur la technique des *Use Case*. Unanimement reconnus comme cantonnés à l'ingénierie des besoins, les *Use Case* n'ont à notre sens rien à faire dans le noyau d'UML, où la problématique est par essence la *modélisation*. Nous prétendons que l'ingénierie des besoins n'est pas la modélisation parce que représenter deux cas d'utilisation comme nous le faisons en figure 3.46, ne peut sérieusement pas être appelé « modélisation ». Rappelons de plus que dans la méthode standard des *Use Case*, beaucoup d'éléments descriptifs sont en langage naturel : quel rapport avec la modélisation ?

À ce jour, dans UML 2.x, le plus grand confinement des *Use Case Diagram* est une bonne nouvelle. En effet, en comparaison d'UML 1.x, leur part semble décroître au profit d'autres types de diagramme. De façon informelle et perceptible, le discours marketing autour d'UML 2.x semble aussi confirmer que les enjeux nouveaux portent sur le MDE et XMI, le format XML d'échange de modèles entre outils. À la création d'UML, notamment la version 1.1, la présence des cas d'utilisation était en effet un fort vecteur de promotion du langage.

Un cas d'utilisation est à notre goût anti-objet. C'est une fonctionnalité de taille conséquente rapportée à un utilisateur ou rôle type, ou encore *actor* dans la terminologie consacrée. Dans la figure 3.46, la fonctionnalité *Fault location* (localisation de panne) est de la responsabilité d'un opérateur (*Operator*) dans un système de gestion de réseau sous-marin de télécommunication (STNM, *Submarine Telecommunications Network Management*). Capturer les besoins, les découvrir, les réfuter, les consolider, etc., prouve s'il en est besoin que la technique des *Use Case* correspond à la bonne veille « analyse fonctionnelle ». *Nothing's new under the sun…*

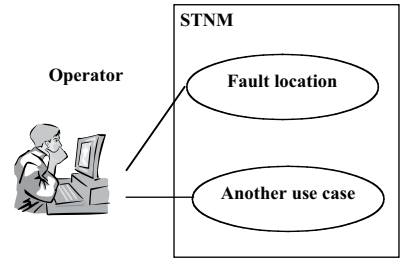


Figure 3.46 – Exemple de *Use Case*.

1. Ils étaient absents de la version 0.8 d'UML appelé *Unified Method*, alors sous la coupe unique de Booch et Rumbaugh.

L'idée originelle de Jacobson était néanmoins riche et intéressante. Devant le constat de la quasi-impossibilité de trouver les objets (voir chapitres 4 et 5 qui parlent tous deux de cueillette des objets), il proposait de trouver les cas d'utilisation puis de les décliner en objets ainsi qu'en relations (structure) et interactions (comportement). Son ouvrage fondateur [8] traite parfaitement et clairement du sujet. Notre reproche est que cette idée a été dénaturée par omission et/ou occultation de la seconde phase. Dans le courant des années quatre-vingt-dix, on s'est gargarisé de « comment trouver les *Use Case* ? », « comment les décrire ? » (en langage naturel surtout), « comment les relier ? », etc. Notre opinion est qu'il aurait plutôt fallu mettre l'accent sur « comment dériver de manière efficace et sûre des modèles objet à partir de cas d'utilisation ? » Nous traitons ce point dans la section sur les *Sequence Diagram* en développant l'étude de cas télécoms de la figure 3.46.

UML 2.x Dans la figure 3.47, on s'intéresse aux représentations de base des cas d'utilisation telles qu'elles sont établies dans la version 2.x d'UML. Dans la philosophie d'UML 2.x, il y a la notion de *subject* incarnée ici par *That's life system*. Le cas d'utilisation *Do this* par exemple est considéré s'appliquer au *Classifier*, *That's life system*. Par comparaison, le cas d'utilisation *Do that* appartient au *Classifier ClassifierThatOwnsDoThat*. Une telle appartenance n'est pas exclusive. Plus généralement donc, un même cas d'utilisation peut appartenir et/ou s'appliquer à différents *Classifier*. Au-delà, la droite de la figure 3.47 comporte des notations alternatives dont l'intérêt, selon nous, n'est pas toujours évident : *Do this* vu comme un *Classifier* dans un rectangle ou encore *Do that* dans une ellipse dessinée au sein de *ClassifierThatOwnsDoThat* pour marquer l'appartenance.

Autre façon de dire, il faut bien distinguer en UML 2.x, l'appartenance d'un cas d'utilisation à une instance de *Classifier* (*e.g. Do that* et *ClassifierThatOwnsDoThat*) de la relation « s'applique à » (qui n'a pas par ailleurs de stéréotype attitré) concernant aussi un *Classifier* (*e.g. Do that* et *That's life system*).

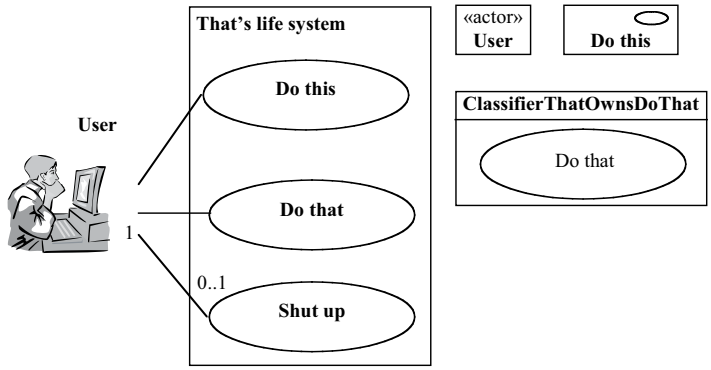


Figure 3.47 – Différents formalismes associés aux *Use Case* dans UML 2.x.

Dans des schémas comme celui de la figure 3.47, on peut exprimer des cardinalités comme cela est fait entre l'acteur *User* (aussi représenté par une boîte et le stéréo-

type *«actor»* dans la figure 3.47) et le cas d'utilisation *Shut up*. C'est le troisième grand type de relation entre un cas d'utilisation et « quelque chose de son environnement » : l'association est ici en l'occurrence une relation ordinaire dans UML.

Relations entre cas d'utilisation

Les trois relations usuelles (figure 3.48) sont :

- *«extend»* ;
- *«include»* ;
- et l'héritage.

Concernant la troisième et dernière relation, du fait que la métaclasse *UseCase* hérite de *BehavioredClassifier*, les instances de *UseCase* dans les modèles peuvent donc intervenir dans des relations de généralisation (à droite de la figure 3.48).

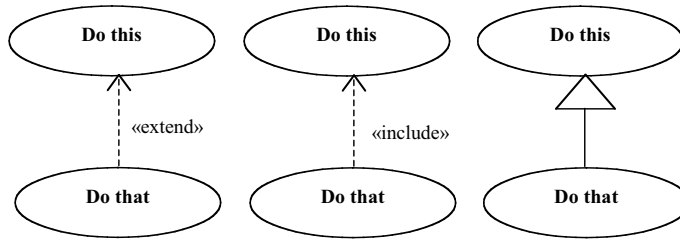


Figure 3.48 – Relations entre *Use Case*.

Les relations *«extend»*, *«include»* et même l'héritage restent peu claires en UML 2.x. Nous ne nous hasardons pas à les expliquer car nous doutons qu'elles aient eu un jour une quelconque sémantique. La relation *«extend»* par exemple, semble ajouter des bouts de fonctionnalités à une fonctionnalité existante mais une telle vision est bien grossière.

« Points d'extension » mais pas de révolution…

Rien de bien nouveau donc par rapport à UML 1.x concernant les cas d'utilisation, sinon la notion de *subject* plus explicite en UML 2.x. Venons-en maintenant à ce qui nous semble être une réelle nouveauté : les points d'extension (figure 3.49). Ils permettent d'exprimer comment s'opère l'extension d'un cas d'utilisation par un autre, cela en réponse au problème précédemment soulevé sur le sens profond de la relation *«extend»*.

On joue de la notation dans la figure 3.49. Le cas d'utilisation *Do this* bénéficie à droite du stéréotype *«use case»* alors que le symbole de l'ellipse avait été préféré dans la figure 3.47 (partie droite, en haut). Dans la figure 3.49, l'idée est de dire que le cas d'utilisation *Do that* étend *Do this* (partie gauche). Une représentation (des plus douteuses à notre goût mais conforme à UML 2.x) du cas d'utilisation *Do this* par un

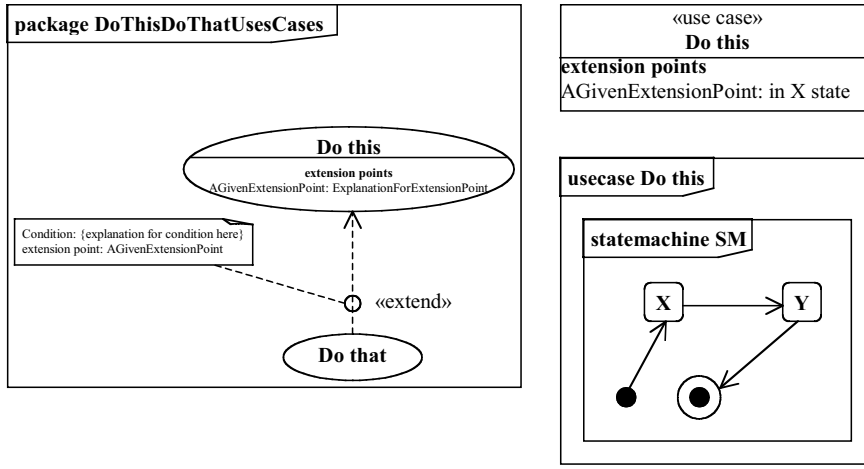


Figure 3.49 – Mise en œuvre plus poussée de la notation *Use Case*.

automate possédant l'état X, fait dire que le point d'extension *AGivenExtensionPoint* est pour *Do this* d'être dans l'état X (expression *in X state*, figure 3.49, en haut à droite).

Toute cette notation est selon nous obscure. Il y a beaucoup de fioritures dans les constructions de modélisation offertes. À ce titre, la documentation actuelle [14, p. 511-527] n'est en rien un guide méthodologique. On n'est en fait jamais en mesure de déterminer en quelles circonstances (type d'application, domaine…), tel ou tel élément, telle ou telle règle de « modélisation », doit être mis en œuvre.

3.5.4 Scénarios (*Sequence Diagram*) et autre exemple de synthèse

Les scénarios sont un moyen de modélisation intuitif, plaisant et à succès, dans le domaine des télécoms notamment. Pour parler des *Sequence Diagram* d'UML, nous nous intéressons à une topologie de réseaux sous-marins à fibre optique (figure 3.50), cas d'étude qui nous a été fourni par la société Alcatel Submarine Networks (voir section « Webographie » en fin de chapitre). Les aspects didactiques de cette application sont (a) d'expérimenter et de comprendre les facteurs d'échelle, (b) l'utilisation intensive et centrale des scénarios dans la modélisation et finalement, (c) les contraintes que peut poser le respect de normes de domaine, ici des normes d'administration de réseaux propres aux télécoms.

Pour le point (a), l'idée est de regarder si la mise en œuvre d'UML est de même difficulté pour des applications de grande envergure ou des « petites » applications. Cette préoccupation connue sous le vocable anglais *scalability* fait référence à la reproductibilité, à grande échelle, de la démarche de modélisation UML. En d'autres termes, la façon de procéder pour concevoir de petites applications n'est pas forcément pertinente dès lors que l'application nécessite le travail de nombreuses personnes (experts du domaine, analystes, concepteurs, développeurs…), de longs mois de

travail (fluctuations importantes des besoins), que le nombre d'outils et couches logiciels est conséquent et que ces outils sont éclectiques.

Quant au point (b), la technique de modélisation par scénarios, dont la première formalisation solide a donné naissance aux MSC (*Message Sequence Charts* [7]), est populaire et incontournable dans le monde des télécommunications. Les *Sequence Diagram* d'UML 2.x, bien qu'ayant fortement évolué depuis UML 1.x, restent néanmoins incomplets et imprécis pour être appliqués ici. Ces deux outils partagent des principes de modélisation communs du fait que les *Sequence Diagram* d'UML s'inspirent beaucoup des MSC (notion de corégion par exemple) tout en s'en différenciant. Les MSC sont adaptés à la spécification de protocoles de communication bas niveau alors que les *Sequence Diagram* d'UML visent les applications métier de grande taille avec abondance de notations. Harel résume néanmoins le problème de façon plutôt péremptoire : « *As a requirements language, all known versions of MSC, including the ITU standard and the sequence diagrams adopted in UML, are extremely weak in expressive power.* » [5]. Dans l'esprit de cette critique acerbe, c'est via OCL que nous proposons une approche améliorée pour utiliser les *Sequence Diagram* d'UML 2.x. Par ailleurs, dans l'esprit de notre discours sur les *Use Case*, nous montrons ici la manière dont un cas d'utilisation peut se décliner en scénario et donc créer une réelle dynamique pour découvrir les objets. Nous traitons ainsi le cas d'utilisation *Fault location* de la figure 3.46.

Le dernier point (c) est relatif à la norme GDMO (*Guidelines for the Definition of Managed Objects*, voir section « Webographie » en fin de chapitre) qui est une norme d'administration de réseaux. Imposée dans l'élaboration de systèmes STNM, son usage concomitant à UML force la nature profonde des modèles, ceux de conception notamment qui doivent donner des architectures conformes aux préceptes de cette norme. La cohabitation GDMO/UML est donc critique dans le processus de conception. C'est en travaillant sur UML et en préparant des formes prédéterminées de modèles qu'il est permis de spécifier l'application en respectant GDMO. Ces formes sont bien entendu des patrons, mais cette fois-ci, des patrons « métier » permettant de capitaliser des micro-architectures modélisant le domaine des systèmes STNM.

Dans la figure 3.50, *SLTE* signifie *Submarine Line Terminal Equipement* et correspond à la terminaison du câble sous-marin au niveau terrestre. *WDM* signifie *Wavelength Division Multiplexing* et correspond au système de transmission logique au niveau multiplex. *BU* signifie *Branch Unit* et correspond aux moyens de commutation des communications sur le câble. Ces unités assurent également, comme les répéteurs de signaux (*Repeater*), une fonction d'amplification du signal optique. La partie ou couche multiplex apparaît tout en bas de la figure 3.50. La partie longueur d'onde (transmissions optiques) apparaît au-dessus, sous forme de « sur-couche », avec différents concepts dont le nom est le plus souvent préfixé de l'adjectif anglais *Optical*. Un chemin optique par exemple (*Optical path*, en haut, à gauche de la figure 3.50) est un canal calculable à partir des différents points de connexion optique (*Optical path layer CTP Bidir* et *Optical path layer TTP Bidir*), eux-mêmes ayant des correspondances au niveau multiplex (*WDM layer TTP Bidir*). L'ensemble est un

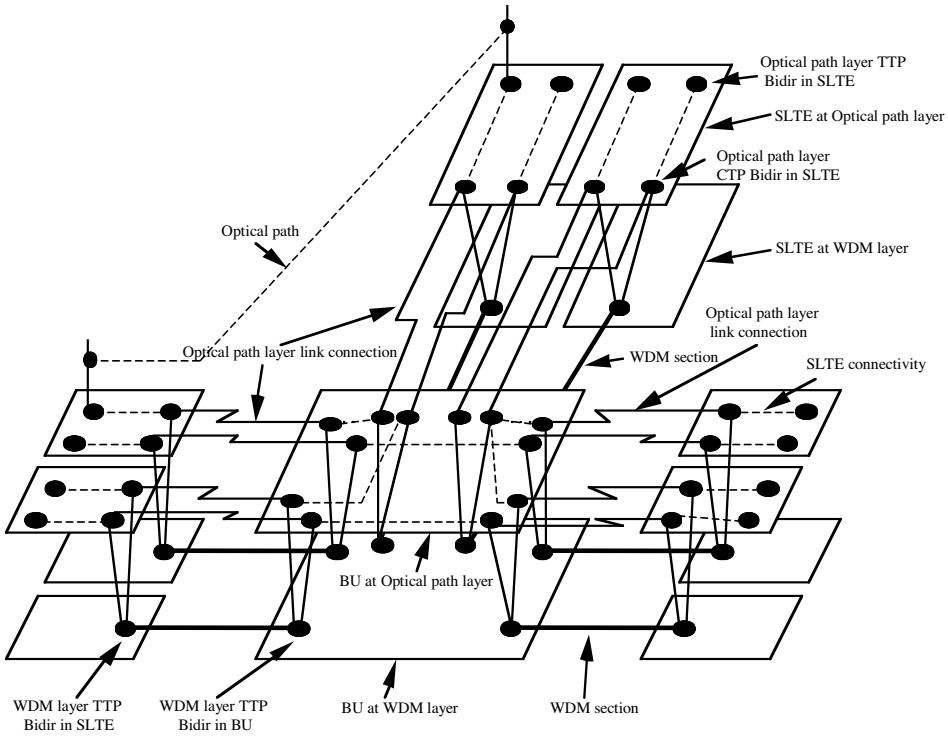


Figure 3.50 – Topologie générale de réseaux sous-marins de télécommunication à fibre optique.

graphe dont les arcs sont de nature différente selon la couche à laquelle ils appartiennent : *WDM section* au niveau multiplex et *Optical path layer link connection* au niveau longueur d'onde.

Modélisation des aspects statiques

Cette section donne les *Class Diagram* et contraintes OCL modélisant la topologie de la figure 3.50 (figure 3.51) ainsi que des types d'objet d'administration qui ne peuvent pas être déduits de la figure 3.50 mais qui sont soit imposés par la norme GDMO (*SLTE agent* par exemple, figure 3.52), soit requis par le cas d'utilisation *Fault location* lui-même pour par exemple, créer des objets relatifs à chaque diagnostic fait à l'issue de chaque panne : c'est le type *Fault diagnosis* de la figure 3.52.

Les invariants du modèle en figure 3.51 sont :

```
context Optical path layer CTP Bidir inv:
   slte connectivity→isEmpty() implies bu connectivity→notEmpty()
   bu connectivity→notEmpty() implies slte connectivity→isEmpty()
context Optical path layer link connection inv:
   support.end.client→includesAll(end)
context WDM layer section inv:
   support.sub-segment end.wdm layer ttp bidir→includesAll(end)
```

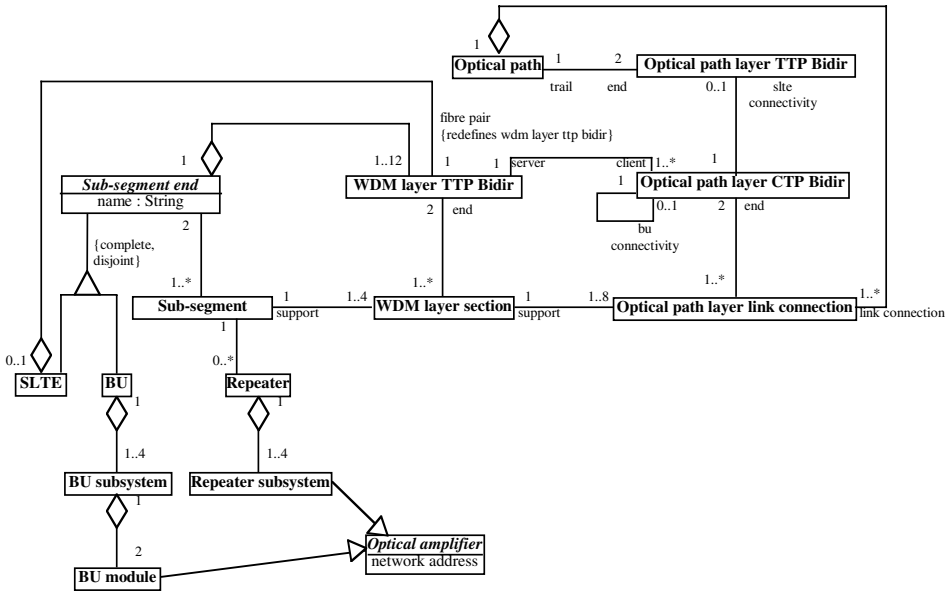


Figure 3.51 – *Class Diagram* du package *Topology*.

Le but des *Class Diagram* et contraintes OCL ici listés est de servir d'adossement aux scénarios à venir. Leur obtention et leur stabilisation, via de nombreuses versions intermédiaires, peuvent être évaluées à 2,5 homme/mois. 3 à 5 personnes physiques différentes, experts en spécification et/ou en UML et/ou en STNM, ont été en moyenne sollicitées. L'explication détaillée de tous ces modèles statiques dépasse donc le cadre de cet ouvrage. Nous invitons le lecteur à se concentrer sur les *Sequence Diagram* et à revenir si nécessaire sur les *Class Diagram* pour comprendre la recherche de cohérence entre modèles statiques et modèles dynamiques.

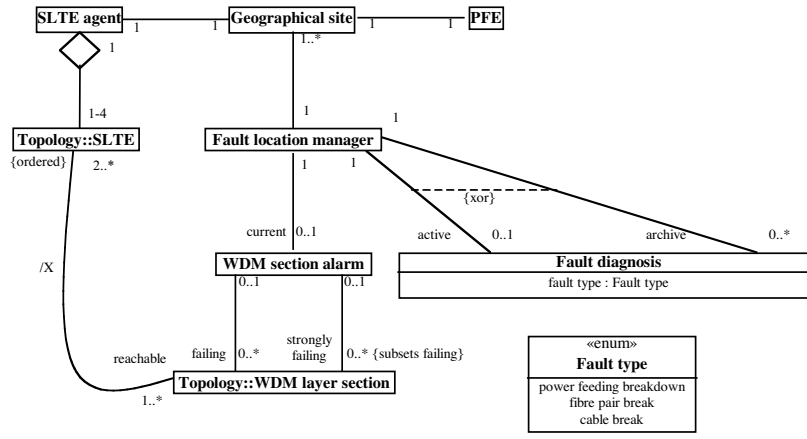


Figure 3.52 – *Class Diagram* du package *Management*.

Dans la figure 3.52, *PFE* signifie *Power Feeding Equipment* : c'est le système d'alimentation du câble au niveau de chaque terminal terrestre. Le système central de gestion des pannes du réseau sous-marin est donc incarné par le type *Fault location manager*. Il gère plusieurs sites géographiques (*Geographical site*) qui ont un et un seul agent SLTE (c'est un concept GDMO) lui-même composé de 1 à 4 terminaux terrestres (*SLTE*). Le but ultime du cas d'utilisation *Fault location* est de créer des objets de type *WDM section alarm* (objets par nature d'administration et donc indépendants de la topologie) au fur et à mesure des alarmes remontant des infrastructures immergées, puis de rattacher les alarmes à des sections du câble au niveau multiplex (rôles *failing* et *strongly failing* vers *WDM layer section*[1]). En complément, les invariants du modèle en figure 3.52 :

```
context SLTE inv X: -- association dérivée /X dans le schéma de la figure 3.52
   reachable = fibre pair.client.slte connectivity.trail.link connection.support
context SLTE inv:
   reachable→includesAll(fibre pair.wdm layer section)
```

Ces deux invariants sont cruciaux dans l'administration du système, en particulier la navigation *reachable*, arrivant à *WDM layer section* en partant de SLTE, qui décrit l'ensemble de tous les bras (« bras » étant synonyme de section ou encore de segment) multiplex (*WDM section*) atteignables via une instance de terminal terrestre donnée. Le second invariant concerne l'instance de *WDM layer TTP Bidir* (figure 3.51), unique dans chaque terminal terrestre (cardinalité *1* vers *Sub-segment end* dont hérite *SLTE* en figure 3.51). Cette instance de *WDM layer TTP Bidir* correspond aux deux directions de communication (abréviation *Bidir*) de la fibre optique (rôle *fibre pair* vers *WDM layer TTP Bidir* en figure 3.51). Cette instance est l'extrémité (rôle *end* vers *WDM layer TTP Bidir* en figure 3.51) du bras multiplex directement lié au terminal terrestre. L'invariant dit donc « simplement » que ce bras multiplex appartient de toute évidence à l'ensemble des sections multiplex atteignables (*reachable*).

Modélisation des aspects dynamiques

Dans UML 1.x, des types d'interaction différents sont possibles dans les *Sequence Diagram*. Ils sont matérialisés par des flèches de formes variées ayant donc des sens distincts. De plus, les lignes de vie verticales qui indiquent l'évolution des objets dans le temps ont aussi deux présentations possibles. Dans la figure 3.53 par exemple, le trait vertical pointillé est la présentation standard alors que la barre blanche verticale (*a priori* disparue en UML 2.x) dénote un objet « concurrent » ou « actif ». Pour être plus précis, c'est un objet qui a son propre flot de contrôle (voir aussi stéréotype *«active»* réservé aux classes et utilisé dans l'étude de cas du chapitre 5).

Par conséquent, il est logique que le modèle de la figure 3.53 se double du modèle de la figure 3.54[2]. Les deux *statechart* montrent en effet les conditions exactes (états et gardes pour l'essentiel) de réception et d'interprétation des messages. En ce qui

1. Ce type d'objet est important car il est originellement membre, étant donné sa notation dans la figure 3.52, du package *Topology*.

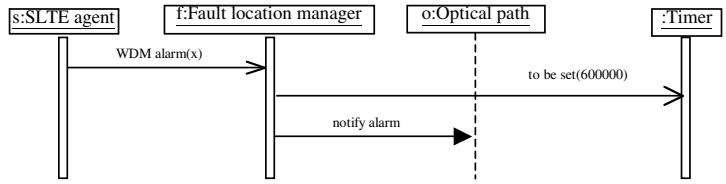


Figure 3.53 – Modes d'interaction (liste non exhaustive) en UML 1.x.

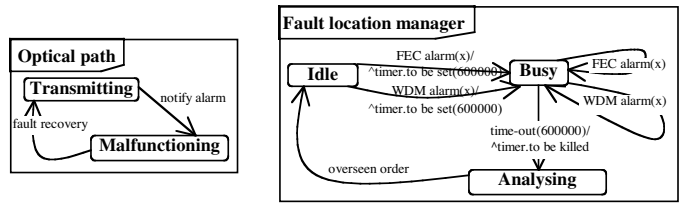


Figure 3.54 – *Statechart* des types d'objet *Optical path* et *Fault location manager*.

concerne les flèches en figure 3.53, celles à extrémité « ouverte » sont des messages asynchrones alors que celle « pleine » (apparemment aussi disparue en UML 2.x) est un « appel d'opération ». Une flèche à extrémité « ouverte » et dont le trait est en pointillé (il n'y en a pas en figure 3.53) est un « retour d'appel d'opération ».

Le danger, ce dont il faut être conscient

Nous goûtons plus que modérément à ce genre de modélisation qui ressemble beaucoup à une représentation graphique d'une quelconque implémentation. Par ailleurs, il y a matière à toutes les erreurs. Par exemple, si un « appel d'opération » arrive sur un objet « actif » (barre blanche verticale) et que repart un « retour d'appel d'opération », c'est totalement incohérent avec le fait que l'objet n'est pas synchronisé et donc disponible pour une telle sollicitation.

Revenant à notre cas télécoms, les modèles de la figure 3.53 et de la figure 3.54 disent qu'une alarme niveau multiplex (message *WDM alarm*) arrive sur le gestionnaire de pannes du réseau qui lance un *Timer* pour être prévenu dix minutes (600 000 millisecondes) plus tard (l'événement *time-out* n'est pas mentionné mais traité par la suite). Un objet de type chemin optique (*Optical path*) est forcé à l'état *Malfunctioning* (figure 3.54). Mais quel chemin optique ? Nous l'avons nommé *o* en figure 3.53 mais le problème historique des *Sequence Diagram* est de travailler avec des instances anonymes et des interactions « un à un » ce qui donne, pour reprendre les termes de Harel, un pouvoir d'expression réduit.

2. Le *statechart* du type d'objet *Timer* est donné et explicité dans le chapitre 5. Il suffit juste de savoir ici que *to be set* arme un *Timer* en millisecondes, *to be killed* le tue et *time-out* est la notification de délai atteint informant le client ayant lancé le service *to be set*.

UML 2.x Dans le glossaire UML 2.x [14], la définition d'une ligne de vie n'apporte pas plus : « *A modeling element that represents an individual participant in an interaction. A lifeline represents only one interacting entity.* » Nous sommes limités par le fait que nous voudrions spécifier non pas des instances anonymes de types d'objet mais des instances précises (ici, le chemin optique parmi tous ceux du réseau qui est en dysfonctionnement), des ensembles d'instances et pas uniquement des individualités (voir plus loin) ou encore comment s'opère l'exécutabilité. Par exemple, si l'objet *Fault location manager* est dans l'état *Analysing* (figure 3.54) et qu'une occurrence d'événement de type *WDM alarm* arrive, cette occurrence est sans effet et est détruite ou alors est-elle mémorisée dans une file d'attente appartenant à l'objet *Fault location manager* ? Nous modélisons pour notre part toujours selon la première approche. En UML 2.x, bien que la forme des interactions ait été rationalisée, des notions de messages trouvés et perdus sont apparues. La documentation d'UML 2.x a certes été dépoussiérée au profit d'un unique profil de flèche (c'est le concept de message en UML 2.x) rejoignant deux lignes de vie. Les lignes de vie elles-mêmes ne se décrivent maintenant plus qu'avec un trait pointillé. La figure 3.55 illustre ces nouvelles données.

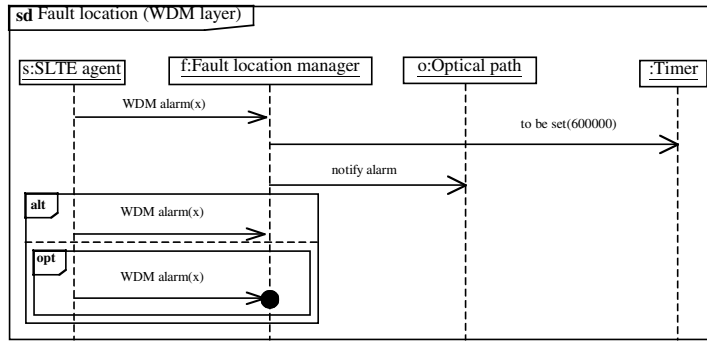


Figure 3.55 – Scénario du cas d'utilisation *Fault location* lorsqu'une alarme est détectée au niveau de transmission multiplex (UML 2.x).

Nous invitons néanmoins vivement le lecteur à ne pas utiliser la notation des messages trouvés (flèche avec point noir recouvrant à son origine) et perdus (flèche avec point noir recouvrant à sa fin, figure 3.55). Là encore, pourquoi et sous quelles conditions particulières un message peut-il être perdu ? Le scénario de la figure 3.55 est à géométrie variable car il dit qu'il y a une option (opérateur *opt*) : le message se produit ou ne se produit pas. S'il se produit, il est perdu. Mais comme le message peut se produire sans être perdu (état *Idle* ou *Busy* de *Fault location manager* en figure 3.54), il faut utiliser l'opérateur d'alternative *alt* qui sépare par un trait pointillé dans un rectangle les deux termes de l'alternative.

À l'aide du métatype *CombinedFragment* qui se matérialise le plus souvent par un rectangle contenu dans le scénario (figures 3.55 et 3.56) épaulé des opérateurs *alt*, *opt*, *break*, *par*, *seq* (ordonnancement lâche), *strict* (ordonnancement strict, figure 3.56), *neg*, *critical* (région critique), *ignore*, *consider* et *assert* pour les plus importants,

L'essentiel, ce qu'il faut retenir

De manière générale, le formalisme des *Sequence Diagram* s'est fortement enrichi avec des modifications notables mais il résout toujours mal les problèmes d'explosion combinatoire : s'il y a mille alternatives, on ne peut les décrire ! À notre avis, sans les *State Machine Diagram* de la figure 3.54, les modèles de la figure 3.53 ou de la figure 3.55 ne peuvent atteindre un bon niveau de qualité. Il apparaît clairement dans cette étude de cas télécoms, mais cela est vrai en général, que les conditions d'interaction doivent être spécifiées entre objets, étant non pas anonymes mais clairement identifiés.

il devient possible d'obtenir un premier niveau de description. Ensuite, avec l'opérateur *assert* par exemple et des *State Machine Diagram*, les conditions d'interaction peuvent être relativement bien formalisées. Nous nous y attachons ici à l'aide d'OCL mais en extériorisant les contraintes des schémas graphiques pour ne pas les alourdir.

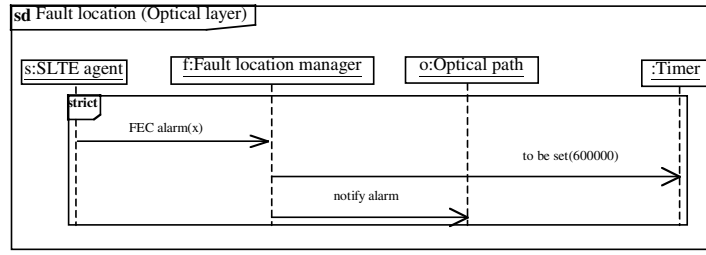


Figure 3.56 – Scénario du cas d'utilisation *Fault location* lorsqu'une alarme est détectée au niveau de transmission longueur d'onde.

Ainsi, le calcul des objets, c'est-à-dire l'expression formelle de leur identité (noms *s*, *f* et *o* donnés aux instances en figures 3.55 et 3.56) peut se faire indépendamment des événements/messages (des attributs que ces derniers véhiculent en particulier) et de leur chronologie/entrelacement. La désignation de ces objets constitue donc des invariants des *Sequence Diagram* de la figure 3.55 et de la figure 3.56 :

```
context s:SLTE agent inv:
   f = geographical site.fault location manager
```

En revanche, les prérequis et les conséquences des événements sont des variants du *Sequence Diagram* de la figure 3.55. Ils sont exprimés par des préconditions et des post-conditions OCL :

```
context Fault location manager::WDM alarm(x : WDM layer TTP Bidir)
   pre: geographical site.slte agent.slte.fibre pair→includes(x)
    -- création si nécessaire d'un objet de type WDM section alarm
    -- pour diagnostic panne
   post: self@pre.oclInState(Idle) implies current.oclIsNew
```

```
   -- détermination des chemins optiques en faute pour identification opérateur
   post: o = x.client.slte connectivity.trail
context Optical path::notify alarm()
   -- accumulation des sections multiplex en faute pour diagnostic panne
   post: f.current.failing = (f.current.failing)@pre→union(link connection.sup-
port)
```

Le calcul des identités d'objet du *Sequence Diagram* de la figure 3.56 donne le même résultat que celui fait pour la figure 3.55, *i.e.* la valeur de *f* est la même. Les préconditions et post-conditions des événements sont les suivantes :

```
context Fault location manager::FEC alarm(x : Optical path layer TTP Bidir)
   pre: geographical site.slte agent.slte.fibre pair.client.slte
        ➥ connectivity→includes(x)
    -- création si nécessaire d'un objet WDM section alarm pour diagnostic panne
   post: self@pre.oclInState(Idle) implies current.oclIsNew
   -- détermination des chemins optiques en faute pour identification opérateur
   post: o = x.trail
context Optical path::notify alarm()
   -- accumulation des sections multiplex en faute pour diagnostic panne
   post: f.current.failing = (f.current.failing)@pre→union(link
        ➥ connection.support)
```

La synthèse (composition) des modèles de la figure 3.55 et de la figure 3.56 donne la figure 3.57. Par convention, nous ne répercutons pas l'événement *notify alarm* de la figure 3.55 et de la figure 3.56 dans la figure 3.57 car il n'entretient pas de relation d'ordre stricte avec tous les nouveaux messages localisés en bas de la figure 3.57. Encore une fois, nous aurions pu utiliser l'opérateur *seq* d'UML 2.x pour décrire un ordre lâche ou faible (*weak order* dans la documentation UML) entre ces nouveaux événements et *notify alarm*. Quel est l'intérêt de décrire des groupes de messages dont l'entrelacement est justement libre ? Aucun.

Le calcul des identités d'objet du *Sequence Diagram* de la figure 3.57 introduit le rôle *s'* qui se différencie du rôle *s* de la manière suivante :

```
context s:SLTE agent inv:
   s' = f.current.strongly failing.slte→first().slte agent
```

En termes algorithmiques, après dix minutes d'écoute des alarmes (niveau multiplex ou niveau longueur d'onde) remontant du réseau, l'objet *f* de la figure 3.57 interroge (message *optical amplifier test*) un ensemble d'objets de type *SLTE agent*, pas n'importe lesquels, ceux vérifiant l'invariant ci-dessus indiqué, *i.e.* ceux supervisant des sections du réseau déjà fortement défaillantes. Nous dérogeons ici à la logique standard d'UML où une ligne de vie ne désigne qu'une et une seule instance alors que nous adressons un ensemble caractérisé par *s'* (c'est un singleton en l'occurrence mais ce n'est qu'une coïncidence !). La remontée des tests des amplificateurs de signaux (*BU module* et *Repeater subsystem* qui héritent d'*Optical amplifier* en figure 3.51) se fait par l'arrivée de l'événement *optical amplifier problem*. L'algorithme se termine par une requête de diagnostic (message *diagnosis*) à un autre ensemble d'objets de type *SLTE agent* caractérisé par le symbole *s''* tout à droite de la figure 3.57. La

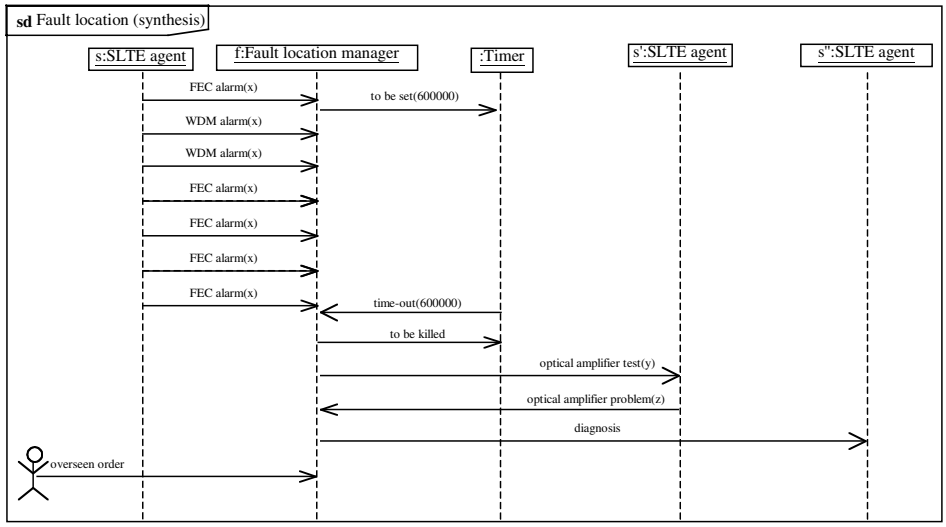


Figure 3.57 – Scénario du cas d'utilisation *Fault location,* synthèse.

valeur de cet ensemble dépend dynamiquement du scénario et plus précisément des attributs (x, y et z) de tout ou partie des événements qui y figurent. En conséquence, s'' est exprimé via des préconditions et post-conditions des événements en figure 3.57 qui sont :

```
context Fault location manager::time-out(600000)
   post: current.strongly failing = current.failing→select(w | w.optical
         ➥ path layer link connection.optical path.oclInState(Malfunctioning))
context SLTE agent::optical amplifier test(y : Set(Optical amplifier))
   pre: y = f.current.strongly failing.support.repeater.repeater
        ➥subsystem→union(f.current.strongly failing.support.end.oclAsType(BU).
        ➥ bu subsystem.bu module)
context Fault location manager::optical amplifier problem(z : Optical amplifier)
   pre: current.strongly failing.support.repeater.repeater
        ➥ subsystem→union(current.strongly failing.support.end.oclAsType(BU).
        ➥ bu subsystem.bu module)→includes(z)
   post: s'' = current.strongly failing→select(w | w.support.repeater.repeater
     ➥ subsystem→union(w.support.end.oclAsType(BU).bu subsystem.bu module
     ➥ →includes(z))→last().slte agent→select(-- un seul, n'importe lequel)
context SLTE agent::diagnosis()
   post: f.active.oclIsNew
context Fault location manager::overseen order()
   post: archive = archive@pre→union(active@pre) and active→isEmpty()
     ➥ and current→isEmpty()
```

L'essentiel, ce qu'il faut retenir

Toutes ces contraintes OCL ne sont pas forcément digestes. Le lecteur y verra peut-être l'argument, voire la preuve, qu'une spécification « plus » formelle qui est la nôtre écarte l'utilisateur « lambda », et que finalement, rien ne vaut un bon

graphique. Le débat est ouvert mais un point important est que cette mini-étude de cas est un cas réel de l'industrie, à méditer donc quant à la pratique récurrente et rigoureuse des *Sequence Diagram* UML : les embûches foisonnent…

Patrons métier

Le modèle statique de la figure 3.58 est un peu en marge du sujet de ce chapitre. Il est intéressant en tant qu'illustration d'un patron métier au sens de Fowler (voir chapitre 1). L'idée de ce patron est d'organiser les modèles de la figure 3.51 et de la figure 3.52 autour du triptyque *Manager/Agent/Network element* de la norme télécoms GDMO.

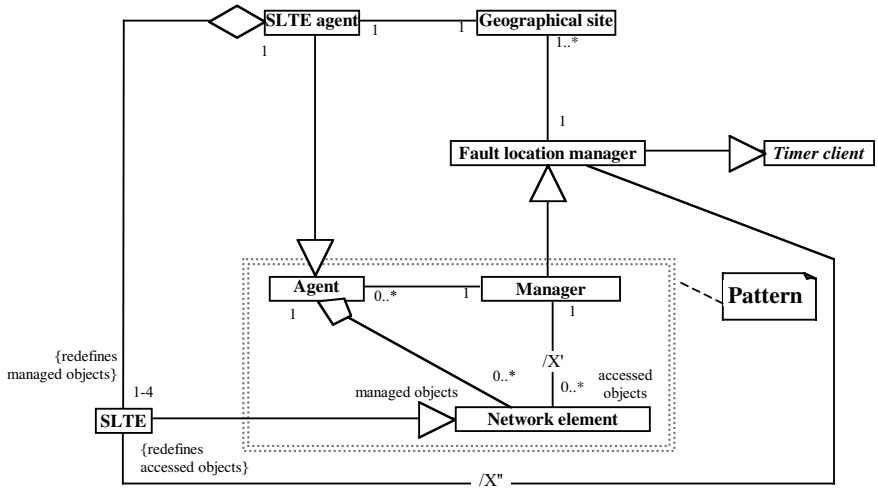


Figure 3.58 – Intégration d'UML et de GDMO sur la base de la réutilisation du patron *Manager/Agent/Network element*.

Des invariants précisant la façon dont les associations X' et X'' sont dérivées, sont nécessaires pour finaliser le modèle de la figure 3.58 :

```
context Manager inv X':
   accessed objects = agent.managed objects
context Fault location manager inv X'':
   slte = geographical site.slte agent.slte
```

3.6 CONCLUSION

Il est difficile de ne pas sortir indemne de la lecture de ce chapitre. L'honnêteté veut que soit présenté ce qu'est exactement UML dans ses versions antérieures puis quelles sont les différences et/ou progrès observables dans 2.x. Nous nous sommes attelés

à cette tâche dans ce chapitre, mais la même honnêteté veut aussi que soient signalées au lecteur toutes les incohérences de la documentation, encore nombreuses aujourd'hui.

À la fin des trois premiers chapitres de cet ouvrage, un professionnel de l'informatique devant décider de l'introduction d'UML dans ses projets et pour ses équipes de développement, pourrait arriver à la conclusion hâtive qu'UML est décidément une « usine à gaz » et qu'il y a donc tous les risques d'échec à son introduction et son usage. Cette opinion doit être tempérée par l'étude des retours d'expérience des chapitres 4, 5 et 6. Nous y tentons de rationaliser la mise en œuvre d'UML tout en continuant certes à soulever des problèmes mais en fournissant au final des applications informatiques « qui tournent ». Il y a donc encore de l'espoir…

Pour le détail des types de diagramme de *Behavior* et leur mise en œuvre, rappelons-nous qu'au tout début du chapitre 2 nous avons parlé pour UML 1.x de modèles essentiels, dérivés et auxiliaires. Nous pensons ainsi que les *State Machine Diagram* d'UML 2.x (*Statechart Diagram* en UML 1.x) restent le cœur de la modélisation dynamique. L'essor pris par les *Activity Diagram* dans UML 2.x fait que l'on ne peut raisonnablement plus les considérer comme des sous-types de *Statechart Diagram* ce qui était explicitement écrit dans la documentation d'UML 1.x. Néanmoins, c'est à notre sens, soit l'un, soit l'autre. En effet, les *State Machine Diagram* et *Activity Diagram* se ressemblent trop, comme souligné dans ce chapitre, pour être mis en œuvre conjointement[1]. Reconnaissons aux *Sequence Diagram* et aux *Activity Diagram* une forte nature didactique : ils sont intelligibles avec beaucoup moins de difficulté que les *State Machine Diagram*. Cette remarque doit servir de réflexion quant à la préoccupation de savoir si l'on utilise UML pour documenter des systèmes logiciels, communiquer entre personnes travaillant sur un même projet, ou radicalement, pour spécifier formellement et générer du code de qualité. Peut-être les deux mais, par expérience, les deux approches sont rarement conciliables même avec l'assistance d'un atelier de génie logiciel. La mode MDA/MDE est à ce titre plutôt cadrée sur la seconde démarche.

Pour les autres types de diagramme, dont notamment les *Use Case Diagram*, *Collaboration Diagram* et *Timing Diagram*, il faut en espérer le minimum. Globalement, ce qui est le plus insatisfaisant et le plus critiquable dans UML, c'est cette approche « marketing » plutôt désagréable qui résulte de l'introduction opportuniste et selon nous à des fins mercantiles, de toutes les notations possibles et imaginables. Les *Activity Diagram* sont une copie des notations d'Objectory/OOSE [8], elles-mêmes largement inspirées de SDL [3]. Certes, le formalisme inhérent aux *Activity Diagram* n'est pas inintéressant car il s'inspire des bonnes veilles méthodes que l'on retrouve aujourd'hui dans les spécifications de *workflow* par exemple. Les adeptes de Merise retrouveront les modèles conceptuels de traitement (MCT), et donc la philosophie des réseaux de Petri, à la UML cependant, c'est-à-dire sans la rigueur originelle de tels outils.

1. L'essai fait dans le chapitre 6 est, en liaison avec cette remarque, à caractère pédagogique.

3.7 BIBLIOGRAPHIE

1. Booch, G. : *Object-Oriented Analysis and Design with Applications*, Second Edition, Benjamin/Cummings (1994)
2. Cook, S., and Daniels, J. : *Designing Object Systems – Object-Oriented Modelling with Syntropy*, Prentice Hall (1994)
3. Ellsberger, J., Hogrefe, D., and Sarma, A. : *SDL – Formal Object-Oriented Language for Communicating Systems*, Prentice Hall (1997)
4. Harel, D. : "Statecharts: A Visual Formalism for Complex Systems", *Science of Computer Programming*, 8, (1987) 231-274
5. Harel, D. : "From Play-In Scenarios to Code: An Achievable Dream", *IEEE Computer*, 34(1), (2001) 53-60
6. Harel, D., and Gery, E. : "Executable Object Modeling with Statecharts", *IEEE Computer*, 30(7), (1997) 31-42
7. International Telecommunication Union : *ITU-T Recommendation Z.120 – Annex B: Formal semantics of Message Sequence Charts* (1998)
8. Jacobson, I., Christerson, M., Jonsson, P., and Övergaard, G. : *Object-Oriented Software Engineering – A Use Case Driven Approach*, Addison-Wesley (1992)
9. Mellor, S. : "Whiter UML? ", *Keynote of Technology of Object-Oriented Languages and Systems Pacific'2000*, Sydney, Australia, November 20-23 (2000)
10. Mellor, S., and Balcer, S. : *Executable UML – A Foundation for Model-Driven Architecture*, Addison-Wesley (2002)
11. Object Management Group : Object Management Architecture Guide (1992)
12. Object Management Group : *OMG Unified Modeling Language Specification*, version 1.3 (1999)
13. Object Management Group : *OMG Unified Modeling Language Specification*, version 1.5 (2003)
14. Object Management Group : *UML 2.0 Superstructure Specification* (2003)
15. Object Management Group : *UML 2.0 OCL Specification* (2003)
16. Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., and Lorensen, W. : *Object-Oriented Modeling and Design*, Prentice Hall (1991)
17. Shlaer, S., and Mellor, S. : *Object Lifecycles – Modeling the World in States*, Prentice Hall (1992)

3.8 WEBOGRAPHIE

Action Semantics Consortium : www.kc.com/as_site/home.html

Alcatel Submarine Networks : www.alcatel.com/submarine/

Guidelines for the Definition of Managed Objects (GDMO) : arthurfoster.com/gdmo-asn1/, www.cellsoft.de/telecom/gdmo.htm

International Telecommunication Union (ITU) : www.itu.int

AGL Rhapsody : www.ilogix.comAGL TAU : www.telelogic.com

4

Prison de Nantes

4.1 INTRODUCTION

L'étude de cas intitulée « prison de Nantes » est extraite de l'ouvrage de Habrias [4]. Dans ce chapitre, nous donnons la spécification IA (*Information Analysis* aussi appelée NIAM, *Nijssen Information Analysis Method*) de cette étude de cas. Nous utilisons son cahier des charges pour appréhender les aspects statiques. Qu'on le veuille ou non, en UML, c'est l'approche entité/relation qui se cache derrière les constructions de modélisation pour les *Class Diagram*. Nous allons donc « faire de l'entité/relation », en suivant la manière dont ce paradigme, universellement connu et reconnu en informatique, est mis en œuvre dans UML.

Par souci de comparaison, nous proposons notre propre version d'une formalisation NIAM de l'étude de cas figurant dans [4]. On trouve en effet dans [4] un modèle NIAM différent du nôtre. Nous offrons aussi et bien entendu le modèle UML complet. L'idée est d'évaluer si UML apporte réellement un plus dans sa réutilisation de l'approche entité/relation. En effet, nombreuses sont les méthodes qui s'appuient sur ce paradigme de modélisation tout en le personnalisant, Merise étant un exemple parmi tant d'autres. Une telle personnalisation est souvent douteuse si elle n'est que syntaxique. En d'autres termes, on obtient un aspect singulier en utilisant une variation particulière de l'approche entité/relation, un *Class Diagram* UML étant ainsi très reconnaissable. Malheureusement, il n'y a pas toujours une avancée sémantique, c'est-à-dire des constructions de modélisation véritablement novatrices. Concrètement, UML est-il source d'apports significatifs à l'approche entité/relation ? Fondamentalement oui, ne serait-ce que par les opérations des entités qui leur confèrent un caractère plus dynamique, un caractère « objet » en l'occurrence. Cela revient à poser le débat plus général des avancées qu'apporte la modélisation objet à l'approche entité/relation. Mais la réponse peut aussi être négative, car comme nous le verrons dans l'étude de cas, on a parfois l'impression d'une régression. L'ouvrage de Habrias cité n'est pas récent, pas plus que ne l'est NIAM. Pourquoi alors, tant

d'années plus tard, se rendre compte que NIAM peut faire aussi bien qu'UML, sinon mieux ? Certes, NIAM n'est pas objet et n'a pas en particulier la prétention de couvrir le spectre que couvre UML. Ainsi NIAM ne peut concurrencer UML que sur les aspects statiques.

D'autres arguments en faveur d'UML sont l'abondance et la variété des types de relations. Malheureusement, l'agrégation et la composition, telles qu'elles sont sémantiquement définies dans UML, ne constituent pas une avancée significative car bon nombre d'approches entité/relation ont déjà intégré la relation tout/partie avec de meilleurs résultats. Les autres relations sont très liées à la conception plutôt qu'à l'analyse : celle de *Realization* par exemple, qui a trait à l'implémentation des types et des interfaces. C'est finalement OCL qui compense le manque de rigueur d'UML et qui va nous permettre d'obtenir un modèle correct pour cette étude de cas.

Alors que la deuxième section donne le cahier des charges, les troisième et quatrième discutent du processus d'ingénierie des besoins, de ses pièges en particulier à travers les fameux sept péchés capitaux de l'analyse. Les quatrième et cinquième sections contiennent le modèle lui-même et quelques variantes intéressantes, en UML et en NIAM, afin de se livrer à une étude comparative. La sixième section s'intéresse aux règles d'implantation dans un système de base de données relationnelle de *Class Diagram* UML. L'objectif poursuivi est de proposer dans la dernière section, une implantation complète et étendue de l'étude de cas sur une architecture distribuée de type *Enterprise JavaBeans* [8].

4.2 CAHIER DES CHARGES

Voici l'interview entre le directeur de la prison (FB) et l'informaticien (SD), prise dans [4].

- FB : à l'arrivée d'un nouveau détenu, l'administration pénitentiaire établit une fiche d'écrou comportant les informations suivantes :
  - un numéro d'écrou qui permet d'identifier un prisonnier à l'intérieur de la prison de Nantes ;
  - le nom, prénom usuel, date de naissance et lieu de naissance du prisonnier ;
  - le numéro d'affaire qui identifie l'affaire parmi les affaires de la juridiction ;
  - le nom de la juridiction d'origine de l'affaire, nom qui identifie parfaitement la juridiction ;
  - la date des faits ;
  - le motif de l'affaire ;
  - et la date d'incarcération.
- SD : qu'entendez-vous par nouveau détenu ? Est-ce qu'une personne qui a quitté « légalement » la prison il y a trois ans et qui y revient est un nouveau détenu ?
- FB : oui.

- SD : il aura donc un nouveau numéro d'écrou ?
- FB : oui. De plus, je dois vous signaler que si nous avons un numéro d'écrou c'est que c'était la seule manière d'identifier parfaitement un prisonnier.
- SD : est-ce que deux prisonniers qui sont actuellement dans une prison française peuvent avoir le même numéro d'écrou ?
- FB : oui, mais pas deux prisonniers de la prison de Nantes.
- SD : est-ce qu'un prisonnier peut arriver à la prison pour plusieurs affaires ?
- FB : il peut être écroué en « préventive ». Dans ce cas, il n'a pas encore été condamné. Mais il subit une décision d'incarcération qui a trait à une affaire. Il a pu être écroué parce qu'il a été condamné pour une affaire. Il peut aussi être écroué parce qu'il a été condamné pour plusieurs affaires.
- SD : est-ce que vous enregistrez toutes les affaires pour lesquelles le prisonnier a été condamné ?
- FB : oui.
- SD : vous enregistrez aussi les affaires pour lesquelles il peut être condamné après son incarcération ?
- FB : oui, tant qu'il est incarcéré à la prison de Nantes.
- SD : un détenu peut même faire l'objet de nouvelles condamnations pour de nouvelles affaires ?
- FB : oui, une personne ayant participé à une affaire, condamnée à la prison (et incarcérée à Nantes) pour cette affaire peut être de nouveau condamnée pour une autre affaire à laquelle elle a participé... avant son incarcération. Depuis que je dirige cette prison, on ne s'en évade plus !
- SD : pour une affaire donnée, pouvez-vous avoir plusieurs prisonniers ?
- FB : oui, mais nous n'y tenons pas. Il faut aussi que je vous dise que l'on complète les fiches d'écrou par toutes les décisions concernant l'incarcération, à savoir :

  – les condamnations ;
  – les réductions de peine ;
  – les libérations définitives.

  Chacune de ces décisions est notée sur la fiche d'écrou avec leur numéro (1, 2 ou 3). Chaque décision a une date. Les condamnations comportent la durée de l'emprisonnement à exécuter indiquée en nombre de jours, les réductions de peine comportent la durée de la réduction et les libérations, la date de libération.
- SD : un détenu peut-il avoir, par exemple, plusieurs réductions de peine ?
- FB : oui.
- SD : vous m'avez dit que vous notiez le motif de l'affaire. Il s'agit du motif pour lequel le détenu a été condamné pour l'affaire qui l'a conduit en prison ?

- FB : oui, en effet, on peut avoir pour une même affaire des individus condamnés pour des motifs différents.
- SD : pouvez-vous me donner des exemples de motifs ?
- FB : oui, on a par exemple :

  – 01-vols et délits assimilés ;
  – 02-coups et blessures ;
  – 03-escroquerie ;
  – 04-port d'armes prohibé ;
  – 05-conduite en état d'ivresse ;
  – 12-abus de confiance ;
  – 14-homicide ;
  – 15-proxénétisme, etc.

- SD : pour un détenu et une affaire, il peut sans doute y avoir plusieurs motifs de condamnation. Vous en notez un seul ?
- FB : oui, le principal.
- SD : vous m'aviez dit aussi que vous enregistriez la date des faits. Mais il se peut que pour une affaire, il y ait des dates des faits. Par exemple, une escroquerie se déroule de telle date à telle date. Pour une affaire, y a-t-il alors plusieurs dates des faits ?
- FB : on enregistre qu'une date des faits.
- SD : pouvez-vous me donner quelques exemples de « dates des faits » ?
- FB : oui, par exemple : « au cours des mois d'avril et mai 1996 », « vers le 6.12.93 ».
- SD : j'aimerais revenir sur les décisions concernant l'incarcération. Est-ce que plusieurs décisions (par exemple, réductions de peine) peuvent être prises à la même date pour le même détenu ?
- FB : non. Je comprends que vous pensez que pour un détenu, à la même date, par exemple deux réductions de peine de même durée peuvent être décidées, chaque décision étant relative à une affaire différente pour laquelle le détenu est en prison. Vous voulez dire : est-ce que deux décisions de même type concernant le même prisonnier peuvent être prises le même jour ?
- SD : oui.
- FB : la réponse est non. Mais on peut décider le même jour d'une réduction et d'une condamnation à des jours de prison. C'est rare.
- SD : voulez-vous enregistrer le fait que telle décision concernant l'incarcération de tel prisonnier a été prise par telle juridiction ?
- FB : non. Pour nous peu importe de savoir quelles sont les juridictions qui ont pris ces décisions et pour quelles affaires.
- SD : pour l'affaire principale pour laquelle le prisonnier est incarcéré, n'y a-t-il pas une juridiction d'origine ?
- FB : oui.

- SD : au sujet des juridictions, est-ce qu'il peut y avoir deux juridictions ayant le même nom ?
- FB : non.
- SD : dans vos fichiers conservez-vous les informations sur les détenus qui ont quitté « légalement » la prison ?
- FB : non. On ne conserve que les informations relatives aux détenus qui sont en train de purger leur peine de détention.
- SD : y compris ceux qui sont en préventive ?
- FB : oui, bien sûr.
- SD : OK merci, je vais faire le modèle UML.

4.3 PRÉANALYSE

L'interview qui précède est extrêmement riche d'un point de vue didactique car elle est truffée de pièges. Avant d'en venir au modèle UML final, ainsi qu'à des variantes possibles et acceptables, nous allons reproduire la démarche intellectuelle de modélisation. Le résultat final est obtenu après plusieurs itérations. Il est à ce titre intéressant de voir où sont les embûches dans le texte et comment les déjouer.

4.3.1 Les sept péchés capitaux de l'analyse

Les sept péchés capitaux de l'analyse sont vieux mais pas autant que les sept péchés bibliques. Ils datent du début de l'ère informatique et sont : Silence, Bruit, Contradiction, Ambiguïté, Référence en avant, Vœu pieux et Surspécification.

Dans l'interview, le directeur de la prison ne facilite pas le travail de compréhension de l'analyste. Certaines choses n'ont pas été dites et comme nous allons le voir dans la section 4.4 de ce chapitre, l'analyste va compenser ces silences par des « vérités » bien à lui. Dans la réalité, les silences constituent environ 25 % du total du parasitage de l'analyse. Tout analyste expérimenté sait qu'il faut mener une véritable investigation de l'information et ne pas se limiter à celle fournie. Ces silences sont par ailleurs le plus souvent et malheureusement, détectés à la livraison des applications lorsque la recette fait apparaître des fonctionnalités manquantes, d'où la nécessité d'anticiper. Jusqu'à quel point est toute la question, pour ne pas tomber dans l'excès inverse : dénaturer les besoins initiaux du client. À la suite de l'interview, l'élaboration par l'analyste d'un questionnaire contradictoire et/ou la validation des premiers modèles UML créés serait déjà un bon moyen de lever les silences. Néanmoins, limiter le nombre des silences reste un problème non technique et est sujet aux aléas de la nature humaine : ce péché capital est vraiment source de tous les maux ! Comment spécifier ce qui n'a pas été dit ou écrit ?

Le bruit a lui une fréquence d'apparition moindre d'environ 20 %. Dans le texte de l'interview, le directeur de la prison dit : « Depuis que je dirige cette prison, on

ne s'en évade plus ! » Petit défi : vous qui êtes analyste, essayez d'implanter dans votre modèle UML cette idée-là ! Ici, l'analyste est dérangé dans sa réflexion par un élément « préoccupant », de la bouche même du demandeur. Néanmoins, cela n'a aucune incidence sur le système d'information que l'analyste a à mettre en place. Un bruit est donc un élément d'information qui brouille la compréhension de la situation et qui n'a finalement aucune valeur ajoutée. Face à la complexité des besoins, au travers de leur nombre notamment, l'écueil est que l'on ne se rend pas toujours compte de l'intérêt d'écarter un bruit durant la phase d'analyse, d'où le risque de s'en encombrer dans la suite, au détriment de la productivité.

La contradiction, avec 10 % de taux d'apparition est *a priori* plus facilement perceptible qu'un bruit dans un discours. Dans l'interview, les deux personnes discutent ainsi : « FB : (…) le motif de l'affaire (…) SD : vous m'avez dit que vous notiez le motif de l'affaire. Il s'agit du motif pour lequel le détenu a été condamné pour l'affaire qui l'a conduit en prison ? FB : oui, en effet, on peut avoir pour une même affaire des individus condamnés pour des motifs différents. » Ainsi il semble qu'il n'y ait dans l'absolu qu'un motif par affaire puis plusieurs motifs par affaire, puisque pour une même affaire, des détenus impliqués sont condamnés pour des motifs différents. Le directeur de la prison se contredit donc. L'analyste doit-il lier le motif à l'affaire ou au détenu, voire ni à l'un ni à l'autre si le système d'information ne mémorise que le motif de l'affaire principale d'un détenu ? C'est ce dernier choix qui a été fait dans le modèle solution. De manière générale, face à des contradictions identifiées par un analyste, une personne exprimant des besoins peut aisément trancher, à condition que ces contradictions sont reformulées par l'analyste lui-même : « Vous aviez dit A, vous aviez dit B qui contredit A, choisissez maintenant A ou B. » On peut considérer qu'un analyste détecte plus facilement et au plus tôt les contradictions, par le fait qu'elles sont plutôt émergentes dans un texte ou dans une conversation.

L'ambiguïté, avec 10 % de taux d'apparition, est du même acabit que la contradiction au sens où il s'agit d'une « perturbation » que l'analyste s'attend naturellement à combattre dans un discours. Voyons l'exemple : « FB : (…) Mais il subit une décision d'incarcération qui a trait à une affaire (…) FB : (…) Il faut aussi que je vous dise que l'on complète les fiches d'écrou par toutes les décisions concernant l'incarcération (…) ». Dans ce texte, il faut comprendre qu'une décision d'incarcération est différente d'une décision concernant l'incarcération. Pas facile, n'est-ce pas ? Relisez pourtant bien le texte. Une décision d'incarcération est associée à une date, celle où l'on décide d'incarcérer le prisonnier, alors que les décisions concernant l'incarcération sont les réductions de peine, les libérations définitives et les condamnations prises à l'encontre du prisonnier durant sa présence à la prison. L'ambiguïté semble donc plus insidieuse que la contradiction car dans l'exemple, l'analyste peut évidemment ne pas voir la différence entre les deux sens donnés au mot « décision ».

La référence en avant, quant à elle (5 %), est à rapprocher du bruit. Elle n'est pas source d'erreur mais de gêne dans la démarche intellectuelle de l'analyste. Alors que celui-ci isole un sous-problème important, le demandeur fait dériver la conversation vers un autre sous-problème. Certes, chaque sous-problème a son importance mais

l'analyste souhaiterait guider fermement la discussion de manière à gérer la complexité. Le demandeur au contraire, disposant d'une vision globale, entraîne l'analyste sur tous les terrains en même temps. Voici : « SD : est-ce qu'un prisonnier peut arriver à la prison pour plusieurs affaires ? FB : il peut être écroué en « préventive ». Dans ce cas, il n'a pas encore été condamné (…) ». Le problème de l'affaire qui fait qu'un détenu est incarcéré, est un problème épineux en soi. Interrogé sur le sujet, le directeur fait référence au problème de « préventive », certes primordial, mais momentanément inopportun dans l'esprit et dans la question de l'analyste. Le directeur de la prison répond donc « à côté » mais les informations qu'il donne revêtent une grande importance pour la fabrication du modèle, d'où la nécessité pour l'analyste de tout noter au risque de ne traiter qu'en partie le problème posé.

Avant-dernier péché, le vœu pieux (20 %) est révélateur de la problématique de capture des besoins. L'analyste qui, par expérience, sait que le système d'information qu'il construit ne sera jamais l'image artificielle dans un ordinateur de la situation qu'on lui décrit mais beaucoup plus (lire pour plus de détails, l'excellent, et toujours d'actualité, ouvrage de Tabourier [10]), doit-il alors créer/inventer les besoins ? En d'autres termes, l'existence même du système d'information qu'il construit modifiera l'organisation ce qui entraînera d'autres besoins et donc un autre système d'information. Pourquoi alors, dès le départ, ne pas considérer ce système d'information comme un système évolutif, dont les structures ne seront pas figées mais adaptatives. Cela suppose que toute affirmation en analyse n'est en rien immuable dès que l'exploitation du système d'information montrera l'intérêt d'aller plus loin en fonctionnalités. Concernant le vœu pieux, le demandeur oscille entre un besoin à prendre en charge et une perspective d'amélioration de l'organisation et de ses règles de gestion, perspective qui peut paraître illusoire mais dont l'analyste, par extrapolation, peut juger réaliste. Dans l'exemple qui suit, l'organisation « rêvée » par le directeur de la prison est justement plutôt irréaliste : « SD : pour une affaire donnée, pouvez-vous avoir plusieurs prisonniers ? FB : oui, mais nous n'y tenons pas (…) ». Il n'en résulte pas vraiment d'alternative de modélisation pour l'analyste sinon que de revoir en profondeur la notion même d'affaire : quelque chose qui n'est relatif qu'à un prisonnier dans le vœu pieux du directeur. Au-delà de cette simple illustration, la modélisation objet est peut-être un outil adéquat pour créer des modèles plus adaptatifs aux fluctuations des besoins que ne le permet l'approche entité/relation.

Pour en terminer, la surspécification (10 %) est un phénomène à l'origine de la redondance dans les modèles. Spécifier deux fois ou plus le même besoin, c'est l'implanter plusieurs fois et donc gaspiller du temps et de l'argent. Voici un exemple dans l'échange entre l'analyste et le directeur de la prison : « FB : il peut être écroué en "préventive". Dans ce cas, il n'a pas encore été condamné. » La notion de préventive revient de manière récurrente dans le discours du directeur. Ceci étant, nous montrerons qu'il n'est nullement nécessaire de représenter ce concept par un type d'objet. Dans le cas contraire, l'analyste crée de la redondance dans son modèle. Là où parfois cette redondance peut s'avérer nécessaire dans le processus de compréhension/validation (voir la notion d'élément de modèle « dérivé » dans le chapitre 2), en d'autres endroits, elle peut compliquer les modèles ne serait-ce que par l'accrois-

sement injustifié du nombre de types d'objet s'y trouvant. De manière plus générale, la « minimalité » d'un modèle objet est à notre sens, source de simplicité et donc favorise sa compréhension. La surspécification n'est donc intéressante que si elle est parfaitement maîtrisée. Dans la section 4.4.1, nous donnons justement la manière dont la notion de préventive peut être tout à fait émergente dans un modèle sans induire une quelconque redondance nuisible.

4.3.2 Cueillette des objets, ingénierie des besoins

L'éternel problème présidant à la construction d'un modèle est de justifier de la provenance des objets. Les explications qui suivent ne sont pas une démonstration mais une sensibilisation à la démarche de « cueillette des objets », selon l'expression maintenant restée célèbre et que l'on peut attribuer à Meyer. Nous reviendrons sur ce problème d'ingénierie des besoins dans le cadre d'une étude de cas de domotique (chapitre suivant) pour l'illustration des aspects dynamiques. En effet, la réflexion est plus intéressante car fondée sur l'analyse des comportements et des interactions et pas seulement sur l'étude des dépendances structurelles qu'entretiennent les objets entre eux, comme ici pour le cas de la prison de Nantes.

Revenant au contexte de la prison de Nantes, le type d'objet *Fiche écrou* mis en exergue par le directeur de la prison au début de l'interview s'impose naturellement. En figure 4.1, ce type comporte la grande majorité des informations relatives à un détenu.L'esprit de la démarche est ici par réfutation/correction, c'est-à-dire dessiner des types d'objet, les caractériser pour l'essentiel par leurs attributs et leurs relations puis les amender voire les supprimer au fur et à mesure que la connaissance du système s'affirme pour l'analyste. Rien de bien nouveau, cette approche étant décrite dans presque tous les ouvrages sur le sujet. Malgré son caractère empirique, elle est le plus souvent efficace même si, pour une même étude de cas traitée dans deux ouvrages majeurs de la modélisation orientée objet ([7] et [9]), on trouve des dissymétries assez remarquables dans les modèles. En clair, on ne peut pas parler de méthode, au sens d'une démarche codifiée et reproductible, car cela supposerait que différents analystes traitant du même problème obtiennent la même solution en appliquant la méthode. Le lecteur attentif fera le lien avec la technique des *Use Case* discutée au chapitre 3 (sans objet pour cette étude de cas de la prison de Nantes) censée prendre en charge la problématique d'ingénierie objet des besoins, ce qui est loin d'être le cas, les *Use Case* ne couvrant pas toutes les attentes en la matière.

En fait, *Fiche écrou* est déjà un élément logique et informationnel dans le système de gestion de la prison de Nantes d'où son inadéquation pour un *nouveau* système. L'adjectif « nouveau » est ici important car il sous-entend que la façon dont l'information existe est dépendante de l'organisation. La façon dont elle est « formatée » (au sens large) et est utilisée, est une chose dont il faut absolument s'abstraire. Par conséquent, dans la figure 4.2, le type d'objet *Détenu* annule et remplace partiellement l'idée de *Fiche écrou*, idée finalement éphémère au contraire du concept de *Détenu* plus stable, mieux à même de franchir les étapes successives d'évolution du système d'information. *Prisonnier*, parce qu'il concurrence et crée de la redondance

Figure 4.1 – Processus d'élaboration du *Class Diagram* (phase 1).

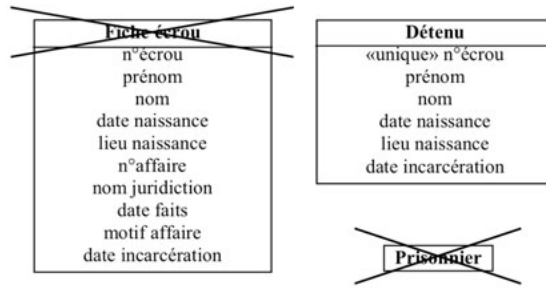


Figure 4.2 – Processus d'élaboration du *Class Diagram* (phase 2).

avec le type d'objet *Détenu*, s'efface aussi. Finalement, *n°écrou* est adjoint du stéréotype *«unique»* pour bien faire apparaître que cette propriété sert d'identifiant.

Ce procédé d'apprentissage des connaissances liées au système de gestion de la prison de Nantes a tout à gagner s'il est mené en conjonction avec les acteurs de l'organisation. Ainsi, dans la figure 4.3, la caractérisation précise du concept d'affaire nécessite de savoir, d'un point de vue légal, que les affaires s'identifient précisément par leur numéro et par la juridiction, unique, dont elles dépendent. La spécification du type d'objet *Affaire* est donc cruciale car une erreur supposerait la non-conformité du système défini à la législation en vigueur.

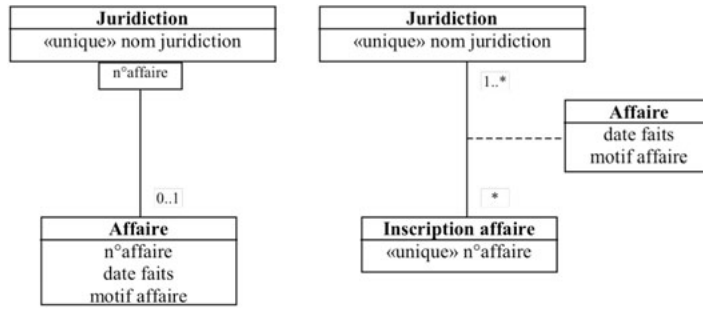


Figure 4.3 – Processus d'élaboration du *Class Diagram* (phase 3).

Dans le modèle de la figure 4.3, à droite, le type *Inscription affaire* est purement artificiel. Il sert à porter l'attribut *n°affaire* pour créer une association avec le type *Juridiction* puis voir le couple formé comme une association classe de manière à obtenir le type *Affaire*. Ce procédé assure qu'*Affaire* a aussi pour identifiant la concaténation de *nom juridiction* et *n°affaire*. À gauche, l'utilisation d'un outil comme le *qualifier* d'UML est plus intuitive, pour s'assurer du même résultat : *Affaire* a pour identifiant la concaténation de *nom juridiction* et *n°affaire*. Le fragment de modèle (à gauche) étant retenu aux dépens de l'autre, il devient intéressant de placer le type d'objet *Motif* (figure 4.4). Question : combien y a-t-il d'instances de *Motif* évoquées dans le cahier des charges ? Réponse : huit et plus *a priori* compte tenu de la liste décrite. Derrière cette question anodine et la réponse qui s'ensuit, il apparaît qu'un motif est caractérisé en fonction, et uniquement en fonction, de l'information *n°motif*, ou éventuellement de l'information *libellé motif* (les deux propriétés sont associées au stéréotype *«unique»* en figure 4.4). Une autre façon de juger de sa pertinence et donc de son incorporation dans le modèle est de le relier à *Détenu* ou à *Affaire* (voir discussion préalable sur la contradiction en tant que péché capital). Deux ébauches de modèle sont en concurrence en figure 4.4. Elles témoignent de la difficulté à augmenter la stabilité du modèle ou à l'autre extrême, à le casser car un nouveau concept arrivant peut tout chambouler (voir la section 4.4 pour finalement se rendre compte qu'aucune des deux solutions en figure 4.4 ne sera retenue).

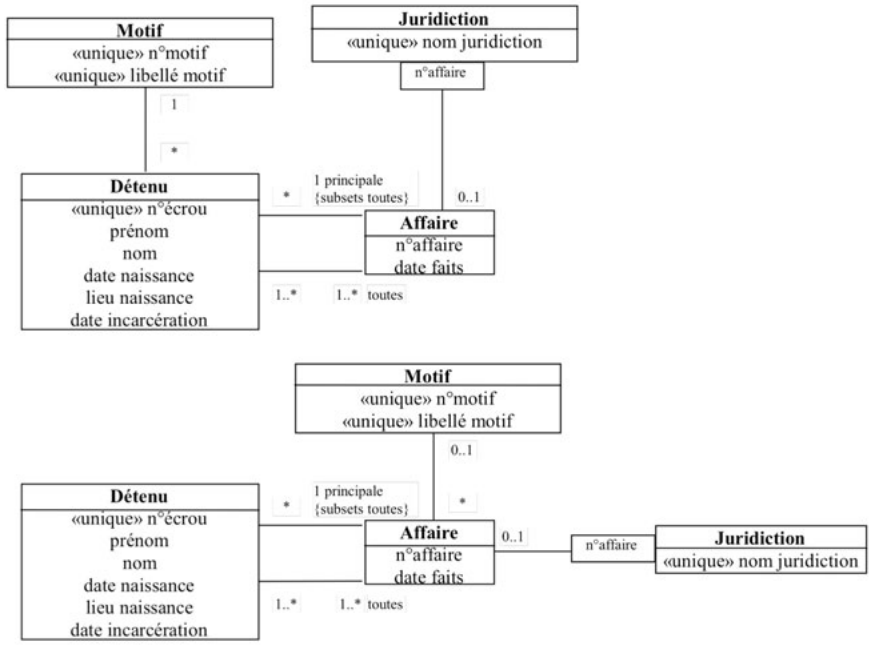


Figure 4.4 – Processus d'élaboration du *Class Diagram* (phase 4).

Dans la figure 4.5, les choses s'affinent pour aboutir par exemple à l'éviction du type d'objet *Préventive* pour la bonne et simple raison qu'un détenu ayant subi zéro condamnation (borne basse de la cardinalité * vers *Condamnation* en figure 4.5) n'est jamais qu'en préventive. Cas intéressant s'il en est de surspécification possible. Autre problème en figure 4.5, la non-factorisation du modèle relative au fait que *Condamnation*, *Réduction peine* et *Libération définitive* partagent l'essentiel de leurs propriétés : la même relation sémantique avec *Détenu* surtout. L'approche objet est sur ce point originale, au sens où elle a pour objectif majeur la qualité des modèles au sens du génie logiciel, ce qui se traduit ici par l'accroissement de la réutilisabilité autour d'un type d'objet *Décision*, qui factorise *Condamnation*, *Réduction peine* et *Libération définitive* (figure 4.6).

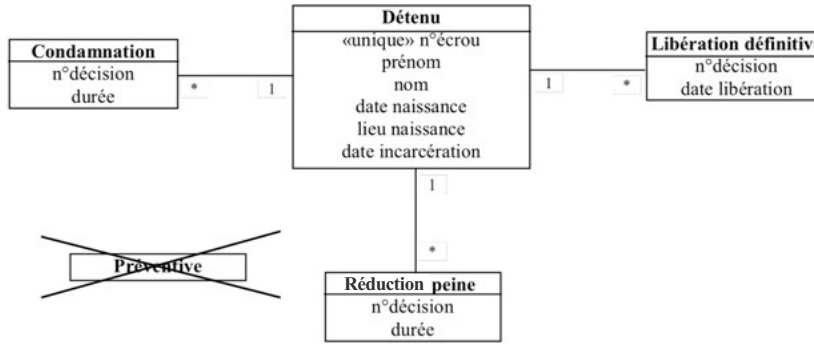


Figure 4.5 – Processus d'élaboration du *Class Diagram* (phase 5).

Force est de constater que la démarche d'ingénierie des besoins est finalement plutôt exploratoire. Chaque nouveau type d'objet identifié amène à reconsidérer les autres, phénomène s'amplifiant lorsqu'il y a des modèles dynamiques qui remettent eux-mêmes en cause les modèles statiques. La démarche n'est donc pas incrémentale au sens où tout nouveau « bloc » dans le modèle peut influer sur le contenu de ceux précédemment établis. En outre, il faut être conscient que ces propagations de modification prennent de plus en plus d'amplitude dans les phases ultérieures du développement (conception et implémentation), d'où la difficulté de mener conjointement création et évolution.

4.4 MODÈLE

Après mûres réflexions, le modèle converge vers un état quasi stable présenté en figure 4.6. Les points clés sont :

- l'isolement de l'affaire principale d'un détenu parmi toutes ses affaires (contrainte d'inclusion *{subsets <property name>}*) permettant de proprement ressortir le type *Incarcération* (comme une association classe en l'occurrence)

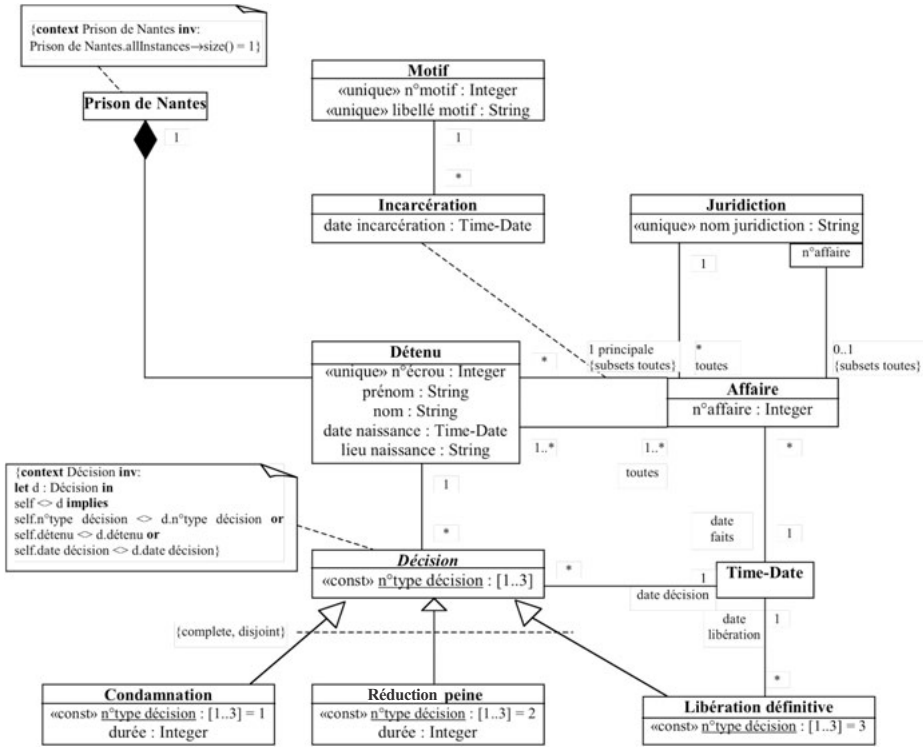


Figure 4.6 – *Class Diagram* du cas de la prison de Nantes.

de manière à porter la propriété *date d'incarcération* et dire que le motif n'est connu que pour l'affaire principale d'un détenu, *i.e.* l'affaire qui l'a fait incarcérer ;

- la création d'une classe abstraite *Décision* (en italique) ayant un attribut de classe (souligné en UML). Attention au texte qui évoque des numéros de décision alors qu'en fait le modèle lève l'ambiguïté en mentionnant bien *n°type décision* et un domaine de valeur compris entre 1 et 3. L'avantage de cette classe abstraite est en particulier de synthétiser par une contrainte OCL, le fait qu'une décision de même type pour un détenu ne peut pas se produire le même jour (note reliée à *Décision*) ;
- le type *Time-Date* est dans cet exemple utilisé parfois comme un *value type*, *i.e.* un type primitif (voir *Incarcération* ainsi que le chapitre 2 où *Time-Date* est partiellement défini) ou comme un type à part entière : associations allant sur le type *Time-Date* et offrant les rôles *date faits*, *date décision* ainsi que *date libération* ;
- le type *Prison de Nantes* est là comme point d'entrée du système (une seule instance de ce type existe en fait : voir note en haut à gauche). Ce genre d'objet est aussi appelé singleton en *Design Patterns* [3]. La composition (losange noir) signifie de manière informelle que l'existence de tout le système d'information

est sujette à l'existence même de cette instance de *Prison de Nantes*. Notons néanmoins le caractère plutôt documentaire de ce type *Prison de Nantes* que sa nécessité absolue pour l'implantation ;

- les types concrets de décision à savoir *Condamnation*, *Réduction peine* et *Libération définitive* héritent directement de *Décision*. La contrainte *{complete, disjoint}* dit que la réunion des instances des trois sous-types est égale à l'ensemble des instances de *Décision* (*i.e.* il n'y a pas d'autres sous-types de décision). De plus, l'intersection des trois est vide, mais si l'on veut être plus précis, il faudrait spécifier que les trois intersections deux à deux sont vides et donc ajouter, en sus et trois fois, le label *{incomplete, disjoint}* sur chaque paire de liens d'héritage. Nous ne l'avons délibérément pas fait pour ne pas surcharger le modèle.

Le modèle en figure 4.6 n'est pas l'unique solution mais il faut prendre garde aux optimisations douteuses, de type court-circuit. Par exemple, *Motif* peut être directement associé à *Détenu* puisque finalement, il y a bijection entre les instances d'*Incarcération* et les instances de *Détenu*. Cette simplification abaisserait cependant la compréhensibilité du modèle par rapport au texte du cahier des charges. Autre exemple : supprimer *Décision* est possible (voir figure 4.5) mais cela diminuerait l'évolutivité et la réutilisabilité du modèle. Il suffit pour cela d'imaginer l'accroissement des sortes de décision et par voie de conséquence la duplication de la contrainte OCL (en commentaire en figure 4.6) pour chacune ainsi que la duplication de la relation avec *Détenu*.

4.4.1 Modèle, variations

Parmi les variantes de modélisation, il existe des ajustements simples comme l'usage de la contrainte *{xor}* par exemple en remplacement de *{subsets <property name>}*. En figure 4.6, la contrainte *{subsets toutes}* signifie précisément qu'étant donné une instance *d* de *Détenu*, l'ensemble singleton constitué de l'affaire principale de *d* (navigation nommée *principale*) est inclus dans l'ensemble de toutes les affaires où *d* est impliqué (navigation nommée *toutes*). Différemment en figure 4.7, l'ensemble singleton constitué de l'affaire principale de *d* est disjoint de l'ensemble constitué des « autres » (on a renommé la navigation *toutes* par *autres*) affaires dans lesquelles *d* est impliqué. En figure 4.7, la disjonction que l'on souhaite décrire ne porte pas sur des sous-ensembles du produit cartésien *Détenu x Affaire* mais sur des sous-ensembles de l'ensemble de toutes les instances de *Affaire*. La notation UML 2.x est ambiguë sur le sujet. On triche un peu en plaçant le trait pointillé le plus à droite possible, *i.e.* vers *Affaire*, mais il manque de notation formelle. En Syntropy [2], ce problème était résolu par le fait que *{xor}* seule concerne des sous-ensembles du produit cartésien des deux types reliés par les deux associations. Dans le cas où cela concerne des disjonctions entre sous-ensembles d'un même type, *Affaire* par exemple, on utilise alors la notation *{Détenu::xor}* signifiant : « étant donné un détenu, alors ce sous-ensemble d'affaires est disjoint de cet autre sous-ensemble d'affaires ». En clair, dans la figure 4.7, on dit que l'affaire principale d'un prisonnier est un ensemble singleton disjoint

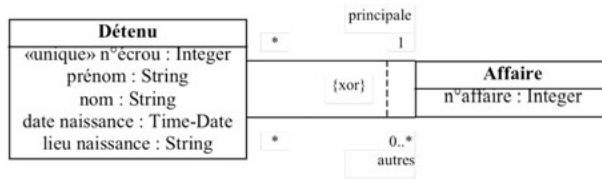


Figure 4.7 – Alternative à la contrainte {*subsets <property name>*} en figure 4.6, la contrainte {*xor*}.

des autres affaires où ce même prisonnier est impliqué, cet ensemble pouvant être vide, contrairement à la navigation *toutes* en figure 4.6 où la cardinalité est *1..** et non *0..**. Au-delà, ce genre de disjonction est une construction de modélisation typique de NIAM (contrainte d'exclusion entre rôles). On a donc parfois l'impression de régression car en NIAM, ce genre de problème se traite tout à fait aisément.

La figure 4.8 fait appel à des choses plus subtiles, la contrainte *{redefines <property name>}* en l'occurrence, apparue seulement dans la version 2 d'UML et fort utile en général (voir chapitre 2). L'idée est de dire qu'une relation héritée fait l'objet d'une redéfinition, le plus souvent pour ce qui concerne les cardinalités. Dans l'exemple de la figure 4.8, le but n'est que de préciser une contrainte supplémentaire qui s'applique sur la relation au niveau du sous-type alors que cette contrainte est sans objet au niveau du supertype. On procède tout d'abord par une redéfinition de l'association entre *Détenu* et *Décision* : *{redefines décision}* signifie donc que l'association entre *Détenu* et *Réduction peine* n'est pas une nouvelle association mais celle héritée avec un nouveau nom de rôle : *toutes* (la cardinalité * ne bouge pas). Ensuite, une association qualifiée est mise en œuvre entre *Détenu* et *Réduction peine*. Le résultat est intéressant en termes de pouvoir d'expression car la contrainte OCL en note en figure 4.6 (celle qui s'applique au type *Décision*) devient caduque. L'interprétation du modèle de la figure 4.8 donne : étant donné une instance de *Détenu* et une instance de *Time-Date* (*qualifier* nommé *date décision*), on a alors ou non (*i.e.* pas nécessairement car l'instance de *Time-Date* en entrée est arbitraire) un lien sur une instance de type *Réduction peine*. Autrement dit, une décision de type *Réduction peine* pour un détenu ne peut pas se produire le même jour, ce qui correspond au cahier des charges. Si l'on fait pareil pour *Condamnation* et *Libération définitive*, on en revient à la contrainte OCL sur le type *Décision* de la figure 4.6.

Pour terminer, nous introduisons ici une variante de modélisation basée sur le concept de *state type* de Syntropy brièvement évoqué au chapitre 3 (ce concept est y appelé *state class*). Évincé d'UML comme concept de premier plan, il n'est pas exclu de l'utiliser à bon escient pour justement ici, « régénérer » l'idée de préventive. Nous avions montré que la préventive était plus un statut du détenu qu'un type d'objet à faire apparaître dans le modèle solution, essentiellement par le fait que cette information était déductible. La figure 4.9 a l'avantage de présenter la notion de préventive (/ pour signaler que le type d'objet est dérivé, *i.e.* calculé) tout en maîtrisant la redondance. La contrainte OCL exprime l'invariant de classe en disant qu'un détenu en préventive n'est jamais qu'un détenu (héritage) avec la restriction qu'il n'a subi aucune condamnation.

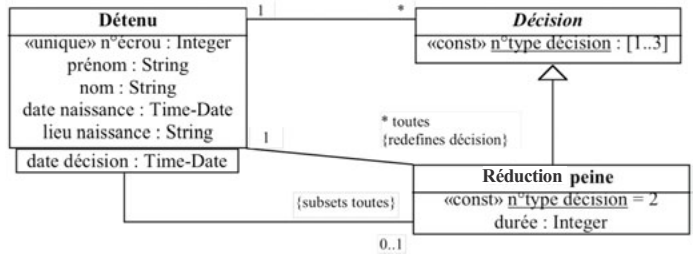


Figure 4.8 – Extension/redéfinition des relations structurelles à l'aide de la contrainte {*redefines <property name>*}.

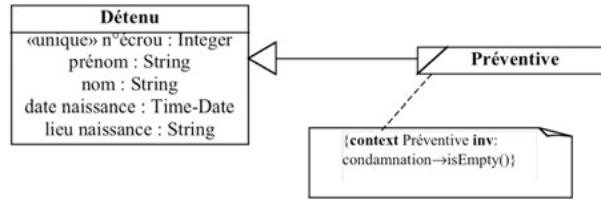


Figure 4.9 – Utilisation du concept de *state type*.

En Syntropy, rappelons que la formalisation de la figure 4.9 suppose l'existence d'un état *Préventive* dans le *State Machine Diagram* de *Détenu*. Rappelons aussi et encore que cette figure n'est pas de la notation UML.

4.4.2 Traitements

Les traitements relatifs au cas de la prison de Nantes sont l'exploitation des informations contenues dans la base de données implantant le modèle de la figure 4.6 : requêtes d'extraction d'information, calculs éventuels (statistiques par exemple) ainsi que globalement ajout, suppression et modification d'information. Les types de modèle UML comme les *Sequence Diagram*, *Activity Diagram* ou encore *State Machine Diagram*, pour ne citer qu'eux, sont ici surdimensionnés par rapport aux besoins, si ceux-ci sont de spécifier la manière dont le modèle de la figure 4.6 est exploité « à l'exécution ». Par exemple, si une préoccupation est de connaître toutes les affaires d'un détenu, il suffit de l'écrire en OCL, ou ici en l'occurrence plus simplement, de repérer la navigation adéquate : rôle *toutes* vers *Affaire* en partant de *Détenu*. Pour plus de clarté, cette propriété peut être typée : *toutes : Set(Affaire)*.

En procédant de la sorte, il est facile jusqu'à un certain stade de complexité de se limiter à de simples expressions pour écrire les traitements demandés par le client. Voici quelques exemples d'expressions OCL de manipulation des modèles de la figure 4.6, la figure 4.7 et la figure 4.8 :

```
context d : Détenu inv:
   d.toutes.n°affaire -- retourne un bag de type Bag(Integer)
```

```
context Détenu inv:
   toutes.n°affaire→asSet() -- retourne un set de type Set(Integer)
context Détenu inv:
   let t : Time-Date in réduction peine[t] -- dans la figure 4.8, retourne un
objet de type Réduction peine ou un Set vide ou un singleton de type
Set(Réduction peine). La différence entre objet et Set s'opère en fonction du
reste de l'expression (voir doc. OCL)
context Incarcération inv:
   self.détenu→size() = 1
context Affaire inv:
   self.incarcération→isEmpty() implies … -- assertion éventuelle ici
context Juridiction inv:
   self.oclIsTypeOf(Juridiction) = true
context Condamnation inv:
   self.oclIsTypeOf(Décision) = false
context Condamnation inv:
   self.oclIsKindOf(Décision) = true
context Détenu inv:
   Détenu.allInstances→forAll(d1,d2 | d1 <> d2 implies d1.n°écrou <>
      ➥ d2.n°écrou) -- sens formel du stéréotype «unique»
context Détenu inv:
   Détenu.allInstances→isUnique(n°écrou) = true -- autre moyen de spécifier
                                                --le stéréotype «unique»
context Détenu inv:
   oclInState(Préventive) implies … -- assertion éventuelle ici
```

4.5 NIAM

Le modèle NIAM exposé en figure 4.10 et en figure 4.11 est équivalent au modèle UML de la figure 4.6. Quelle conclusion en tirer ? UML n'est pas un langage de modélisation révolutionnaire et extraordinaire même s'il a le mérite d'être le fruit d'un consensus international : un standard. Il y a vingt ans environ, on faisait autant et aussi bien. La progression est donc minime. Nous avons choisi NIAM comme référentiel de comparaison mais tout autre support aurait pu montrer qu'UML reste redevable de ses prédécesseurs ou plus volontiers de tous les travaux scientifiques sur l'approche entité/relation.

La version moderne de NIAM est ORM (*Object Role Modeling*) adoptée par l'éditeur de logiciel Microsoft en particulier. L'article [5] fait aussi une comparaison avec UML et arrive à des critiques semblables aux nôtres. Puisque nous critiquons UML, critiquons aussi NIAM qui montre également des lacunes en matière de modélisation.

Sur la forme, l'aspect général de la figure 4.10 et de la figure 4.11 est loin de favoriser la compréhensibilité. Cela est essentiellement dû aux LOT qui correspondent aux attributs dans UML et qui sont dessinés chacun dans des bulles autonomes (contour en pointillé). Sur le fond, la figure 4.11 est révélatrice d'autres problèmes liés à la dichotomie LOT/NoLOT : l'information *durée* est dupliquée via deux LOT (en bas, tout à gauche de la figure 4.11) pour syntaxiquement distinguer la durée d'une

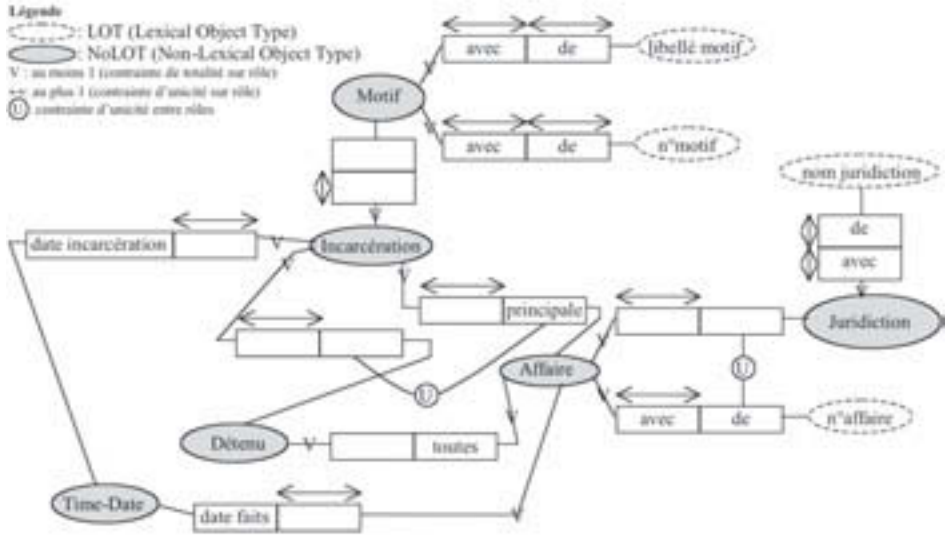


Figure 4.10 – Modèle NIAM (première partie) équivalent au modèle UML de la figure 4.6.

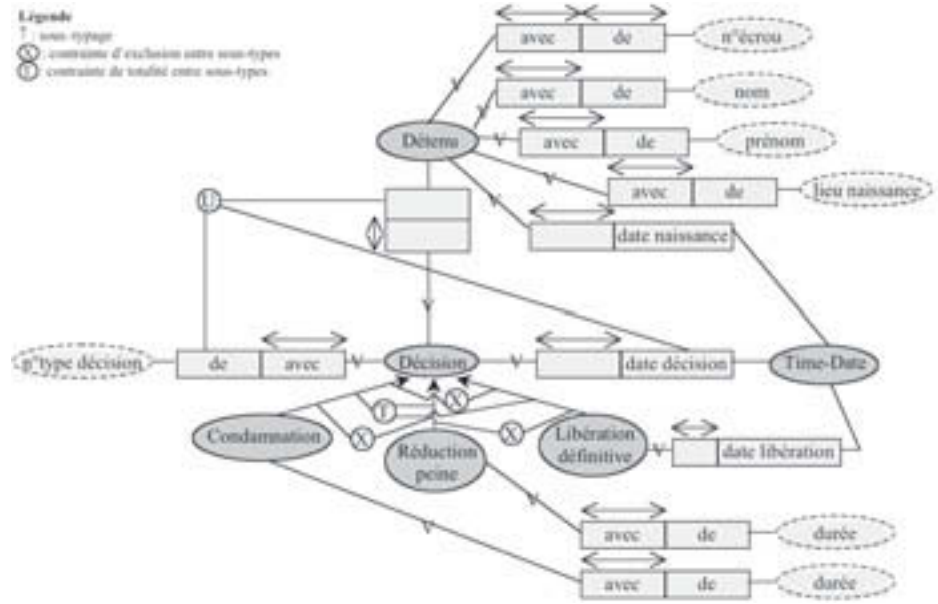


Figure 4.11 – Modèle NIAM (seconde partie) équivalent au modèle UML de la figure 4.6.

condamnation de la durée d'une réduction de peine. Mais sémantiquement, c'est finalement la même chose puisqu'elles s'expriment en jours. Cela dénote donc la même idée et que le domaine de valeurs est le même. Si on ne dessine qu'une fois l'information *durée*, les deux associations pointent vers le même LOT, ce qui est

extrêmement gênant puisque cela ressemble à du pseudo-héritage alors que NIAM possède en natif le principe du sous-typage (interprété de manière ensembliste comme une inclusion). Il s'agit d'un exemple de déficience de NIAM.

A contrario, comme levier de la qualité en modélisation, NIAM est basée sur le modèle relationnel pseudo-binaire (par opposition au binaire « pur »). Il n'y a donc pas en particulier de relation n-aire avec $n > 2$, chose à éviter et évitable en UML via l'écriture de contraintes additionnelles OCL (voir chapitre 2 pour plus de détails). Ne sont jamais utilisées dans les études de cas de cet ouvrage, les relations n-aires de UML avec $n > 2$, grâce justement à cette philosophie du pseudo-binaire dont NIAM est le représentant. Cela améliore la compréhensibilité mais surtout simplifie l'implémentation.

Les défauts et qualités sont donc divers : les LOT ne sont pas typés ; d'un point de vue graphique, les attributs de classe (*e.g. n°type décision*) ne sont pas identifiables, les classes abstraites (*Décision*) sont implicitement le résultat de l'utilisation des contraintes de totalité en sous-types (lettre *T* entourée), etc. Mais par ailleurs, NIAM offre en standard des contraintes puissantes comme la contrainte d'unicité entre rôles (lettre *U* entourée) pour bien cerner l'identifiant de chaque NoLOT, ce qui n'est pas forcément demandé en UML.

4.6 IMPLANTATION EN BASE DE DONNÉES RELATIONNELLE

L'implémentation en base de données relationnelle à l'aide de SQL est plutôt simple. En effet, la façon dont les objets référencent les autres objets est sujette à l'application des règles bien connues de dérivation d'un modèle conceptuel en modèle logique. Ceci étant, quelques subtilités demeurent. Elles sont en particulier liées à l'héritage.

Nous nous limitons ici pour l'implantation à la troisième forme normale sans aller jusqu'à la cinquième. Pour ce qui est des optimisations, comme la « dénormalisation » par exemple qui concerne souvent les applications industrielles en vue d'accroître leur performance, nous les abordons de manière empirique. Rappelons seulement que toute optimisation est néfaste à l'évolution. Cette assertion est universelle en informatique, et s'applique ici au regroupement, voire à la suppression, de différentes tables correspondant à des types dans le modèle (voir ci-dessous pour plus de détails).

4.6.1 Première règle

Pour bien appréhender cette section, on lira avec intérêt l'ouvrage [6] pour se familiariser avec les grandes lignes du relationnel et de SQL (en tant que langage de définition de données). Le livre précité ainsi que cette étude de cas utilisent le SGBDR Access comme support de réalisation : le code SQL qui suit ne fonctionne donc

qu'avec Access. Cet outil étant limité en fonctionnalités, un script SQL plus riche est donné en fin de chapitre. Il a été testé avec le SGBDR Oracle. Ce code ainsi qu'un fichier Access intitulé *Prison_de_Nantes.mdb* sont finalement disponibles sur Internet (voir section « Webographie » en fin de chapitre) pour toute personne souhaitant mettre en œuvre l'étude de cas ou adapter le code à son SGBDR favori.

Un modèle statique UML n'est pas censé fournir systématiquement des identifiants (stéréotype *«unique»*) pour chaque type d'objet décrit. Cependant, lorsqu'il existe, cet identifiant permet de transformer le type d'objet du modèle en table. Exemple avec le type *Détenu* :

```
create table Detenu(
   n_ecrou Integer,
   prenom Text(30),
   nom Text(30),
   date_naissance Date,
   lieu_naissance Text(30),
   constraint Detenu_key primary key(n_ecrou)
);
```

Les types *Text* et *Integer* sont ceux offerts en standard dans Access. On les remplacera avantageusement par *Char* (ou *Varchar*) et *Number* dans une approche SQL plus courante (avec Oracle par exemple).

4.6.2 Deuxième règle

En cas de présence multiple du stéréotype *«unique»* dans un même type, cela signifie que plusieurs identifiants sont candidats. Il convient d'en choisir un pour devenir la clé primaire, les autres se voyant attribuer la contrainte d'unicité SQL. Exemple avec le type *Motif* :

```
create table Motif(
   n_motif Integer,
   libelle_motif Text(50),
   constraint Motif_key primary key(n_motif),
   constraint Motif_unique unique(libelle_motif)
);
```

4.6.3 Troisième règle

Un type d'objet ne présentant qu'un seul attribut, celui-ci étant identifiant, peut être supprimé en tant que table. Cette simplification est néfaste à l'évolution. Par exemple, l'ajout éventuel de nouvelles propriétés à *Juridiction* en figure 4.6 produira une maintenance plus fastidieuse du schéma de base de données. En effet, *Juridiction* peut tout naturellement se développer (nouveaux attributs, nouvelles associations, etc.) dans des versions futures du modèle, en fonction de l'évolution des besoins. Malheureusement, si lors de la première implantation, la table *Juridiction* n'a pas été créée, il faudra ensuite la créer et retoucher les tables qui en dépendent, d'où une maintenance plus conséquente et donc coûteuse.

4.6.4 Quatrième règle

La notion de *qualifier* induit le plus souvent une notion d'unicité relative. En d'autres termes, la cardinalité sur le bras d'une association à l'extrémité opposée au *qualifier* est le plus souvent, mais pas toujours, *1* ou *0..1* comme par exemple, la cardinalité *0..1* côté *Affaire* sur l'association portant le *qualifier n°affaire* (figure 4.6, en haut à droite).

Soit le type d'objet où se colle le *qualifier* (*Juridiction* dans notre exemple). Si ce type est lui-même directement identifiable (*i.e.* présence du stéréotype *«unique»* dans la boîte) ou indirectement (par le fait que l'implémentation d'autres associations permet d'attribuer une clé primaire à ce type d'objet sur lequel le *qualifier* est collé), alors la clé primaire du type d'objet qualifié est la concaténation de la clé primaire du type d'objet identifiable (*Juridiction* dans notre exemple) et du *qualifier*. La clé primaire du type *Affaire* est donc *n°affaire + nom juridiction*. En fait, on exploite la propriété qui dit que dans une juridiction donnée, on ne peut pas trouver deux affaires ayant le même numéro. Cette situation est modélisée via un *qualifier* qui engendre une unicité relative. Rappelons que cela n'est vrai que dans l'hypothèse d'une cardinalité inférieure ou égale à 1 côté *Affaire*. Résultat donc avec l'exemple du type *Affaire* :

```
create table Affaire(
   n_affaire Integer,
   nom_juridiction Text(30),
   date_faits Date,
      constraint Affaire_key primary key(n_affaire,nom_juridiction)
);
```

Le code SQL qui précède est le résultat de l'application conjointe des troisième et quatrième règles. *Juridiction* ne devient pas une table et sa seule propriété *nom juridiction* est supportée par la table *Affaire* qui l'utilise comme partie de clé. Pour compléter le code SQL qui précède, notons qu'il est généralement souhaitable de créer des index avec doublons éventuels sur chaque champ composant une clé primaire composée, cela pour favoriser la rapidité des requêtes.

```
create index Affaire_index on Affaire(n_affaire);
create index Affaire_index2 on Affaire(nom_juridiction);
```

La notion de *qualifier* en UML est donc un pendant à la notion d'identifiant. Qualifier un objet, c'est parfois dire comment l'identifier sans ambiguïté. Par exemple, il n'y a pas qu'une seule affaire ayant le numéro 29 en France, mais dans la juridiction de Besançon, s'il y a une affaire portant le numéro 29, cette affaire est unique. Cela ne marche cependant pas dès lors que l'objet à partir duquel on fait la qualification n'est lui-même pas identifié. En l'occurrence dans l'exemple, si les juridictions n'étaient pas identifiables par leur nom, le qualifier *n°affaire* ne pourrait pas faire office d'outil de caractérisation des affaires. Par ailleurs, insistons sur ce point, la cardinalité à l'extrémité opposée d'un *qualifier* peut être multiple (figure 4.12). La quatrième règle que nous énonçons ici est alors inapplicable. En figure 4.12, pour

une instance de X et une occurrence de *a given qualifier*, il y a plusieurs instances de Y caractérisées, *i.e.* reliées via l'association.

Figure 4.12 – Qualification multiple.

4.6.5 Cinquième règle

Cette règle concerne l'implémentation des associations simple/simple, c'est-à-dire celles dont la cardinalité à chaque extrémité est inférieure ou égale à 1 (*i.e. 0..1*, *1..1* aussi écrit *1*). Vu que nous n'avons pas ce cas précis en figure 4.6, considérons une association simple/simple entre un type d'objet X et un type d'objet Y. Si au moins une des deux cardinalités est strictement égale à 1, côté X supposons (figure 4.13), on est alors en mesure de créer les tables X et Y sous réserve que X présente tout d'abord un identifiant :

```
create table X(
   primary_key_of_X Integer,
     constraint X_key primary key(primary_key_of_X)
);
```

Figure 4.13 – Association simple/simple.

Pour Y, on a alors (version 1) :

```
create table Y(
   primary_key_of_Y Integer,
   foreign_key_to_X Integer,
   constraint Y_key primary key(primary_key_of_Y),
   constraint Y_foreign_key foreign key(foreign_key_to_X) references
      ➥ X(primary_key_of_X)
);
```

Le code SQL qui précède fait l'hypothèse d'un identifiant disponible dans le modèle UML pour *Y*. Ainsi, si un identifiant fait défaut pour le type d'objet *Y*, on peut aboutir à (version 2) :

```
create table Y(
   foreign_key_to_X Integer,
   constraint Y_key primary key(foreign_key_to_X),
   constraint Y_foreign_key foreign key(foreign_key_to_X) references
      ➥ X(primary_key_of_X)
);
```

Une souplesse de la relation simple/simple est de profiter du choix qui est offert dès lors que la cardinalité à une des deux extrémités est strictement égale à 1. L'idée est de doter la table d'un des deux types (*Y* en l'occurrence) de la clé primaire de la table de l'autre type (*X*), ce qui est formellement valide puisque la fonction n'est plus partielle mais totale (au sens mathématique, on appelle aussi cela une « application »). Nous l'avons donc fait ci-dessus en donnant à la table *Y* la clé primaire de la table *X* (version 2), celle-ci devient clé étrangère dans *Y* (ce qui correspond normalement à l'implantation de la relation). Elle devient aussi la clé primaire puisque la cardinalité est strictement égale à 1.

Rappelons quand même qu'avec SQL, par défaut, les champs d'une table peuvent avoir la valeur *null* à l'exception de ceux déclarés comme clé primaire. La première version est donc incomplète car il faut écrire que *foreign_key_to_X* est *not null*. Avec le SGBDR Access, il faut le faire via l'interface utilisateur contrairement au SQL standard, où dans l'instruction *create table* la clause *constraint* autorise aussi d'imposer des valeurs non nulles ainsi que de satisfaire des domaines de valeurs, d'où une troisième version de code SQL non supporté par le SGBDR Access :

```
create table Y(
   primary_key_of_Y Number
   constraint Y_key primary key(primary_key_of_Y),
   foreign_key_to_X Number
   constraint foreign_key_to_X_not_null not null
   constraint Y_foreign_key foreign key(foreign_key_to_X) references
      ➥ X(primary_key_of_X)
);
```

Maintenant, si à chaque extrémité, la cardinalité est *0..1*, il faut créer une table tierce en appliquant la sixième règle (voir ci-dessous) ou s'appuyer sur le fait qu'un champ clé étrangère, s'il n'est pas en même temps clé primaire, peut avoir la valeur *null* pour simuler « 0 », *i.e.* la borne basse de *0..1*. La deuxième version devient alors appropriée puisque le champ *foreign_key_to_X* peut prendre la valeur *null*.

4.6.6 Sixième règle

Cette règle concerne l'implémentation des associations multiple/multiple, c'est-à-dire celles dont la borne haute de la cardinalité à chaque extrémité est strictement supérieure à 1 (*i.e. 1..**, *0..** ainsi que toutes les autres fioritures de notation UML pour les cardinalités, du genre *2,4-9*).

Là encore, il vaut mieux que chaque type d'objet aux deux extrémités de l'association soit identifiable via le stéréotype *«unique»*. Sinon, l'obtention d'une clé primaire résultant de l'implantation préalable d'autres liens est possible. Par exemple, dans le modèle de la figure 4.6, *Affaire* n'a pas à l'origine d'identifiant mais l'implantation du lien entre *Affaire* et *Juridiction* a permis de déterminer une clé primaire dans la table *Affaire*. Par conséquent, d'autres liens impliquant *Affaire* peuvent bénéficier de ce résultat.

Dans la sixième règle, il faut créer de toutes pièces une nouvelle table que l'on nommera, s'il existe, du nom de l'association (il apparaît en italique sur l'association dans un *Class Diagram*). En cas d'absence, la concaténation des noms des deux types à chaque extrémité a l'avantage de favoriser la traçabilité ; exemple avec le couple *Détenu/Affaire* :

```
create table Detenu_Affaire(
   n_ecrou Integer,
   n_affaire Integer,
   nom_juridiction Text(30),
   constraint Detenu_Affaire_key primary key(n_ecrou,n_affaire,nom_juridiction),
   constraint Detenu_Affaire_foreign_key foreign key(n_ecrou) references
      ➥ Detenu(n_ecrou),
   constraint Detenu_Affaire_foreign_key2 foreign key(n_affaire,nom_juridiction)
      ➥ references Affaire(n_affaire,nom_juridiction)
);
```

L'introduction de clés étrangères permet bien entendu de créer l'intégrité référentielle des données dans la table *Detenu_Affaire* par rapport aux tables *Détenu* et *Affaire*. Par ailleurs, il est aussi sain de créer des index (voir plus haut) sur des champs officiant en tant que clé étrangère.

Le fondement des relations multiple/multiple est que la table tierce qui est créée et qui n'a, par définition, aucune correspondance en terme de type d'objet dans le modèle UML, récupère pour clé primaire la concaténation des clés primaires des deux tables extrémités. Dans le cas qui nous intéresse, la table *Detenu_Affaire* va répertorier quel détenu est impliqué dans quelle affaire, une affaire pouvant inclure de nombreux détenus, un détenu pouvant participer à plusieurs affaires.

4.6.7 Septième règle

Cette règle concerne l'implémentation des associations simple/multiple, c'est-à-dire celles dont la borne haute de la cardinalité à une extrémité est inférieure ou égale à 1, ainsi que strictement supérieure à 1 à l'autre extrémité. C'est finalement un cas de figure assez fréquent, spécialement quand la cardinalité simple est égale à *1..1*. Immuablement, dans le cas simple/multiple, la référence se fait du type d'objet où la cardinalité est multiple (prenons l'exemple du type *Incarcération* en figure 4.6) vers le type d'objet où la cardinalité est simple (en l'occurrence *Motif*). Le résultat final avec ce couple *Incarcération/Motif* (la table *Incarceration* est délibérément incomplète pour l'instant) est :

```
create table Incarceration(
   date_incarceration Date,
   n_motif Integer,
   constraint Incarceration_foreign_key foreign key(n_motif) references
      ➥ Motif(n_motif)
);
```

Comme précédemment, il ne faut pas oublier de préciser que le champ *n_motif* est *not null* pour bien respecter la borne basse de *1..**. Par ailleurs, des configurations telles que * d'un côté et *0..1* de l'autre peuvent soit être implantées sur la base de la sixième règle, soit être implantées à l'aide de la septième règle. La sixième règle induit une structuration de base de données plus lourde (ajout d'une table « technique ») avec les conséquences que cela pose à grande échelle notamment, en matière de performance. En comparaison, la septième règle engendre un schéma de base de données plus compact mais moins évolutif, en particulier si des propriétés s'ajoutent sur la relation (voir huitième règle dédiée aux associations classes).

4.6.8 Huitième règle

Cette huitième et dernière règle concerne les classes qui sont des associations. En Merise, cela correspond tout simplement aux propriétés des relations par opposition aux propriétés assignées aux entités. Il y a deux façons d'implanter une *Association Class*. La première est calquée sur la sixième règle sauf qu'il n'y a pas création d'une nouvelle table, c'est l'*Association Class* elle-même qui devient cette table. En clair, si l'on prend l'exemple du type d'objet *Incarcération*, la table *Incarceration* est créée et fait référence à la table *Detenu* ainsi qu'à la table *Affaire* (nous complétons la dernière version de la table *Incarceration* écrite ci-dessus) :

```
create table Incarceration(
   n_ecrou Integer,
   n_affaire Integer,
   nom_juridiction Text(30),
   date_incarceration Date,
   n_motif Integer,
   constraint Incarceration_key primary key(n_ecrou,n_affaire,nom_juridiction),
   constraint Incarceration_foreign_key foreign key(n_ecrou) references
      ➥ Detenu(n_ecrou),
   constraint Incarceration_foreign_key2 foreign key(n_affaire,nom_juridiction)
      ➥ references Affaire(n_affaire,nom_juridiction),
   constraint Incarceration_foreign_key3 foreign key(n_motif) references
      ➥ Motif(n_motif)
);
```

Une propriété toujours vraie est que la concaténation des clés primaires des tables extrémités, *Detenu* et *Affaire* ici en l'occurrence, est unique dans la table *Incarceration* (la sixième règle s'appuie sur cette possibilité). Une analyse fine montre néanmoins que la cardinalité côté *Détenu* est * alors qu'elle est *1* côté *Affaire* pour l'association incarnée par *Incarcération*. Certains y verront le moyen, au sein même du modèle UML, de lier *Incarcération* directement à *Détenu* et ainsi lui faire perdre son statut d'*Association Class*. Comme nous l'avions mentionné, on perd alors en

compréhensibilité (l'incarcération d'un détenu est relative à l'affaire principale qui l'a conduit en prison), ce qui est néfaste en analyse. En revanche, ici en conception, la table *Incarceration* peut être optimisée pour tirer profit du fait qu'une des deux cardinalités à chaque extrémité est inférieure ou égale à 1. On aboutit alors à la simplification de la clé primaire de la table *Incarceration* comme suit :

```
create table Incarceration(
   n_ecrou Integer,
   n_affaire Integer,
   nom_juridiction Text(30),
   date_incarceration Date,
   n_motif Integer,
   constraint Incarceration_key primary key(n_ecrou),
   constraint Incarceration_foreign_key foreign key(n_ecrou) references
      ➥ Detenu(n_ecrou),
   constraint Incarceration_foreign_key2 foreign key(n_affaire,nom_juridiction)
      ➥ references Affaire(n_affaire,nom_juridiction),
   constraint Incarceration_foreign_key3 foreign key(n_motif) references
      ➥ Motif(n_motif)
);
```

Ce qui change, c'est donc que la clé primaire de la table *Incarceration* est maintenant *n_ecrou* du fait d'une étude plus fine des cardinalités. De manière générale, si les cardinalités sont multiples à chaque extrémité, la clé primaire est la concaténation des clés primaires, sinon elle se simplifie.

Pour terminer, insistons sur la façon dont on est amené à considérer l'affaire principale qui a conduit un détenu en prison : via justement la table *Incarceration* en fait, qui comporte toutes les informations relatives au caractère « principale » de l'affaire (*i.e.* la date d'incarcération).

4.6.9 Contraintes

Les contraintes qui renforcent la qualité d'un modèle UML sont de nature diverse, plus ou moins formelles, directement implantables dans un cadre relationnel ou non. La contrainte *{subsets <propery name>}* signifie pour un type d'objet *Y* relié par deux associations à *X*, la nécessité de satisfaire une contrainte d'inclusion. Un sous-ensemble d'instances de *Y* est inclus dans un autre sous-ensemble de *Y*, ces deux sous-ensembles étant atteignables à partir de *X*. Par exemple, l'affaire portant un numéro donné dans une juridiction, lorsqu'elle existe, n'est jamais qu'un singleton inclus dans l'ensemble de toutes les affaires gérées par cette juridiction. Le mécanisme de clé étrangère peut être un outil pour vérifier une telle contrainte. Encore faut-il que les tables soient structurées de manière à favoriser cette vérification et que la vérification soit pertinente. En effet, si l'on considère la contrainte *{subsets toutes}* sur l'association de *Juridiction* vers *Affaire*, l'implantation de la table *Affaire* que nous avons retenue la vérifie implicitement via la clé primaire choisie. En revanche, il n'est pas inutile de contrôler la contrainte *{subsets toutes}* pour les deux associations entre *Détenu* et *Affaire* cette fois.

Nous allons cependant légèrement déroger à cela en travaillant sur le produit cartésien *Détenu x Affaire* plutôt que sur l'ensemble des instances d'*Affaire* seul. Cette transgression du modèle résulte du manque de pouvoir de la contrainte *{subsets <propery name>}*. Ainsi, formellement, la table *Detenu_Affaire* représente informatiquement un sous-ensemble du produit cartésien *Détenu x Affaire*. La table *Incarceration* incarne informatiquement un autre sous-ensemble de ce produit cartésien inclus dans le premier sous-ensemble. Les contraintes d'intégrité référentielle vers *Detenu* et *Affaire* sont déjà codées dans *Detenu_Affaire*. Il n'est pas utile donc de les implanter dans la table *Incarceration*. Il suffit d'implanter une intégrité référentielle de la table *Incarceration* vers la table *Detenu_Affaire*, le contrôle d'intégrité référentielle vers les tables *Detenu* et *Affaire* s'effectuant transitivement, d'où la version finale de la table *Incarceration* :

```
create table Incarceration(
   n_ecrou Integer,
   n_affaire Integer,
   nom_juridiction Text(30),
   date_incarceration Date,
   n_motif Integer,
   constraint Incarceration_key primary key(n_ecrou),
   constraint Incarceration_foreign_key foreign
      ➥ key(n_ecrou,n_affaire,nom_juridiction) references
      ➥ Detenu_Affaire(n_ecrou,n_affaire,nom_juridiction),
   constraint Incarceration_foreign_key2 foreign key(n_motif) references
      ➥ Motif(n_motif)
);
```

Dans ce code SQL, la contrainte *Incarceration_foreign_key2* est la réalisation de la contrainte d'inclusion appliquée aux ensembles de couples *Détenu/Affaire*. La contrainte *{xor}* en figure 4.7 décrit des intersections entre ensembles qui sont vides. Son implantation est plus délicate car *a priori* procédurale ou tout du moins plus liée aux moyens offerts par un langage d'implantation plus moderne que SQL2 : SQL99. De manière plus générale, l'hétérogénéité des types de contrainte fait appel à une implantation plus experte et par là même moins méthodique.

En implantation relationnelle, le plus grand problème est d'équilibrer le contrôle de violation des contraintes entre le moteur de la base de données et les programmes clients de manipulation de ces données. À l'extrême, une bonne interface utilisateur peut toujours très facilement limiter le choix de saisie de manière à faire tendre vers zéro la probabilité d'entrer une donnée inappropriée. Croire que toutes les applications clientes le font et le feront bien est non seulement utopique mais est source de duplication de ce code de contrôle, à moins d'une approche composant logiciel « sûre », impliquant le partage de ce code par les clients (introduction d'un intergiciel, *middleware* en anglais, voir la technologie EJB en fin de ce chapitre). Par ailleurs, le support d'activation et de désactivation du code de satisfaction des contraintes (clause *disable* en SQL par exemple) dans les SGDBR laisse une certaine marge de manœuvre (pour jouer sur la rapidité des requêtes par exemple), et doit donc inciter à centraliser le contrôle au niveau du serveur ou des couches intermé-

diaires (*tiers*). On rentre ici dans des considérations de distribution/répartition et d'architecture qui dépassent le simple cadre de ce chapitre.

4.6.10 Héritage

Le cas de l'héritage est intéressant car spécifique. Pour aborder l'implantation relationnelle de modèles UML comme celui de la figure 4.6, il faut tout d'abord interpréter la relation d'héritage comme une relation d'inclusion. En d'autres termes, en figure 4.6, cela signifie que l'ensemble des instances du type d'objet *Condamnation* est inclus dans l'ensemble des instances du type d'objet *Décision*. Brièvement, ce genre d'interprétation est celui de la relation conceptuelle *Is-a* (voir chapitre 2) par opposition aux autres sortes d'héritage *Subclassing* et *Subclassing*. UML, de manière plus ou moins soignée et claire, supporte aussi ces deux dernières notions. Ensuite, il faut regarder les possibilités offertes par le langage de définition de données. SQL99 offre une syntaxe et une sémantique pour l'héritage simple mais son implémentation dans les SGBDR du marché reste plutôt éclectique (clause inherits dans PostgreSQL, under dans Oracle avec beaucoup de limitations, absent dans Access, etc.). Nous proposons ici une approche plus pragmatique et en tout cas plus portable d'un SGBDR à l'autre.

Cette hypothèse posée, deux heuristiques simples permettent de juger de l'intérêt de l'implantation d'un supertype. *Décision* dans le modèle de la figure 4.6 joue ce rôle de supertype pour trois sous-types de décision. Première heuristique : ce supertype *Décision* factorise-t-il réellement autre chose qu'un comportement ? Autrement dit, en s'intéressant aux aspects statiques, *Décision* possède-t-il des relations de structure propres ? La réponse est oui : la relation avec *Détenu*, la propriété *date décision*, etc. La seconde heuristique pose la question : ce supertype est-il identifiable ? En d'autres termes, peut-on trouver une clé primaire assurant une distinction sûre des instances du type *Décision*, qu'elles soient directes ou indirectes, « indirectes » au sens où elles résultent de l'instanciation de sous-types de ce supertype ? Primo, *Décision* étant une classe abstraite, elle n'a pas d'instances directes. Secundo, la contrainte OCL relative à *Décision* n'est rien d'autre qu'un identifiant. En conclusion, toutes les instances de *Décision* doivent avoir une valeur de clé primaire différente, quel que soit le sous-type de *Décision* à partir duquel elles ont été créées.

Dans un premier temps, nous laissons de côté le problème d'implantation des propriétés de classe. La table *Decision* pouvant être dotée d'une clé primaire compte tenu de la contrainte écrite à gauche en note dans la figure 4.6, on obtient :

```
create table Decision(
   n_type_decision Integer,
   n_ecrou Integer,
   date_decision Date,
   constraint Decision_key primary key(n_type_decision,n_ecrou,date_decision),
   constraint Decision_foreign_key foreign key(n_ecrou) references
      ➥ Detenu(n_ecrou)
);
```

À partir de là, l'implantation de la table *Reduction_peine* est immédiate car, rappelons-le, le lien d'héritage étant vu comme l'inclusion, une contrainte de clé étrangère peut suffire à réaliser le lien d'héritage. Néanmoins, cette implantation n'est pas élégante au regard de l'attribut de classe *n°type décision*. Certes, ce champ contribue à supporter la réalisation du lien d'héritage vu qu'il fait partie de la clé étrangère dans *Reduction_peine*, clé étrangère qui fait référence à la table *Decision*. Mais en termes de mémorisation, il informe de manière redondante que les réductions de peine sont numérotées *2* puisqu'il est implanté comme variable d'instance. Voici le résultat (version 1) :

```
create table Reduction_peine(
   n_type_decision Integer,
   n_ecrou Integer,
   date_decision Date,
   duree Integer,
   constraint Reduction_peine_key primary
      ➥ key(n_type_decision,n_ecrou,date_decision),
   constraint Reduction_peine_foreign_key foreign
      ➥ key(n_type_decision,n_ecrou,date_decision) references
      ➥ Decision(n_type_decision,n_ecrou,date_decision)
);
```

Pour les avantages, il est clair que cette implantation empêche automatiquement d'entrer dans la table *Reduction_peine*, une instance qui aurait le même identifiant qu'une condamnation par exemple, cela à cause de la factorisation du contrôle d'unicité dans la table *Decision* elle-même ! La contrainte d'héritage *{disjoint}* du modèle en figure 4.6 est donc systématiquement satisfaite. En fait, on ne peut vérifier les contraintes de disjonction entre sous-types que si la table *Decision* est construite.

Différemment, on peut fixer pour règle absolue qu'une classe abstraite n'est jamais incarnée sous forme de table. En ce cas, la table *Reduction_peine* change quelque peu de forme (version 2) :

```
create table Reduction_peine(
   n_type_decision Integer,
   n_ecrou Integer,
   date_decision Date,
   duree Integer,
   constraint Reduction_peine_key primary key(n_ecrou,date_decision),
   constraint Reduction_peine_foreign_key foreign key(n_ecrou) references
      ➥ Detenu(n_ecrou)
);
```

Le lecteur attentif verra dans ce code SQL l'exacte implantation du modèle de la figure 4.8 où le *qualifier date décision* est mis en œuvre. Le type d'objet *Décision* n'étant pas transformé en table, les contraintes se doivent d'être prises en charge dans chaque sous-type. Par exemple, dans la version 1, la contrainte d'intégrité référentielle vers la table *Detenu* est écrite une fois pour toutes dans la table *Decision* alors que dans la version 2, il faut la gérer dans *Reduction_peine* mais aussi dans les tables *Condamnation* et *Liberation_definitive*. De plus, la contrainte *{disjoint}* ne peut

être contrôlée. En figure 4.14 se trouve le schéma de base de données définitif généré par le SGBDR Access.

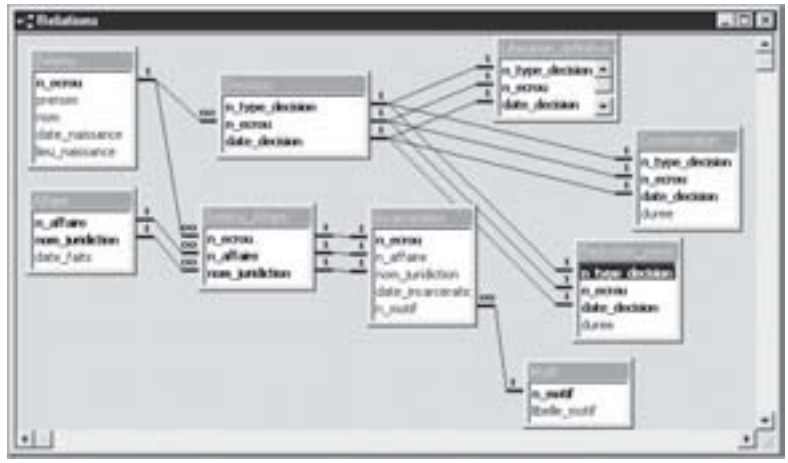


Figure 4.14 – Schéma de base de données relationnelle du cas « prison de Nantes ».

Le formalisme utilisé en figure 4.14 (propre à Access) est ∞ pour signifier la possibilité d'existence multiple de la clé étrangère dans la table. Par exemple, on peut trouver plusieurs fois la même valeur pour le champ *n_ecrou* dans la table *Decision*. En effet, plusieurs décisions peuvent être prises à l'égard du même prisonnier. Le symbole *1* indique le contraire.

4.6.11 Agrégation/composition

Donnons un bref mot sur l'implantation relationnelle de l'agrégation et de la composition UML. Le manque évident de sémantique précise pour ces deux sortes de relation ne permet pas de considérer une différence avec l'association. Cependant, on peut juste souligner la possibilité d'utiliser l'option *on delete cascade* lors de la définition d'une clé étrangère. Spécialement pour la composition qui suggère la coïncidence de vie du composé et du composant.

L'exemple qui suit suppose que l'on crée la table *Prison_de_Nantes* avec un identifiant quelconque *Prison_de_Nantes_id*. Rappelons le caractère pédagogique de cette création car, comme nous l'avons évoqué, le type d'objet *Prison de Nantes* n'est là qu'à des fins de documentation en figure 4.6. On obtient alors une version amendée de la table *Decision*. Cette implantation n'est pas supportée dans Access par le fait que l'option *on delete cascade* ne peut être activée que via l'interface utilisateur mais pas par le moyen du code SQL que l'on écrit. Par comparaison, ce code tourne parfaitement avec Oracle :

```
create table Detenu(
   n_ecrou Integer,
```

```
    prenom Text(30),
    nom Text(30),
    date_naissance Date,
    lieu_naissance Text(30),
    composition Integer,
    constraint Detenu_key primary key(n_ecrou),
    constraint Detenu_foreign_key foreign key(composition) references
        ➥ Prison_de_Nantes(Prison_de_Nantes_id) on delete cascade
);
```

4.6.12 Règles générales, synthèse

La première chose à bien comprendre est que la méthode d'implantation énoncée ci-dessus peut dans sa mise en œuvre, conduire à un goulot d'étranglement. En d'autres termes, le modèle UML peut être suffisamment mal fait et/ou incomplet pour que l'implantation relationnelle soit irréalisable. C'est le cas lorsqu'il manque des identifiants qui vont gêner, voire empêcher, la transformation MDE en tables. Pour s'en sortir, l'ajout de clé primaire « artificielle » est à notre sens la pire des choses (à éviter donc) car c'est souvent un travers de l'informaticien que de poser en termes techniques, des problèmes qui résultent d'une analyse inachevée, voire ratée. Il faut toujours réfléchir avant d'introduire une information supplémentaire (voir étude de cas QUALIF en toute fin de chapitre 2) comme peut l'être une clé primaire « technique », compte tenu du coût inhérent direct et indirect de sa gestion. Il est en effet rare dans une organisation que ces identifiants *a priori* manquants ne soient pas cachés quelque part, dans les esprits et les habitudes en particulier. Par ailleurs, un modèle conceptuel objet n'est pas sujet à la règle « tout type d'objet doit avoir un identifiant ». C'est un problème purement culturel car en informatique industrielle (par opposition à l'informatique de gestion) les relations entre objets sont souvent uniquement simples (cardinalité *1* par exemple, voir étude de cas du chapitre 5) d'où l'inutilité d'identifiants. De plus, l'implantation relationnelle n'est jamais qu'une forme particulière d'implantation d'où une minimisation du caractère indispensable des identifiants en approche objet. Finalement, il existe toujours, mais en ultime recours, des moyens de pallier ces problèmes de clé. Le concept d'OID en SQL99 (adresse absolue d'un objet ou ligne dans une table) est un subterfuge, certes douteux mais éventuellement utile, pour obtenir un identifiant implicite d'objet en implantation relationnelle (on parle aussi plus volontiers de « relationnel objet »). Nous ne conseillons cependant pas son usage en « relationnel pur » à cause des technologies composant comme celle des EJB qui s'appuient fortement sur des clés primaires explicites et non des adresses système implicites (voir ci-dessous).

Une seconde chose à bien intégrer est la difficulté d'automatisation du processus d'implantation énoncé ci-dessus. Nombreux sont aujourd'hui les outils MDE du marché qui offrent une fonctionnalité de génération de code SQL. La grande question est de savoir comment ils prennent en compte les contraintes OCL, par exemple celles décrivant le critère d'unicité du type d'objet *Décision*. C'est surtout à ce titre que NIAM offre un formalisme plus sûr au sens où les algorithmes de transfor-

mation de modèles tels ceux en figure 4.10 et figure 4.11 sont aujourd'hui écrits, fiables et disponibles. Pour traiter le problème en UML, notre crainte est que beaucoup d'outils font des courts-circuits en ignorant les expressions OCL. On peut donc s'interroger sur la qualité du schéma logique généré et à ce titre, l'étude de cas de ce chapitre nous paraît être un excellent test.

Par ailleurs, le format de stockage des modèles UML est maintenant normalisé à travers l'instanciation nommée XMI (DTD pour *Document Type Definition*) du métalangage XML[1]. Un générateur de code SQL ne peut donc s'appuyer que sur les informations qu'il va trouver dans un texte XMI, sachant qu'OCL est mal supporté pour l'instant. En clair, notre démarche reste donc difficilement automatisable alors qu'un outil doté d'un interpréteur OCL et de quelques règles de modélisation rigoureuses pourraient formellement créer des schémas relationnels de bonne tenue. Il suffit en particulier de s'inspirer des algorithmes utilisés pour NIAM.

Plus globalement, nous allons nous intéresser à la construction d'applications sur une plus grande échelle, l'implantation relationnelle n'étant alors qu'une partie du problème. Techniquement, cela se résume surtout à la prise en charge de la persistance. Dans le cadre du client/serveur, la conception, la génération des programmes client, la réflexion sur l'architecture logicielle à mettre en place ainsi que bien d'autres facteurs montrent en outre l'intérêt de formalismes puissants et ouverts basés sur XML, cela pour être en mesure de « muscler » UML. Par exemple, les stéréotypes UML peuvent voir leur sémantique définie via OCL (*«unique»* par exemple) puis être exprimée sur la base d'une DTD XML pour être utilisés tout au long du processus de développement. Ces stéréotypes peuvent être particulièrement efficaces pour des cadres de développement bien bornés, tels les EJB.

4.7 IMPLANTATION EJB

Nous donnons ici une implémentation « technologiquement ouverte » du cas de la prison de Nantes, nous appuyant pour cela sur le cadre de développement EJB. C'est en effet un cadre puissant au sens où il supporte la « distribuabilité » des applications dans le monde Java. Les composants logiciels peuvent en l'occurrence être associés (concept de déploiement) à des machines virtuelles Java différentes, tournant si besoin sur des machines physiques différentes. Les EJB étant des composants logiciels « orientés serveur », il est au minimum nécessaire de faire tourner une instance d'un tel serveur qui prend en charge toute la gestion des EJB, de leur création à leur mort en passant par des phases types.

Dans un tel contexte, la conception d'une application revient à élaborer des EJB dont la répartition sur un réseau est dissociée du développement proprement dit. En d'autres termes, une première activité d'administration se crée consistant pour le

1. XMI 2 est cependant non stabilisé, à la date de rédaction de ces lignes.

programmeur d'EJB à écrire en XML des descripteurs de déploiement. Ceux-ci indiquent et déclarent, pour l'essentiel, des entités et ressources requises dans l'environnement où ils vont s'exécuter. On entend par là des points d'entrée à des sources de données (connexions à des bases de données relationnelles), des files d'attentes de messages où ils peuvent poster et consommer des messages, des points d'accès et de communication avec d'autres EJB (concept de *Home Interface*, voir ci-après), etc.

L'esprit est au niveau du serveur de favoriser la mutualisation des traitements et des services aux applications clientes, d'avoir des moyens de développer des politiques de répartition (équilibrage de la charge) et plus généralement de concentrer la tâche de programmation proprement dite sur le volet « métier ». Un contexte alternatif verrait le programmeur manipuler, dans le meilleur des cas, des outils de programmation distribuée comme RMI (*Remote Method Invocation*) dans le monde Java ou encore des *sockets*. Il résulte de tout cela une seconde tâche d'administration qui consiste à associer aux entités et ressources « logiques » (auxquelles le code fait référence dans les descripteurs de déploiement), des systèmes « réels » comme l'URL d'une machine où tourne un SGBDR ainsi qu'un point d'entrée (nom du service accédé dans le SGBDR, un compte utilisateur et son mot de passe par exemple). À deux niveaux donc, on peut en changeant les valeurs des attributs dans les fichiers XML associés aux EJB (tâche du programmeur) et celles des propriétés définies dans le serveur J2EE des EJB (voir section « Webographie » en fin de chapitre pour une liste des principaux serveurs EJB du marché), rendre les applications plus pérennes et plus réutilisables par le fait que leur code s'abstrait des caractéristiques réelles de l'environnement de fonctionnement.

Le but de ce chapitre et de cet ouvrage n'étant cependant pas de parcourir toute l'immensité de la problématique des EJB, nous renvoyons le lecteur à la spécification originelle [8] ainsi qu'aux nombreux livres (en anglais pour l'essentiel) sur le sujet. Nous allons donc donner quelques éléments techniques clés sans détailler néanmoins « le pourquoi du comment ». D'un point de vue didactique, il est en effet plus probant de découvrir les concepts inhérents aux EJB non pas dans l'absolu, mais comme des supports pour implanter des modèles UML en général et pour le cas de la prison de Nantes en particulier. Nous mettons donc en exergue les aspects architecturaux : quels EJB dériver des modèles UML ? Quelles relations implanter ? Comment les implanter ? Quel est le code Java ? Quels sont les fichiers XML pour le déploiement ? Comment s'opèrent, succinctement, la mise en place et l'administration des entités et ressources requises ? Celles-ci doivent être en fait configurées pour la bonne marche d'une application de gestion de la prison de Nantes que nous voulons assez générale : ajout (figure 4.15), suppression et mise à jour d'informations ainsi que quelques calculs simples : quels sont les détenus en préventive par exemple ? La figure 4.15 donne l'idée de la couche « présentation » de type Web pouvant être implémenté via la technologie des *Servlet*.

L'architecture finale dite « à trois niveaux » (*3-tiers*) (figure 4.16, dans sa version la plus simple) revient à considérer le serveur J2EE comme un intergiciel : il est « au milieu », « à l'intersection », de la base de données de la prison de Nantes et des

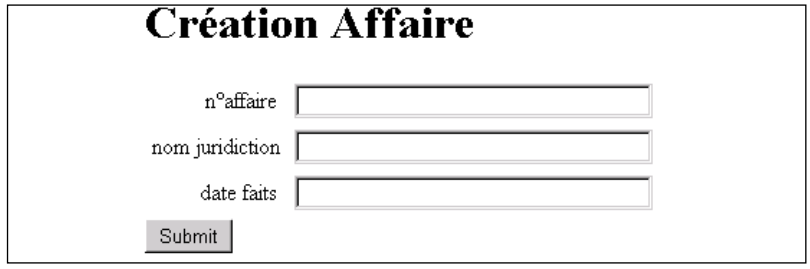


Figure 4.15 – Interface Web de saisie d'une instance du type d'objet *Affaire*.

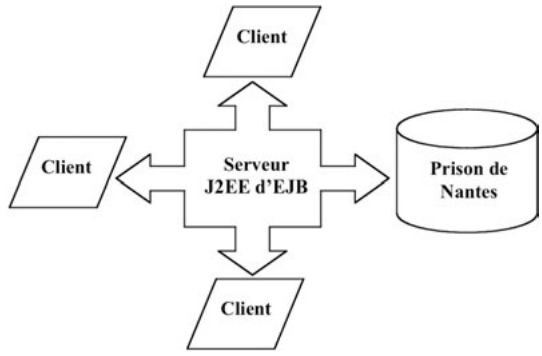


Figure 4.16 – Architecture classique à trois niveaux (présentation, intergiciel et base de données) en EJB.

applications clientes dotées d'une interface utilisateur comme en figure 4.15 et établies selon les besoins.

4.7.1 Principes élémentaires des EJB

Un EJB est un agglomérat de classes et d'interfaces Java préformatées possédant chacune à ce titre un ensemble de services prédéfinis. Chacun de ces services a en particulier une signature bien arrêtée que le serveur d'EJB est censé appeler. Les EJB se distinguent en trois catégories : *Session Bean* (avec les sous-catégories *Stateless* et *Stateful*), *Entity Bean* et *Message-Driven Bean* (voir aussi chapitre 2 concernant le profil UML dédié). La contribution du programmeur se résume à donner éventuellement un corps aux fonctions prédéfinies, en personnaliser certaines (celles de création par exemple avec des paramètres spécifiques) et en inventer de nouvelles, qualifiées de « métier ».

Le serveur d'EJB associe un conteneur à chaque EJB qui est en fait un contrôleur jouant le rôle d'intermédiaire avec l'environnement global. Ce conteneur gère le cycle de vie de l'EJB via l'appel de méthodes prédéterminées. Le programmeur se charge de donner un corps à ces méthodes qui parfois est purement et simplement vide, comme nous allons le voir. À ce jour, les conversations autorisées entre le con-

teneur et l'EJB sont plutôt réduites pour, de manière saine, minimiser une tendance néfaste qui voudrait que le programmeur s'informe en détail sur le contexte d'exécution et « câble » alors son code en fonction, rendant les composants logiciels peu réutilisables. Tout au plus, des synchronisations sur les débuts et fins de transaction par exemple permettent dans le code d'agir en tenant compte des moments clés. Le lien avec l'environnement s'opère toujours comme suit (c'est la première méthode appelée par le conteneur) :

```
private javax.ejb.SessionContext _ejb_context; // type javax.ejb.EntityContext
pour Entity Bean
public void setSessionContext(javax.ejb.SessionContext ejb_context) { //
setEntityContext pour Entity Bean
   _ejb_context = ejb_context;
}
```

Le champ *_ejb_context* va donc être la passerelle pour accéder au contexte d'exécution jusqu'à ce qu'il soit invalidé par le serveur ou par le programmeur, second cas propre aux *Entity Bean* :

```
public void unsetEntityContext() {
   _ejb_context = null;
}
```

Puisque toute application engendre naturellement et quasi systématiquement des besoins en persistance, la technologie EJB s'appuie sur les bases de données relationnelles pour prendre en charge de telles exigences techniques. Il en résulte que l'implantation EJB du cas de la prison de Nantes est une extension naturelle de l'implantation relationnelle de la section 4.6. Ce sont les *Entity Bean* qui ont vocation de persistance et qui vont donc incarner, mimer (au regard des applications clientes) le schéma relationnel dans la « couche intergiciel ».

La spécificité du cas de la prison de Nantes nous amène à découvrir et manipuler intuitivement les *Entity Bean* et les *Session Bean* (un seul dans l'application, appelé *Prison_de_Nantes* et incarnant le type *Prison de Nantes* du modèle UML de la figure 4.6) mais pas les *Message-Driven Bean* qui possèdent peu des propriétés des deux premières catégories.Pour cela et plus globalement, un certain nombre de technologies Java transversales à la technologie EJB (*i.e.* utilisables indépendamment des EJB mais incontournables pour les EJB) sont nécessaires. Ce sont pour l'essentiel RMI (dont la manipulation usuelle est masquée à l'exception de la gestion d'exceptions de type *java.rmi.RemoteException*), JNDI pour *Java Naming and Directory Interface*, JDBC pour *Java DataBase Connectivity* et plus marginalement JMS (*Java Message Service*) pour les *Message-Driven Bean*. On lira donc avec grand intérêt [1] pour une prise en mains de ces technologies.

4.7.2 Organisation canonique des EJB

L'idée de la figure 4.17 est de développer le profil UML vu au chapitre 2 non pas seulement en créant des stéréotypes mais en créant des métaclasses pour chaque constituant d'un EJB. Nous avions déjà introduit les métaclasses *EJB home interface*, *EJB*

remote interface et *EJB implementation class* au chapitre 2 qui toutes nous semblaient manquer dans UML 2.x. Nous avons ajouté ici d'autres métaclasses pour aboutir à la figure 4.17 qui donne une vision synthétique d'un EJB.

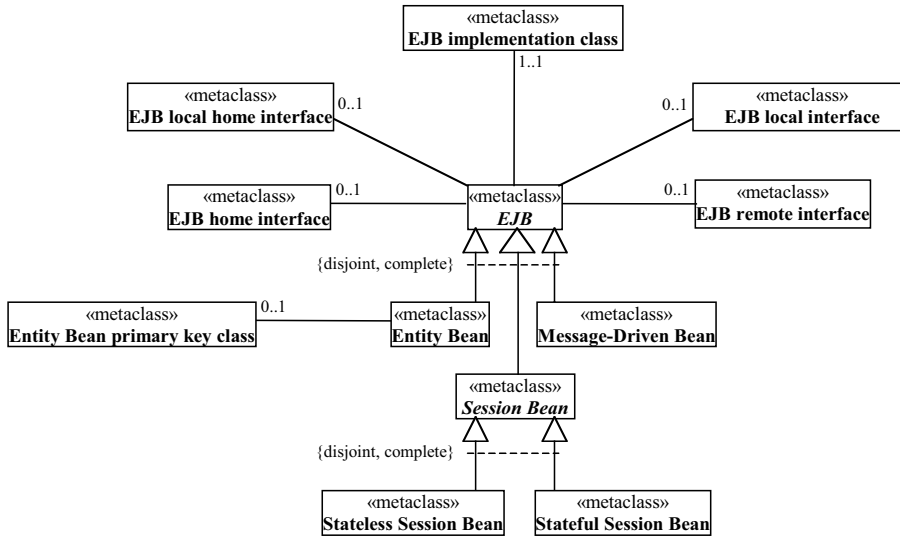


Figure 4.17 – Structure standard d'un EJB (constituants) vue au niveau métamodèle.

La figure 4.17 montre que les trois grandes catégories d'EJB diffèrent par les propriétés qu'elles possèdent. Par exemple, un *Message-Driven Bean* n'a pas de *Home Interface* et de *Remote Interface*, qu'elles soient locales (adressage de l'EJB par un autre EJB dans une même machine virtuelle Java) ou non. Cela donne en OCL :

```
context Message-Driven Bean inv:
   ejb local home interface→isEmpty()
   ejb local interface→isEmpty()
   ejb home interface→isEmpty()
   ejb remote interface→isEmpty()
```

Il faudrait ajouter de nombreuses autres contraintes OCL pour tout préciser. Nous nous limitons ici seulement à une découverte des aspects techniques des EJB, sans exhaustivité, via le cas de la prison de Nantes.

Ainsi, pour bien comprendre, il nous faut éclaircir les notions de *Home Interface* et de *Remote Interface*. La *Remote Interface* est l'angle d'attaque prioritaire d'un programme client. Pour une instance d'EJB, cette interface décrit les services offerts. Par exemple pour l'EJB *Affaire*, nous pouvons imaginer la lecture (*getter* Java) des propriétés *n°affaire* et *nom juridiction* mais pas leur écriture[1] (*setter* Java) car ces deux attributs constituent la clé. Pour la propriété *date faits*, on peut au contraire tolérer

1. Attention au code qui suit : des *setter* sur *n°affaire* et *nom juridiction* existent mais ne sont pas dans la *Remote Interface* de l'EJB *Affaire*. Ils sont en fait utilisés en interne par le serveur J2EE.

lecture et écriture. Sinon, la norme EJB impose des constructions canoniques selon le type d'EJB. Hériter de *EJBObject* est par exemple obligatoire pour une *Remote Interface*. Lever l'exception *RemoteException* est aussi une clause rédhibitoire lors de la déclaration de services tels que ceux que nous avons évoqués, etc. Cela correspond au « formatage » des interfaces et des classes permettant de se conformer au cadre des EJB.

La *Home Interface* est un point d'entrée à toutes les instances d'un EJB. C'est aussi une usine (*factory*) au sens où tous les *Entity Bean* de type *Affaire* par exemple, sont recherchés, créés et détruits via une instance d'une classe (créée par le serveur J2EE au moment du déploiement) qui implémente (au sens Java, relation *implements*) la *Home Interface*. Le premier travail d'un programme client est donc de localiser cet objet « serveur d'EJB » d'un type donné. C'est la technologie JNDI qui est utilisée (voir exemples de code plus loin). Dans la même ligne de raisonnement que pour la *Remote Interface*, des archétypes de micro-architectures sont nécessaires : hériter de *EJBHome*, etc.

4.7.3 Codage des EJB

La figure 4.18 donne de façon synthétique les navigations induites par le schéma Access[1] présent en figure 4.14. En d'autres termes, les contraintes d'intégrité référentielle ne sont rien d'autres que des liens orientés d'un type d'objet, incarné sous forme de table, vers un autre, lui aussi représenté par une table. La conception EJB doit donc logiquement, mais pas seulement, fournir des services en cohésion avec ces navigations. Par exemple, créer un nouvel objet *Incarcération*, c'est nécessairement préciser à quel couple *Détenu/Affaire* fait référence cette instance d'*Incarcération* (association orientée d'*Incarceration* vers *Detenu_Affaire* en figure 4.18). Satisfaire cette contrainte d'intégrité référentielle est immédiat dans un moteur où la base de données relationnelle est centralisée mais moins évident si la base est répartie, si les programmes client travaillent avec des caches, si les objets sont répliqués, etc. Les transactions deviennent donc un élément critique dans le maintien de la cohérence globale de la base de données d'où la nécessité d'un outillage approprié et bien étoffé, offert en l'occurrence par la technologie EJB.

Notre stratégie d'implantation de cette étude de cas va donc maximiser l'utilisation des facilités système offertes en standard par le serveur J2EE. En pratique, le code Java à produire est donc minimal en regard des paramètres de déploiement à fixer, eux plus nombreux. Plus précisément, une facilité est justement la possibilité de faire gérer au serveur les relations entre EJB en fonction de contraintes d'intégrité référentielle comme celles de la figure 4.18 ainsi que de se décharger de la gestion « à la main » de la persistance et des transactions. Le code Java n'utilise donc pas JDBC pour la persistance et JTA (*Java Transaction Architecture*) pour les transactions mais repose sur le serveur.

1. L'implantation Oracle est donnée en fin de chapitre.

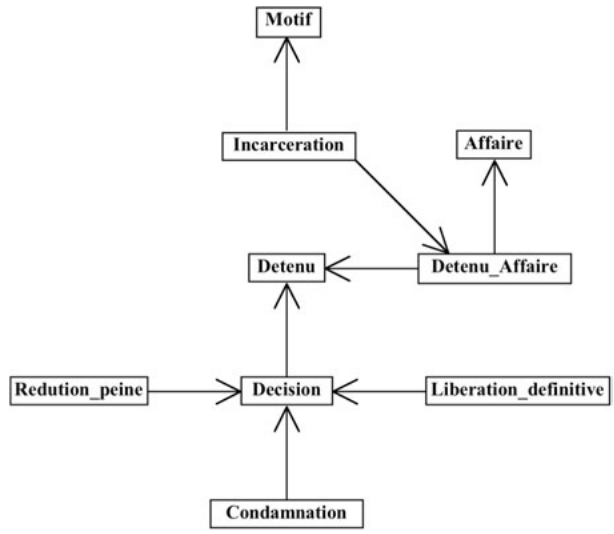


Figure 4.18 – Couplage relatif à l'implantation relationnelle.

La figure 4.18 est le résultat d'une découverte et d'une consolidation de notions stables dans la logique de gestion de la prison de Nantes. Chaque type d'objet de la figure 4.18 a déjà une connotation opérationnelle par le fait qu'il existe sous forme de table dans la base de données. Puisque ces tables constituent la structure nécessaire et suffisante pour gérer toutes les informations de la prison de Nantes, il devient « juste nécessaire » de lister les *Entity Bean* (figure 4.19) qui vont servir de courtier entre les programmes client et la base, ce qui impose la description d'une correspondance (*mapping* en anglais) formelle entre les *Entity Bean* et les tables.

La technologie des EJB supporte deux modes de pilotage des *Entity Bean* qui sont le mode *Bean-Managed Persistence* (BMP) et le mode *Container-Managed Persistence* (CMP). Le premier laisse au programmeur la charge d'implémenter le protocole de dialogue entre la base de données et l'*Entity Bean* (appel SQL embarqué dans le code Java, technologie JDBC, voir chapitre 6) alors que le second laisse cette tâche au serveur J2EE. De manière plus générale, la deuxième méthode est beaucoup plus flexible et évolutive au sens où le code Java va s'alléger : les références « en dur » aux noms de champ d'une table par exemple vont disparaître des programmes au profit des descripteurs de déploiement écrits en XML (voir plus loin).

Dans le mode CMP retenu, tout ou partie des relations orientées de la figure 4.18 peuvent aussi être décrites dans le descripteur de déploiement pour prise en charge par le serveur. Concernant les requêtes engendrées par des besoins précis, le langage EJB QL (*EJB Query Language*) se substitue à SQL de manière à les écrire hors du code, encore une fois. De plus, l'automatisation de la gestion des relations impose une *Local Home Interface* et une *Local Interface* (voir figure 4.17) pour tout EJB impliqué dans une relation gérée par le serveur J2EE. Par exemple, si l'on implante les EJB *Incarceration* et *Motif* en reproduisant entre eux et à l'exact, la contrainte d'intégrité

référentielle de la figure 4.18[1], on doit doter *Motif* d'une *Local Home Interface* et d'une *Local Interface*[2].

Il faut donc distinguer les contraintes d'intégrité référentielle existant en dur dans la base (figure 4.18), des navigations vues comme des services offerts aux clients, navigations qui gagnent à se calquer sur les contraintes OCL qui dictent en général la façon dont les objets se manipulent. À titre d'illustration, les services de navigation prévus de la figure 4.19, sont pour certains des dépendances fonctionnelles de la base (lien d'*Incarceration* à *Motif*) et pour d'autres non (lien d'*Incarceration* à *Affaire* : rôle *principale*).

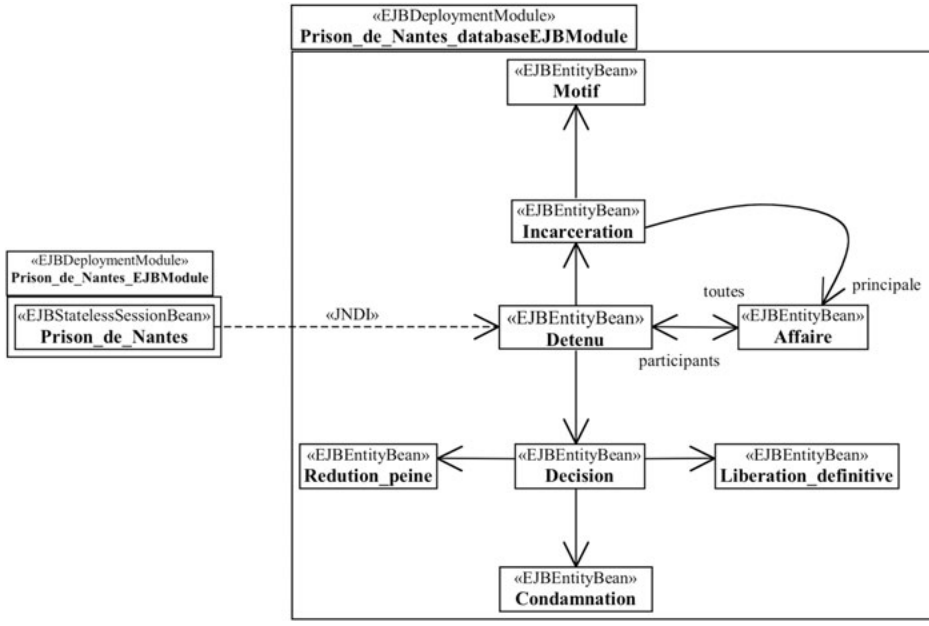


Figure 4.19 – Couplage relatif à l'implantation EJB (référencement effectué dans les *Entity Bean*).

La correspondance entre la figure 4.18 et la figure 4.19 se fait au niveau du descripteur de déploiement du module EJB appelé *Prison_de_Nantes_databaseEJB-Module* en figure 4.19, qui est par ailleurs vu comme un package UML. Le mécanisme de *mapping* est propre au serveur J2EE utilisé et donc non (encore ?) normé dans la spécification EJB 2.1.

Dans la figure 4.19, il est intéressant de remarquer la disparition de la table *Detenu_Affaire*, purement technique en l'occurrence, qui ne devient pas un EJB. En

1. Ce n'est pas obligatoire. On pourrait uniquement offrir au client une navigation de l'EJB *Motif* à l'EJB *Incarceration* alors que le schéma de base de données possède l'inverse.
2. Ce n'est pas exclusif. Il peut en plus posséder une *Home Interface* et une *Remote Interface* standard en vue d'être accédé par un EJB d'une autre machine virtuelle Java.

revanche, le schéma de dépendance des EJB de la figure 4.19 et la contrainte OCL qui suit, montre bien comment un client peut bénéficier de services adéquats (*i.e.* conformes aux besoins en l'occurrence et au modèle de la figure 4.6) s'il veut vérifier le fait que l'affaire principale qui a amené un détenu en prison, c'est-à-dire celle liée à son incarcération, est bien incluse dans toutes les affaires où ce dernier est impliqué. Au regard de la figure 4.19, cela s'écrit :

```
context Detenu inv i1:
   toutes→includesAll(incarceration.principale)
```

Les relations entre *Entity Bean* constituent donc un enjeu certain et ô combien intéressant. La spécification UML va ici « méthodiquement » jouer un rôle central dans la mesure où les contraintes OCL, voire des associations orientées dans les modèles empêchant certains sens de navigation, vont arrêter la manière dont les données doivent être exploitées (voir section 4.4.2). On peut ainsi envisager des référencements entre *Entity Bean* qui pratiquement s'opèrent via des adresses valides à travers tout le réseau. Pratiquement, des références (« pointeurs ») sur les *Remote Interface* (ou sur des *Local Interface* ce qui est suffisant si l'on se trouve dans la même machine virtuelle Java) sont suffisantes. En fait, les instances des classes implémentant ces interfaces (relation Java *implements*) et construites dynamiquement par le serveur, peuvent s'échanger à tout moment en exécution en vue d'accéder et de manipuler un EJB donné. En conséquence, les liens multiples, comme le rôle *participants* de l'EJB *Affaire* à l'EJB *Detenu* deviennent une méthode dont le type de retour est *Collection* ou un de ses sous-types en Java, *Set* étant le plus approprié dans l'esprit entité/relation. Les éléments contenus dans la collection seront alors des références sur des EJB de type *Détenu*.

*Exemple d'*Entity Bean

Le premier exemple ici fourni est le type UML *Affaire* traduit à la fois sous forme de table dans la base de données et comme *Entity Bean* de type CMP. Compte tenu que la clé est composée de deux champs (*n°affaire* et *nom juridiction*), une classe *AffaireKey* est créée et adjointe à l'EJB lui-même (figure 4.20). Autrement dit, lorsqu'un *Entity Bean* a pour clé un seul champ, celui-ci est usuellement de type *String* (ou un format numérique propre à Java comme *java.math.BigDecimal*) et il ne devient donc pas nécessaire que l'EJB ait une *Entity Bean primary key class* (cardinalité 0..1 en figure 4.17 : il a ou il n'a pas de classe clé primaire).

La classe *AffaireKey* va contractuellement implémenter en Java la règle qui dit que deux affaires sont différentes si et seulement si leur clé composée est différente. La méthode *equals* héritée d'*Object* est donc masquée ainsi que la fonction de hachage, aussi héritée d'*Object*, pour éviter des collisions au moment de l'indexation d'objets *AffaireKey* distincts :

```
public final class AffaireKey implements java.io.Serializable {
   public String nAffaire;
   public String nomJuridiction;
   // constructeurs ici donc un sans argument obligatoire
   public boolean equals(Object object) {
```

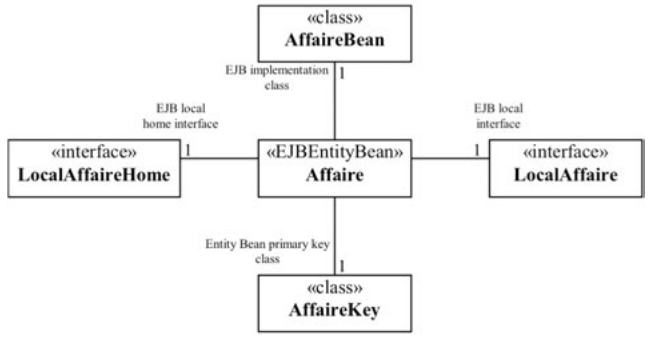


Figure 4.20 – EJB *Affaire* et ses constituants : exemple d'instanciation du métamodèle de la figure 4.17.

```
        if(this == object) return true;
        if(! (object instanceof AffaireKey)) return false;
        return ((nAffaire == null ? ((AffaireKey) object).nAffaire == null :
        ➥ nAffaire.equals(((AffaireKey) object).nAffaire)) && (nomJuridiction ==
        ➥ null ? ((AffaireKey) object).nomJuridiction == null :
        ➥ nomJuridiction.equals(((AffaireKey) object).nomJuridiction)));
        }
    public int hashCode() {
        return ((nAffaire == null ? 0 : nAffaire.hashCode()) ^ (nomJuridiction
        ➥ == null ? 0 : nomJuridiction.hashCode()));
    }
}
```

La gestion automatique des liens entre *Entity Bean* propre au mode CMP impose la définition d'une *Local Home Interface* et d'une *Local Interface* pour tout *Entity Bean* de type CMP. Le premier type d'interface permet au serveur J2EE de dériver une classe concrète qui implémente des services de création et de localisation (*finder method*) d'instances d'*Affaire* :

```
public interface LocalAffaireHome extends javax.ejb.EJBLocalHome {
    public LocalAffaire findByPrimaryKey(AffaireKey affaireKey) throws
        ➥ javax.ejb.FinderException;
    public LocalAffaire create(final String nAffaire,final String
        ➥ nomJuiridiction,final java.sql.Date dateFaits) throws
        ➥ javax.ejb.CreateException;
}
```

Dans le code qui précède, la méthode *findByPrimaryKey* est obligatoire. Elle charge une instance d'*Affaire* (si trouvée) en fonction d'une ligne dans une table. Dans le mode CMP, tout est automatique et la fonction *ejbLoad* est par essence vide (voir classe d'implémentation *AffaireBean* ci-dessous). En mode BMP, le programmeur doit coder le chargement sur la base des technologies JDBC ou JDO. Quant à la méthode *create* à trois arguments, elle est typiquement appelée après activation du bouton *Submit* de la figure 4.15. Outre la création d'une instance d'EJB *Affaire*, une ligne est automatiquement insérée dans la table *Affaire*. Encore une fois, les descrip-

teurs de déploiement donnent au serveur J2EE toutes les informations nécessaires à la réalisation de telles actions.

Quatre catégories de services existent pour les *Entity Bean* :

- *System*
- *Finder/Creation*
- *Selector*
- *Business*

La première catégorie est régie par la norme EJB elle-même (méthodes *setEntityContext*, *ejbLoad*, etc.). Un bon générateur de code peut les prendre en charge pour s'assurer qu'un EJB est conforme à la norme. La seconde est illustrée ci-avant. Les deux dernières sont à la demande. Par exemple, s'il n'y en a pas pour l'EJB *Affaire*, sa *Local Interface* va être vide :

```
public interface LocalAffaire extends javax.ejb.EJBLocalObject {
}
```

Finalement, l'implémentation (*i.e.* ce qui est réellement à la charge du programmeur) de l'EJB *Affaire* se réduit à très peu de chose : les méthodes *System* sont vides, les relations entre EJB désirées, selon ce qui apparaît en figure 4.19, sont signées dans l'implémentation mais sont abstraites. Encore une fois, via l'héritage cette fois-ci, le serveur génère une classe concrète qui implémente ces fonctions selon les informations portées par le descripteur de déploiement d'*Affaire*.

```
public abstract class AffaireBean implements javax.ejb.EntityBean {
   private javax.ejb.EntityContext _ejb_context;
   public void setEntityContext(javax.ejb.EntityContext ejb_context) {
   _ejb_context = ejb_context;
   }
   public void unsetEntityContext() {
   _ejb_context = null;
   }
   public void ejbActivate() { // en mode CMP, cette zone est a priori
                               // par définition vide
   }
   public void ejbPassivate() { // en mode CMP, cette zone est a priori
                                // par définition vide
   }
   public void ejbRemove() { // en mode CMP, cette zone est a priori
                             // par définition vide
   }
   public void ejbLoad() { // en mode CMP, cette zone est par définition vide
   }
   public void ejbStore() { // en mode CMP, cette zone est par définition vide
   }
   public abstract String getNAffaire();
   public abstract void setNAffaire(String nAffaire);
   public abstract String getNomJuridiction();
   public abstract void setNomJuridiction(String nomJuridiction);
   public abstract java.sql.Date getDateFaits();
   public abstract void setDateFaits(java.sql.Date dateFaits);
```

```
    // On a juste ci-dessous la matérialisation de la relation orientée
    //(rôle participants) d'Affaire à Detenu de la figure 4.19 :
    public abstract java.util.Collection getParticipants();
    public abstract void setParticipants(java.util.Collection participants);
    // Des méthodes de création et post-création associées à la méthode create
    // de la Local Home Interface sont appelées automatiquement par le serveur J2EE
    public AffaireKey ejbCreate(final String nAffaire,final String
       ➥ nomJuridiction,final java.sql.Date dateFaits) throws
       ➥ javax.ejb.CreateException {
    return new AffaireKey(nAffaire,nomJuridiction);
   }
    public void ejbPostCreate(final String nAffaire,final String
       ➥ nomJuridiction,final java.sql.Date dateFaits) throws
       ➥ javax.ejb.CreateException {
    }
    // on note ici l'absence de ejbFindByPrimaryKey due au mode CMP
}
```

En bilan, le cycle de vie d'un *Entity Bean* (BMP ou CMP) apparaît en figure 4.21. Les opérations *create*, *remove* et *findByPrimaryKey* (et toutes les opérations *Finder* plus généralement) sont accessibles sur la *Home Interface* (et/ou *Local Home Interface*).

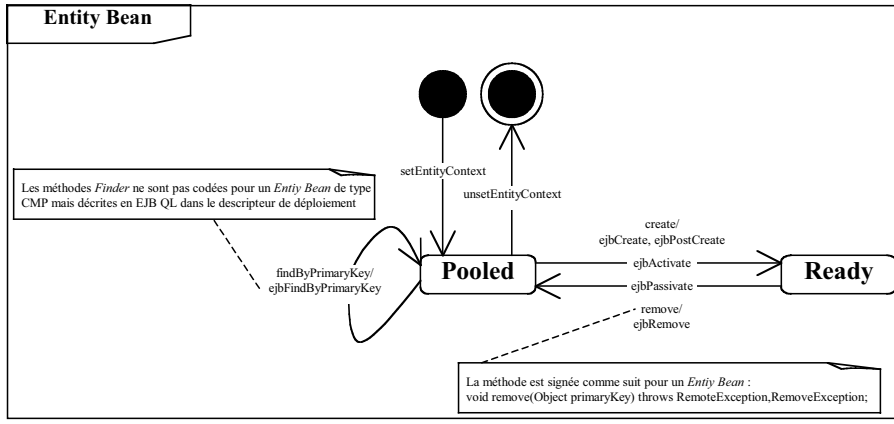


Figure 4.21 – Cycle de vie d'un *Entity Bean*.

Le statechart en figure 4.21 indique les deux états clés d'un *Entity Bean* : *Pooled* et *Ready*. L'état *Pooled* caractérise en particulier la fin du déploiement d'un EJB (sortie de l'état initial, point noir UML). L'état *Ready* est le seul état dans lequel les *Business* opérations sont activables. Il y a une activation système (sortie de l'état *Pooled*, méthode *ejbActivate*) au préalable si l'instance était passive. Dans le même esprit, la méthode *ejbPassivate* permet au serveur J2EE d'assurer l'équilibrage de charges si un EJB tend à ne pas être utilisé, et donc devient ponctuellement inutile. *ejbCreate*, *ejbPostCreate* et *ejbRemove* quant à elles, sont indirectement initiées par le client. En fait, *create* et *remove* sont lancées, comme nous l'avons dit, sur la *Home Interface* ou la *Local Home Interface*. En mode BMP, le codage de ces trois fonctions (idem pour

ejbActivate et *ejbPassivate*) sert surtout à la gestion des ressources (allocation, respectivement libération) comme par exemple l'ouverture, respectivement la fermeture, de ports ou de connexions.

Pour terminer avec l'EJB *Affaire*, on remarque dans son descripteur de déploiement écrit en XML et figurant ci-dessous que la propriété *persistence-type* est bien entendu *Container* (*i.e.* CMP). Les propriétés de mapping avec la base de données ne sont pas gérées au niveau EJB mais au niveau EJB module. Un EJB module (voir *Prison_de_Nantes_databaseEJBModule* dans la figure 4.19 vu comme un package UML) regroupe plusieurs EJB déployés et donc configurés ensemble.

```
<?xml version="1.0" encoding="UTF-8"?>
<ejb-jar>
  <enterprise-beans>
    <entity>
      <display-name>Affaire</display-name>
      <ejb-name>Affaire</ejb-name>
      <local-home>com.FranckBarbier.Java._Prison_de_Nantes.LocalAffaireHome<
      ➥ /local-home>
      <local>com.FranckBarbier.Java._Prison_de_Nantes.LocalAffaire</local>
      <ejb-class>com.FranckBarbier.Java._Prison_de_Nantes.AffaireBean<
      ➥ /ejb-class>
      <persistence-type>Container</persistence-type>
      <prim-key-class>com.FranckBarbier.Java._Prison_de_Nantes.AffaireKey<
      ➥ /prim-key-class>
      <reentrant>False</reentrant>
      <abstract-schema-name>Affaire</abstract-schema-name>
      <cmp-field>
        <field-name>nAffaire</field-name>
      </cmp-field>
      <cmp-field>
        <field-name>nomJuridiction</field-name>
      </cmp-field>
      <cmp-field>
        <field-name>dateFaits</field-name>
      </cmp-field>
    </entity>
  </enterprise-beans>
</ejb-jar>
```

Relations entre Entity Bean

Le second exemple d'*Entity Bean* que nous nous proposons d'étudier est *Detenu* parce qu'il est choisi comme point d'entrée dans la figure 4.19. Outre des compléments d'information sur les *Entity Bean*, nous allons illustrer via son intermédiaire, les relations entre *Entity Bean*.

La *Local Home Interface* de *Detenu* offre une méthode *Finder* retournant une collection de tous les détenus en préventive, c'est-à-dire, rappelons-le, tous les détenus pour lesquels aucune décision de condamnation n'a été prononcée.

```
public LocalDetenu findByPrimaryKey(String nEcrou) throws
    ➥ javax.ejb.FinderException;public interface LocalDetenuHome extends
    ➥ javax.ejb.EJBLocalHome {
```

```
    public java.util.Collection findPreventive() throws
       ➥ javax.ejb.FinderException;
}
```

Le mode CMP s'appuie sur le langage EJB QL pour indiquer au serveur J2EE comment implémenter la méthode *findPreventive*. La requête EJB QL, *SELECT OBJECT(d) FROM Detenu d WHERE d.decision.condamnation IS EMPTY*, s'appuie sur les navigations retenues en figure 4.19. Sa formulation est très proche de celle d'OCL. L'obligation ici est de véritablement implanter les navigations *decision* et *condamnation* (voir encore figure 4.19) entre les EJB, alors que dans un modèle UML, toutes les navigations sont *a priori* utilisables.

On a ici définitivement la preuve que le type d'objet *Preventive* (figure 4.5) n'est pas nécessaire. Il faut en toute honnêteté néanmoins remarquer qu'un tel constat n'est faisable qu'*a posteriori*. Beaucoup d'applications alternatives à celle que nous proposons auraient pu l'incarner en dur sous forme de table puis d'EJB, sans pour autant créer de bogues. Toujours le même leitmotiv donc : la qualité rien que la qualité. Maintenir le composant logiciel *Preventive* a un coût. Ne pas l'avoir en montrant pratiquement qu'il ne sert à rien, est source d'une meilleure maintenabilité ; voilà pour la leçon.

Dans le même esprit, l'invariant OCL *i1* écrit ci-avant et adossé à la figure 4.19, peut facilement et directement être instrumenté pour son contrôle ultérieur et à la demande. Pour cela, les navigations de *Detenu* à *Incarceration* (rôle *incarceration*) et de *Detenu* à *Affaire* (rôle *toutes*) sont disponibles dans l'implémentation de l'EJB *Detenu* :

```
public abstract LocalIncarceration getIncarceration();
public abstract void setIncarceration(LocalIncarceration incarceration);
public abstract java.util.Collection getToutes();
public abstract void setToutes(java.util.Collection toutes);
```

Il suffit alors de créer une *Business* méthode nommée *i1* (la méthode *principale* est elle-même une *Business* méthode de l'EJB *Incarceration* puisque comme le montre le code qui suit, elle est déclarée sur l'interface locale *LocalIncarceration*) :

```
public boolean i1() {
   return getToutes().contains(getIncarceration().principale());
}
```

C'est en fait dans l'EJB module comportant tous les EJB de type CMP déployés dans la même machine virtuelle Java, que figurent (dans son descripteur de déploiement exactement) les informations sur les relations entre les EJB. Dans le texte XML qui suit, la relation de *Detenu* à *Incarceration* (rôle *incarceration*) est décrite par la propriété *cmr-field*. Bien que nommée (rôle *detenu*), la relation d'*Incarceration à Detenu* est, elle, non gérée par un champ dédié (pas de propriété *cmr-field* valuée) en conformité avec la figure 4.19 encore une fois : on ne désire pas de navigation d'*Incarceration* à *Detenu*.

```
<relationships>
    <ejb-relation>
```

```
        <ejb-relation-name>Detenu-Incarceration</ejb-relation-name>
        <ejb-relationship-role>
          <ejb-relationship-role-name>incarceration</ejb-relationship-role-name>
          <multiplicity>One</multiplicity>
          <relationship-role-source>
            <ejb-name>Detenu</ejb-name>
          </relationship-role-source>
          <cmr-field>
            <cmr-field-name>incarceration</cmr-field-name>
          </cmr-field>
        </ejb-relationship-role>
        <ejb-relationship-role>
          <ejb-relationship-role-name>detenu</ejb-relationship-role-name>
          <multiplicity>One</multiplicity>
          <relationship-role-source>
            <ejb-name>Incarceration</ejb-name>
          </relationship-role-source>
        </ejb-relationship-role>
      </ejb-relation>
      <ejb-relation>
      …
</relationships>
```

Exemple de Session Bean

L'esprit des *Session Bean* est d'implanter la logique métier en amont des *Entity Bean*, c'est-à-dire d'assurer les interactions avec les clients qui mènent à créer une synergie pour calculer, en particulier, de la nouvelle information. En d'autres termes, les *Session Bean* n'étant pas persistants (ils n'ont pas de classe clé primaire, figure 4.17), ils incarnent les processus de gestion de l'information (*business process, workflow…*) et collaborent avec les *Entity Bean* pour partager, stocker, assurer la cohérence des données traitées.

Il existe deux catégories de *Session Bean* : *stateful* et *stateless* (figure 4.17). Ce statut est figé et est défini au déploiement. Bien que *stateless* signifie sans état, cela n'empêche pas un EJB de cette nature de posséder des attributs par exemple. Néanmoins, les valeurs de ces attributs ne sont vraiment utilisables par un client que le temps d'exécution d'une *Business* méthode. Aucun état conversationnel n'est maintenu entre l'appel de deux opérations différentes ou lors de l'appel successif de la même opération. Un *Session Bean stateless* est donc d'un usage souple et surtout peu coûteux car il peut être facilement et rapidement affecté par le serveur d'EJB aux clients requérant les opérations qu'il supporte. Au contraire, un *Session Bean stateful* conserve un état entre deux requêtes. Il est donc dédié à un seul client durant son cycle de vie, ce dernier bénéficiant de l'exploitation éventuelle de résultats intermédiaires qu'il aurait été amené à calculer auparavant.

Il résulte foncièrement de la dichotomie *stateful/stateless* que la création et la destruction d'un EJB *stateless* relèvent d'une décision du serveur J2EE. Ce dernier régule la charge induite par les clients en partageant des *Session Bean stateless* entre plusieurs clients ou en en créant de nouveaux pour améliorer la qualité de service. Au

contraire, les *Session Bean stateful* sont créés et détruits par le client les nécessitant. Il en résulte les cycles de vie standard de la figure 4.22.

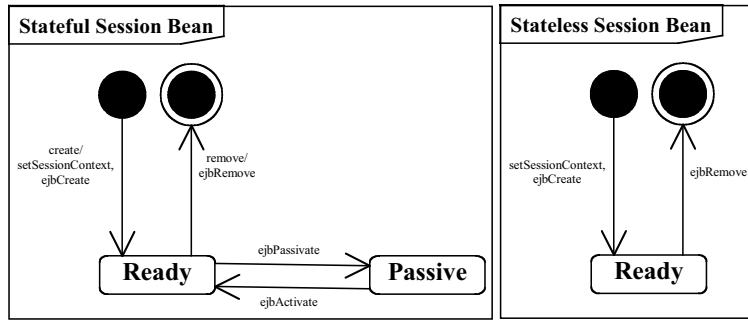


Figure 4.22 – Cycle de vie d'un *Session Bean*.

Compte tenu de ces propriétés, l'EJB *Prison_de_Nantes* (figure 4.19) qui émane du type d'objet *Prison de Nantes* de la figure 4.6 est choisi *stateless*. Pour le justifier, considérons que cet EJB ne rend que le service de détermination des prisonniers en préventive. Un client de *Prison_de_Nantes*, un service Web typiquement, n'a pas besoin d'état conversationnel. À la demande, on obtient la liste de ces détenus particuliers. La classe d'implémentation de l'EJB *Prison_de_Nantes* (*Prison_de_NantesBean*) va donc tout d'abord prendre en charge la localisation de l'EJB *Detenu* via la technologie JNDI symbolisée par le stéréotype *«JNDI»* sur la dépendance entre *Prison de Nantes* et *Detenu* dans la figure 4.19. Cette localisation a lieu à la création et à l'activation de l'EJB sachant qu'à sa destruction et à sa passivation, le contexte de nommage est fermé. Ensuite, la *Business* méthode *preventive* ne fait qu'assurer le service attendu des clients en appelant la méthode *findPreventive* préalablement évoquée et supportée par la *Local Home Interface* de *Detenu*.

```
public class Prison_de_NantesBean implements javax.ejb.SessionBean {
   /** JNDI */
   protected javax.naming.Context _jndi_context = null;
   protected LocalDetenuHome _localDetenuHome = null;
   /** EJB */
   protected javax.ejb.SessionContext _ejb_context;
   public void setSessionContext(javax.ejb.SessionContext ejb_context) {
   _ejb_context = ejb_context;
   }
   public void ejbActivate() {
   try {
   _jndi_context = new javax.naming.InitialContext();
   _localDetenuHome = (LocalDetenuHome)_jndi_context.lookup(
      ➥ "java:comp/env/ejb/Detenu");
   }
     catch(javax.naming.NamingException ne) {
           System.err.println("ejbActivate(): " + ne.getClass().toString()
     ➥ + ", " + ne.getExplanation());
     }
   }
```

```
    public void ejbPassivate() {
    try {
    jndi_context.close();
    }
    catch(javax.naming.NamingException ne) {
          System.err.println("ejbPassivate(): " + ne.getClass().toString()
       ➥ + ", " + ne.getExplanation());
    }
    }
    public void ejbRemove() {
    try {
       _jndi_context.close();
    }
    catch(javax.naming.NamingException ne) {
    System.err.println("ejbRemove(): " + ne.getClass().toString() + ", "
       ➥ + ne.getExplanation());
    }
    }
    public void ejbCreate() throws javax.ejb.CreateException {
    try {
    _jndi_context = new javax.naming.InitialContext();
    _localDetenuHome = (LocalDetenuHome)_jndi_context.lookup(
       ➥ "java:comp/env/ejb/Detenu");
    }
    catch(javax.naming.NamingException ne) {
    System.err.println("ejbCreate(): " + ne.getClass().toString() + ", "
       ➥ + ne.getExplanation());
    }
    }
    public java.util.Collection preventive() {
    java.util.Collection preventive = null;
    try {
    preventive = _localDetenuHome.findPreventive();
    }
    catch(javax.ejb.FinderException fe) {
    System.err.println("preventive(): " + fe.getMessage());
    }
    return preventive;
    }
}
```

On rappelle avec insistance ici que les interfaces *Local Home Interface* et *Local Interface* sont dédiées aux EJB qui cohabitent dans la même machine virtuelle Java. Dans l'absolu, un EJB donné peut posséder à la fois des interfaces des deux types précités ainsi qu'une interface *Home* et une interface *Remote* « plus standard ». Intuitivement, les services qu'offrent les interfaces qualifiées de « locales » sont réalisés avec un coût moindre puisque les EJB ne sont pas réellement distribués sur le réseau. L'usage ici intensif des interfaces « locales » est contraint par le mode CMP. On pourrait donc remarquer que l'EJB *Detenu* étant le point d'entrée de la base de données de la prison de Nantes, il aurait grandement bénéficié d'une interface *Home* et d'une interface *Remote*. En effet, le code qui précède impose que le *Session Bean Prison_de_Nantes* soit dans la même machine virtuelle Java que *Detenu*, ce qui est limitatif.

4.8 CONCLUSION

Dans ce chapitre, nous avons caractérisé le processus d'utilisation d'UML dans le contexte d'une étude de cas où les aspects statiques sont dominants. Le *Class Diagram* (figure 4.6) est dans un tel cadre un modèle difficile à bâtir alors que les aspects dynamiques sont restreints (flux de contrôle limité à un échange avec une interface graphique qui fait vivre le système d'information). Les traitements peuvent être en partie spécifiés en OCL, pour en particulier décrire les navigations dont on a besoin sur le *Class Diagram*, puis sur des modèles de type conception comme l'architecture de dépendance des EJB (figure 4.19). Effectivement, ce dernier modèle restreint de façon coercitive ce qu'il est réellement possible de faire dans la technologie des EJB.

Avant d'en arriver là, le cheminement est long. L'essentiel de la difficulté réside en effet dans la phase d'ingénierie des besoins où ici, le discours du directeur de la prison de Nantes est l'archétype de l'interview truffée de pièges. Il en résulte que même si UML, en tant que langage de modélisation, n'est pas une contribution significative à l'approche entité/relation (on fait avec UML ce que l'on savait faire depuis longtemps avec d'autres outils comme NIAM), une bonne maîtrise d'UML par l'analyste ainsi qu'une utilisation parallèle judicieuse et poussée d'OCL mènent à des modèles de bonne tenue. Cela est en particulier vérifié par la possibilité de facilement dériver un *Class Diagram* sous forme de schéma relationnel (figure 4.14 en Access et dans la section qui suit concernant Oracle).

L'implantation EJB est une suite naturelle pour fournir entre le système d'information géré via un serveur J2EE et les programmes clients, une cloison logicielle où est incarnée la logique métier (principe des architectures client/serveur dites « à trois niveaux » et des intergiciels plus généralement). Cela offre un haut niveau de réutilisabilité au travers d'*Entity Bean* et de *Session Bean*, qui proposent aux programmes client toutes les fonctionnalités standard et récurrentes de gestion de la prison de Nantes. En effet, la dichotomie développement/déploiement, native dans les EJB, est d'une souplesse extraordinaire pour dissocier dans le code les aspects purement métier des aspects techniques, certes critiques comme la sécurité ou encore les transactions mais à même d'être automatiquement paramétrés et pris en charge par le serveur d'applications via les descripteurs de déploiement. Finalement, le problème ne réside plus que dans le fait de trouver une technique de conception des EJB efficace, notamment l'identification des *Session Bean* par exemple, leur statut *stateful* ou *stateless*, ainsi que leurs relations entre eux et avec les *Entity Bean*. À ce titre, le mode CMP est le plus flexible et le plus puissant car il supporte la gestion des relations entre *Entity Bean* au niveau du serveur, réduisant de fait les lignes de code nécessaires pour gérer les allers/retours avec la base de données.

4.9 CODE ORACLE DE L'ÉTUDE DE CAS

Le programme de création de la base de données est le suivant :

```
create schema authorization barbier /* le nom de schéma ne peut être que le
login avec Oracle */
create table Detenu(
   n_ecrou Char(10),
   prenom Char(30),
   nom Char(30),
   date_naissance Date,
   lieu_naissance Char(30),
   constraint Detenu_key primary key(n_ecrou))
create table Affaire(
   n_affaire Char(10),
   nom_juridiction Char(30),
   date_faits Date,
   constraint Affaire_key primary key(n_affaire,nom_juridiction))
create table Detenu_Affaire(
   n_ecrou Char(10),
   n_affaire Char(10),
   nom_juridiction Char(30),
   constraint Detenu_Affaire_key primary key(n_ecrou,n_affaire,nom_juridiction),
constraint Detenu_Affaire_foreign_key foreign key(n_ecrou) references
Detenu(n_ecrou),
   constraint Detenu_Affaire_foreign_key2 foreign key(n_affaire,nom_juridiction)
references Affaire(n_affaire,nom_juridiction))
create table Motif(
   n_motif Char(10),
   libelle_motif Char(50),
   constraint Motif_key primary key(n_motif),
   constraint Motif_unique unique(libelle_motif))
create table Incarceration(
   n_ecrou Char(10),
   n_affaire Char(10),
   nom_juridiction Char(30),
   date_incarceration Date,
   n_motif Char(10),
   constraint Incarceration_key primary key(n_ecrou),
   constraint Incarceration_foreign_key foreign
key(n_ecrou,n_affaire,nom_juridiction) references
Detenu_Affaire(n_ecrou,n_affaire,nom_juridiction),
   constraint Incarceration_foreign_key2 foreign key(n_motif) references
Motif(n_motif))
create table Decision(
   n_type_decision Char,
   n_ecrou Char(10),
   date_decision Date,
   constraint Decision_key primary key(n_type_decision,n_ecrou,date_decision),
   constraint Decision_fk foreign key(n_ecrou) references Detenu(n_ecrou))
create table Reduction_peine(
   n_type_decision Char,
   n_ecrou Char(10),
   date_decision Date,
   duree Integer,
   constraint Reduction_peine_key primary
key(n_type_decision,n_ecrou,date_decision),
   constraint Reduction_peine_fk foreign
key(n_type_decision,n_ecrou,date_decision) references
Decision(n_type_decision,n_ecrou,date_decision))
create table Liberation_definitive(
```

```
   n_type_decision Char,
   n_ecrou Char(10),
   date_decision Date,
   constraint Liberation_definitive_key primary
key(n_type_decision,n_ecrou,date_decision),
   constraint Liberation_definitive_fk foreign
key(n_type_decision,n_ecrou,date_decision) references
Decision(n_type_decision,n_ecrou,date_decision))
create table Condamnation(
   n_type_decision Char,
   n_ecrou Char(10),
   date_decision Date,
   duree Integer,
   constraint Condamnation_key primary
key(n_type_decision,n_ecrou,date_decision),
    constraint Condamnation_fk foreign key(n_type_decision,n_ecrou,date_decision)
references Decision(n_type_decision,n_ecrou,date_decision));
```

Le programme de destruction de la base est le suivant :

```
drop table Detenu cascade constraints;
drop table Affaire cascade constraints;
drop table Detenu_Affaire cascade constraints;
drop table Motif cascade constraints;
drop table Incarceration cascade constraints;
drop table Decision cascade constraints;
drop table Reduction_peine cascade constraints;
drop table Liberation_definitive cascade constraints;
drop table Condamnation cascade constraints;
```

4.10 BIBLIOGRAPHIE

1. Armstrong, E., Ball, J., Bodoff, S., Carson, D., Evans, I., Fisher, M., Green, D., Haase, K., and Jendrock, E. : *The J2EE 1.4 Tutorial*, Sun Microsystems (2003)
2. Cook, S., and Daniels, J. : *Designing Object Systems – Object-Oriented Modelling with Syntropy*, Prentice Hall (1994)
3. Gamma, E., Helm, R., Johnson, R., and Vlissides, J. : *Design Patterns – Elements of Reusable Object-Oriented Software*, Addison-Wesley (1995)
4. Habrias, H. : *Le modèle relationnel binaire – méthode I.A. (NIAM)*, Eyrolles (1988)
5. Halpin, T., and Bloesch, A. : "Data modeling in UML and ORM: a comparison", *Journal of Database Management*, 10(4), (1999), 4-13
6. Roman, S. : *Access Database – Design & Programming*, Second Edition, O'Reilly (1999)
7. Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., and Lorensen, W. : *Object-Oriented Modeling and Design*, Prentice Hall (1991)
8. Sun Microsystems : *Enterprise JavaBeans Specification, Version 2.1* (2003)
9. Wirfs-Brock, R., Wilkerson, B., and Wiener, L. : *Designing Object-Oriented Software*, Prentice Hall (1990)

10. Tabourier, Y. : *De l'autre côté de Merise – Systèmes d'information et modèles d'entreprise*, Les Editions d'Organisation (1986)

4.11 WEBOGRAPHIE

ORM, site officiel du langage de modélisation *Object Role Modeling* : www.orm.net
Serveur d'EJB BEA WebLogic : www.bea.com/products/weblogic/index.html
Serveur d'EJB JBoss : www.jboss.org
Serveur d'EJB Jonas : www.evidian.com/jonas
Sources de l'étude de cas : www.PauWare.com
Technologie EJB : java.sun.com/products/ejb

5

Système de domotique

5.1 INTRODUCTION

L'étude de cas qui suit est extraite du livre de Rumbaugh *et al.* [7]. Elle a été étendue et corrigée pour respectivement gagner en crédibilité et pour remédier aux erreurs de spécification qui figurent dans l'ouvrage précité. Nous revenons sur ce dernier point en relevant les principales fautes de modélisation. Par ailleurs, nous exposons des variantes de spécification pour montrer la faisabilité de plusieurs modèles. Parfois l'accroissement d'une qualité comme l'évolutivité peut se faire au détriment d'une autre qualité, par exemple la performance. Un arbitrage se justifie donc souvent. Il n'existe jamais « un » modèle solution mais plusieurs, chacun présentant des avantages et des inconvénients. Nous insistons sur une qualité rarement prise en charge : la capacité d'un modèle à être facilement implanté. Cela ne signifie pas que la spécification n'est qu'une représentation graphique de ce que sera la version codée du système en C++, Java ou tout autre langage objet. Bien au contraire, nous prouvons l'indépendance de la spécification ainsi que son aptitude à être déclinée sans heurt vers tout d'abord, un modèle de conception par nature teinté de considérations opérationnelles, puis du code, tout cela dans l'esprit du MDE.

Les caractéristiques de cette étude de cas sont le fort usage du cadencement, c'est-à-dire de services de *Timer* qui, du fait de leur sophistication, ont conduit à l'élaboration d'un patron spécifique et de son outillage en Java. Nous traitons par ailleurs brièvement de la manière dont un composant logiciel, s'il requiert des services de *Timer* via sa *required interface*, peut s'octroyer de tels services. Finalement, du fait que l'application est fortement axée sur les *State Machine Diagram*, nous montrons l'architecture et la mise en fonctionnement de la bibliothèque *PauWare.Statecharts*, qui automatise l'implémentation de ces types de diagramme UML 2.x. Bien entendu, l'étude de cas utilise aussi les *Class Diagram* et de façon plus anecdotique les *Communication Diagram*. Nous voyons que dans le domaine de la dynamique, les types de diagramme autres que les *State Machine Diagram* sont plutôt accessoires.

Sans faire office de preuve, ce constat confirme l'avis exposé dans le chapitre 3 sur la qualité de ces derniers.

5.2 CAHIER DES CHARGES

Un thermostat programmable contrôle un chauffage et un conditionneur d'air en fonction de données entrées, à l'aide d'un pavé de dix boutons activé par un utilisateur. En fonctionnement, le thermostat agit soit sur le chauffage, soit sur le conditionneur d'air, pour faire converger la température ambiante vers la température cible courante. Au chauffage et au conditionneur d'air s'ajoute un ventilateur également contrôlé par le thermostat. En fonctionnement, le thermostat contrôle chacun de ces trois éléments par le biais de trois relais de puissance. Le thermostat établit la température ambiante à l'aide d'un capteur de température. Le cycle et le mode de rafraîchissement de cette information ne sont pas connus car ils dépendent du dispositif retenu pour l'installation finale. En effet, en fonction de contraintes matérielles et logicielles ultérieurement définies (type de capteur, protocole de communication, etc.), la spécification doit être suffisamment flexible pour s'adapter à ces contraintes. Le thermostat a besoin à tout moment de quatre informations : la température ambiante, l'heure et le jour courants, la température cible courante, ainsi que la constante « delta », la condition d'arrêt du chauffage étant exprimée par « température ambiante > température cible courante + delta », et la condition d'arrêt du conditionneur d'air étant exprimée par « température ambiante < température cible courante – delta ».

Le ventilateur est piloté par un commutateur comportant deux modes : le mode forcé entraîne le fonctionnement du ventilateur quelle que soit la situation. Le mode automatique, au contraire, n'entraîne la mise en marche du ventilateur que si le chauffage ou le conditionneur d'air se lance. En mode automatique, le ventilateur s'arrête également si le chauffage ou le conditionneur d'air s'arrête.

La température cible est décrite dans une table de huit valeurs préalablement saisies par l'utilisateur. Avec chaque valeur est enregistrée une heure. Par convention, les quatre premières heures enregistrées décrivent les instants d'une journée de la semaine auxquels la température cible va changer. Les quatre dernières heures décrivent les instants d'une journée du week-end auxquels la température cible va changer. Exemple :

- Période de programmation n° 1, heure : 22 heures, température cible : 21 °C à partir de 22 heures
- n° 2, 7 heures, 14 °C à partir de 7 heures
- n° 3, 9 heures, 22 °C à partir de 9 heures
- n° 4, 17 heures, 18 °C à partir de 17 heures
- n° 5, 23 heures, 22 °C à partir de 23 heures

- n° 6, 9 heures, 14 °C à partir de 9 heures
- n° 7, 13 heures, 22 °C à partir de 13 heures
- n° 8, 18 heures, 19 °C à partir de 18 heures

Le thermostat dépend d'une interface utilisateur intégrant le pavé de dix boutons. Un témoin d'activité, un commutateur de saison et un commutateur de ventilation font également partie de l'interface utilisateur. Chaque commutateur accepte une valeur de paramètre parmi deux ou trois possibilités. Les commutateurs et leurs valeurs de paramètre se présentent ainsi :

- commutateur de saison : *Heat* (le thermostat ne contrôle que le chauffage pour réguler la température ambiante), *Cool* (le thermostat ne contrôle que le conditionneur d'air pour réguler la température ambiante) ou *Off*. Dans ce dernier cas, le chauffage et le conditionneur d'air sont arrêtés et ne peuvent être remis en fonctionnement qu'en sortant du mode *Off* ;
- commutateur de ventilation : *On* (le ventilateur opère en permanence) ou *Auto* (le ventilateur n'opère que si le chauffage est actif ou le conditionneur d'air est actif).

Le témoin d'activité ne s'allume que si le chauffage est actif ou si le conditionneur d'air est actif.

L'interface utilisateur est dotée d'un système alternant toutes les deux secondes l'affichage de l'heure et du jour courants avec l'affichage de la température ambiante, ainsi qu'un système d'affichage de la température cible courante (figure 5.1). C'est le mode d'activité de programme (état *Operate*) dans lequel le thermostat vérifie seconde par seconde si l'heure courante est égale à l'instant auquel la température cible doit changer. Dans un tel cas, il faut passer à la période de programmation suivante et fixer la nouvelle température cible courante à partir de la table. Il est cependant possible de ne pas tenir compte du programme en maintenant jusqu'à nouvel ordre la température cible courante (bouton *Hold temp* de l'interface utilisateur). Le mode alternatif au mode d'activité de programme est le mode de configuration/programmation (état *Setup*). Les données affichées par l'interface utilisateur dépendent alors du fait que l'on configure le jour et l'heure courants du thermostat ou que l'on programme la table avec de nouvelles températures cibles à des créneaux horaires précis. L'affichage présente l'aspect suivant.

Dans la figure 5.1, les quatre lignes correspondent à quatre possibilités de visualisation. Dans une ligne donnée, il y a deux zones d'affichage. Selon l'état courant du thermostat programmable, ces zones sont réservées à l'affichage de données bien précises. En ce sens, nous avons mentionné en italique, pour information, l'état dans lequel se trouverait le thermostat pour être en mesure d'afficher le type de données. Par exemple, la première ligne en haut de la figure 5.1 fait apparaître à gauche, le jour et l'heure courants et à droite, la température cible courante à atteindre. C'est seulement dans l'état *Operate* et son sous-état *Target temperature displaying* que cela est possible.

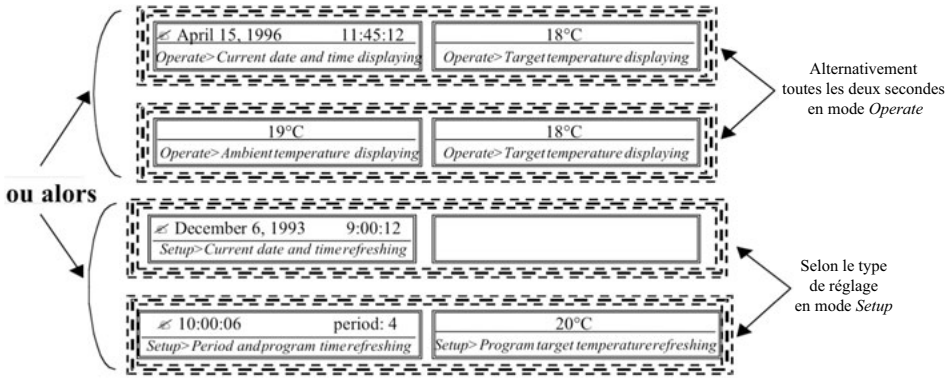


Figure 5.1 – Interface utilisateur.

Chacun des dix boutons du pavé génère un événement chaque fois qu'il est poussé. Il y a donc un type d'événement par bouton :

- *Temp up* : dans le mode d'activité de programme, ce type d'événement incrémente et modifie la température cible courante qui est affichée. Dans le mode de configuration/programmation, ce type d'événement incrémente la température cible d'une période de programmation donnée sans dépasser 90 °F. On passe d'une température cible à une autre dans la table en appuyant sur la touche *View program* (*cf.* ci-après) ;
- *Temp down* : dans le mode d'activité de programme, ce type d'événement décrémente et modifie la température cible courante qui est affichée. Dans le mode de configuration/programmation, ce type d'événement décrémente la température cible d'une période de programmation donnée sans dépasser 40 °F. On passe d'une température cible à une autre dans la table en appuyant sur la touche *View program* (*cf.* ci-après) ;
- *Time forward* : cette touche n'est utilisable que dans le mode configuration/programmation. En plus de la possibilité d'incrémenter l'heure courante (*cf. Set clock*), elle permet d'incrémenter l'heure associée à une période de programmation donnée de quinze secondes à chaque appui (*cf. View program*) ;
- *Time backward* : cette touche n'est utilisable que dans le mode configuration/programmation. En plus de la possibilité de décrémenter l'heure courante (*cf. Set clock*), cette touche permet de décrémenter l'heure associée à une période de programmation donnée de quinze secondes à chaque appui (*cf. View program*) ;
- *Set clock* : en appuyant sur cette touche, on entre dans le mode de configuration/programmation et l'on quitte donc le mode d'activité de programme. Dans cet état, en appuyant de nouveau sur la touche *Set clock*, on affiche les heures si les minutes étaient affichées, on affiche les minutes (affichées par défaut en entrant dans l'état) si les heures étaient affichées. On incrémente ou

décrémente les heures ou les minutes affichées en appuyant respectivement sur les touches *Time forward* ou *Time backward*. De manière générale, la touche *Set clock* permet de modifier l'heure courante du thermostat. Au bout de quatre-vingt-dix secondes sans touche appuyée, on revient dans le mode d'activité de programme et plus spécialement dans l'état *Hold* (*cf.* bouton *Hold temp*) ;

- *Set day* : en appuyant sur cette touche, on entre dans le mode de configuration/programmation et l'on quitte donc le mode d'activité de programme. On incrémente ou décrémente la période de l'année affichée d'une journée en appuyant respectivement sur les touches *Time forward* ou *Time backward*. De manière générale, la touche *Set day* permet de modifier le jour courant du thermostat. Au bout de quatre-vingt-dix secondes sans touche appuyée, on revient dans le mode d'activité de programme et plus spécialement dans l'état *Hold* (*cf.* bouton *Hold temp*) ;
- *View program* : ce type d'événement permet d'entrer dans le mode de configuration/programmation et donc de quitter le mode d'activité de programme. Le système d'affichage de l'heure et du jour courants sert alors à afficher et à modifier une heure associée à une période de programmation donnée (par défaut, l'heure associée à la période de programmation n° 1). Le système d'affichage de la température cible courante sert lui à afficher et à modifier une température cible associée à une période de programmation donnée (par défaut, la température cible associée à la période de programmation n° 1). La période de programmation est aussi affichée. On passe d'une période de programmation à une autre, d'une heure associée à une autre, d'une température cible associée à une autre, en appuyant de nouveau sur cette touche *View program*. Au bout de quatre-vingt-dix secondes sans touche appuyée, on revient dans le mode d'activité de programme et plus spécialement dans l'état *Hold* (*cf.* bouton *Hold temp*) ;
- *Run program* : ce type d'événement permet d'entrer dans le mode d'activité de programme et donc de quitter le mode de configuration/programmation. On entre dans l'état *Run* où il y a prise en compte du programme, par opposition à l'état *Hold* où ce n'est pas le cas ;
- *Hold temp* : cette touche permet d'atteindre l'état *Hold*. Au lieu de comparer seconde par seconde l'heure courante à l'instant auquel la température cible doit changer, le thermostat maintient la température cible courante jusqu'à ce que la touche *Run program* soit appuyée. L'état *Hold* est le sous-état par défaut du mode général d'activité de programme qui est, rappelons-le, *Operate* ;
- *F-C* : cette touche permet d'alterner l'affichage de la température ambiante, la température cible courante, et éventuellement une température associée à une période de programmation donnée, entre les degrés Fahrenheit (affichage par défaut) et les degrés Celsius.

5.3 ANALYSE

Par essence, l'analyse qui est menée ici est strictement indépendante de toute considération d'implémentation. En particulier, aucune des propriétés du langage de programmation objet utilisé ne doit gêner la spécification. On montre en outre comment une même construction peut se transformer, parfois très différemment, dans deux environnements de développement distincts. En effet, les bibliothèques logicielles, *i.e.* les composants disponibles à réutiliser, vont aussi fortement contraindre la manière de coder l'application : ce qu'ils offrent en termes de services, comment ils doivent être réutilisés (par et seulement par héritage, par exemple) ainsi que leur contexte de mise en œuvre (prérequis et post-requis). En développant en tout premier lieu la modélisation des aspects statiques de ce cas de domotique, nous voyons que les constructions de modélisation UML dont nous avons besoin sont, en quantité et en complexité, assez limitées. Néanmoins, le *Class Diagram* de la figure 5.2 joue bien entendu un rôle crucial dans la compréhension et l'implantation du système.

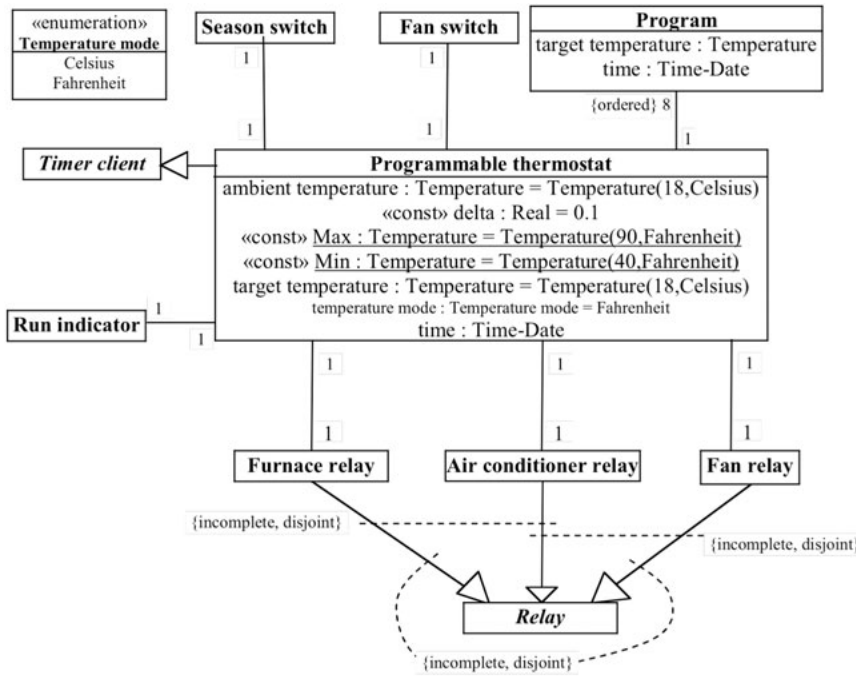


Figure 5.2 – *Class Diagram* du cas de domotique.

5.3.1 Explication du modèle de la figure 5.2

De très nombreuses itérations ont été nécessaires pour aboutir au modèle de la figure 5.2. Bien évidemment, partant du cahier des charges pour aboutir à ce modèle, de

nombreuses erreurs et donc corrections ont été faites dans des versions antérieures à la solution présentée. Nous les discutons par la suite et donnons donc le schéma final qui est celui utilisé pour la conception et l'implémentation.

Les règles générales de modélisation sont de ne pas remplir la partie basse des boîtes où figurent les opérations. Ces dernières apparaissent seulement dans les modèles dynamiques sous forme d'actions ou d'activités. Il existe deux raisons à cela, la première étant la recherche d'une meilleure lisibilité/compréhensibilité des schémas. La seconde est relative au processus de modélisation lui-même. De manière générale, il est beaucoup plus intuitif et naturel de découvrir les opérations au moment de la construction des modèles dynamiques, d'où l'absence de ces dernières dans le modèle de la figure 5.2. Comme nous l'avons déjà envisagé, en analyse mais seulement en analyse, faire figurer les opérations dans les boîtes est rarement nécessaire sous l'hypothèse que les modèles dynamiques existent et donc complètent bien les modèles statiques. À l'exception de types bien ciblés comme le type *Temperature* ci-dessous, les *Class Diagram* sont allégés des opérations dans les classes.

Le type *Programmable thermostat* est central dans le modèle. Chacun de ses attributs a un rôle important caractérisé en particulier en fonction de son usage (lecture/écriture/mise à jour). Les attributs constants (stéréotype *«const»*) sont *delta*, *Max* et *Min*. Le fait que *delta* ne soit pas un attribut de classe suppose que les constructeurs de *Programmable thermostat* soient paramétrés de manière à fixer *delta* à l'instanciation. La propriété *delta* sert comme unité de discrétisation des températures exprimées dans une unité de mesure donnée. Nous avons choisi l'unité °C pour comparer deux températures entre elles (*cf.* figure 5.10, état *Control*), l'unité °F pouvant aussi faire l'affaire. En effet, la valeur par défaut de *delta* égale à 0,1 soustraite ou additionnée à des températures n'a de sens que dans une unité de mesure particulière. Quant à *Max* et *Min*, ils indiquent simplement les seuils maximal et minimal de réglage de la température cible, et sont fixés respectivement, à 90 °F et 40 °F comme le veut le cahier des charges.

On peut noter dans la figure 5.2 deux détails de modélisation intéressants. Le premier est le type énuméré *Temperature mode*. Deux valeurs possibles composent ce type, puisque ce sont les deux unités manipulées dans l'application, cela bien que le type *Temperature* évoqué ci-après gère en plus les degrés Kelvin. Le second détail est la triple utilisation de la contrainte d'héritage *{incomplete, disjoint}*. Bien que déclarée « par défaut » dans le chapitre 2, en conformité avec UML 2.x, elle est nécessaire car elle concerne des sous-types deux à deux : trois couples formés parmi trois sous-types. Si cette contrainte avait été omise, cela signifierait que l'intersection des trois sous-types de *Relay* est vide, alors que nous voyons en figure 5.2 que les trois intersections deux à deux sont vides, ce qui est différent (*i.e.* plus contraignant en l'occurrence).

Type d'objet Temperature

Cette étude de cas intègre de manière patente la notion de température. Nous avons choisi de traiter cette notion comme un *value type* [2], c'est-à-dire un type primitif uniquement utilisé dans la signature des attributs et des opérations. En UML 2.x, un

value type est une sorte de *DataType* utilisateur. Nous n'utilisons cependant pas le stéréotype *«datatype»* dans la figure 5.3 comme nous l'avions fait dans le chapitre 2 pour le type *Time-Date* par exemple. Nous considérons justement que *Temperature* a une connotation trop « métier ».

Que nécessite fondamentalement l'application ? Des calculs sur des températures, par exemple convertir des degrés Celsius en degrés Fahrenheit par la formule bien connue : $9/5 \times t_C + 32 = t_F$. Ce calcul figurera quelque part en tant que service assuré par un type d'objet. Quel est ce type sinon le type *Temperature* lui-même ? Cette évidence est pourtant mise en question si l'on regarde avec attention la spécification originelle de Rumbaugh *et al.* dans [7, p. 108]. Ce calcul est affecté à l'interface utilisateur, un composant graphique en quelque sorte ! Excellent cas d'un point de vue pédagogique, il illustre à la perfection ce qu'il ne faut surtout pas faire. Ainsi dans [7], *convert temp to F* et *convert temp to* C lancent le calcul de conversion. De quoi ? De la température cible courante et/ou des températures enregistrées dans la table de programmation ? Ce n'est pas clair du tout. Dans l'interface utilisateur présentée à la figure 5.1, on s'attend à ce que toutes les données de température affichées à un instant donné s'expriment dans la même unité, dès lors que l'utilisateur a choisi les degrés Celsius par exemple. Sinon, imaginer l'interface utilisateur affichant en degrés Celsius la température ambiante et en degrés Fahrenheit la température cible ne serait vraiment pas satisfaisant.

En fait, attribuer le calcul de conversion à *Temperature* va faciliter la réponse à ces attentes mais aussi montrer que le calcul de conversion n'a à être lancé que sur rafraîchissement de l'affichage (activation du bouton *F-*C) et non pas *a priori*, comme le laisse entendre une opération telle que *convert temp to F*. Cette collaboration de *Température* avec son environnement est aussi importante que les responsabilités qui lui sont assignées. Wirfs-Brock *et al.* dans [9] insistent bien sur ce principe de responsabilité/collaboration où un type incarne une unité logique : il doit être vu comme quelque chose d'indivisible et de complet. Ainsi, les fonctions qu'il regroupe forment un tout en soi : tout ce qui concerne les calculs de conversion, de montée et de descente de température ainsi que de comparaison des températures font l'unité même du type *Temperature*.

En résumé, le type *Temperature* est la représentation d'une notion dont les propriétés peuvent être pensées et par là même spécifiées en UML, en occultant temporairement le cahier des charges de l'application. Il est ensuite incorporé dans l'application comme un type de base. L'idée est de le voir dans le modèle de l'application non pas dans une boîte comme en figure 5.3, mais comme type des attributs (entre autres, voir pour cela la figure 5.2) comme s'il n'était pas modifiable par l'analyste.Dans le même esprit, les calculs temporels nécessaires à l'application sont du ressort du type *Time-Date*. Nous considérons en effet que ces calculs sont universels et disponibles dans n'importe quel environnement de développement. En revanche, nous ne pouvons pas compter sur la même accessibilité pour le type *Temperature* d'où sa description en figure 5.3. Dans le modèle de la figure 5.3, les conversions sont implantées via les opérations *asCelsius*, *asFahrenheit* et *asKelvin*. Des opérateurs de comparaison sont fournis ainsi que les fonctions de montée (*increment*) et de des-

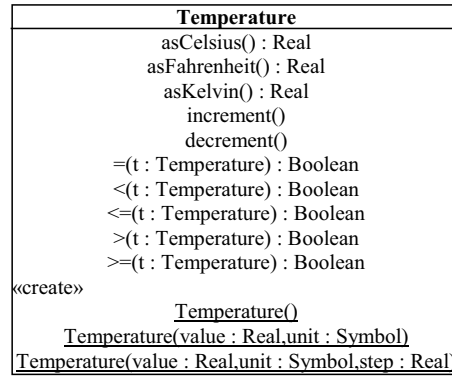


Figure 5.3 – Spécification du type d'objet *Temperature*.

cente (*decrement*) qui simulent l'évolution d'une température. Le pas d'évolution est renseigné si besoin à la création d'une instance de *Temperature* (cf. stéréotype *«create»* en figure 5.3). Pour en terminer, notons que le service *asKelvin*, bien qu'inutile dans l'application, illustre le besoin d'anticipation, c'est-à-dire accroître « avant l'heure » le potentiel de réutilisation de *Temperature*. Dans le futur, on peut très bien imaginer la nécessité de températures exprimées en degrés Kelvin, dans une application de physique/chimie par exemple. Un excellent exemple de démarche d'anticipation est présenté dans [1, p. 293-326] où Booch observe que : « *The careful reader may wonder why we have proposed a class for this abstraction, when our requirements imply that there is exactly one temperature sensor in the system. That is indeed true, but in anticipation of reusing this abstraction, we choose to capture it as a class, thereby decoupling it from the particulars of this one system.* » En d'autres termes, penser les objets au-delà des besoins ponctuels d'une application est un moyen simple mais sûr de recherche de la réutilisabilité.

Type d'objet Program

La structuration de la table de programmation où figurent les créneaux horaires auxquels les températures cibles se modifient est définie en figure 5.2 par le type *Program*. En fait, une instance de *Program* est tout simplement une paire *target temperature/time*. Le thermostat a donc huit paires (représentées par la cardinalité 8), les quatre premières (index 1-4) correspondant aux journées normales de la semaine alors que les quatre autres (index 5-8) correspondent au week-end. La contrainte UML *{ordered}* fournit entre autres un accès direct par indice à chaque couple de valeurs.

Le choix fait en figure 5.2 est que la différence entre *weekday* et *weekend* n'est pas représentée sinon implicitement via la convention que l'intervalle 1-4 concerne des créneaux horaires qui s'appliquent à *weekday* alors que 5-8 est destiné à *weekend*. Cette solution est discutable. Voici de façon non exhaustive deux autres variantes possibles (figure 5.4) qui sont en compétition entre elles et avec le choix fait en figure 5.2.

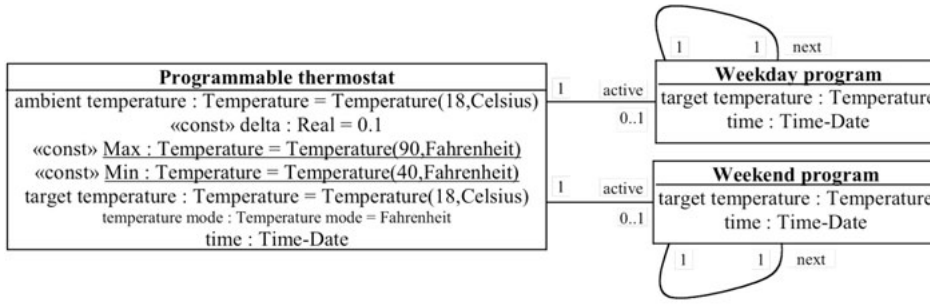


Figure 5.4 – *Class Diagram,* variations sur le type d'objet *Program*.

Les deux solutions en figure 5.4 distinguent deux tables de quatre paires *target temperature/time* chacune. On remarque d'ailleurs une factorisation possible des types *Weekday program* et *Weekend program* compte tenu de la similarité de leur structure, voire une fusion pure et simple qui ferait revenir au modèle de la figure 5.2. Les associations sont plus intéressantes. La version du haut de la figure 5.4 considère à tout moment la disponibilité ou non (cardinalité *0..1*) d'un couple de valeurs actif pour les jours de la semaine ou pour le week-end. En effet, si l'on est le week-end, il n'y a pas de couple actif dans la table concernant les jours normaux de la semaine et vice-versa. Cette paire active, si elle existe, n'est donc jamais qu'une des quatre paires de la table (contraintes *{subsets weekday program}* et contraintes *{subsets weekend program}*).

En outre, le rôle *active* se substitue à la contrainte *{ordered}* de la figure 5.2 pour connaître le créneau horaire en vigueur, *i.e.* celui qui indique la température cible à suivre. La paire *target temperature/time* référencée via le rôle *active* doit donc être interchangeable dès qu'un nouveau créneau horaire se présente. L'algorithme exécuté par le thermostat consiste donc fondamentalement à dire laquelle des deux tables utiliser et à élire, parmi quatre, le couple actif.

La version du bas de la figure 5.4 est plus originale. Elle considère une circularité, c'est-à-dire un créneau horaire actif, l'instance de *Weekday program* ou *Weekend program* référencée à un certain moment, plus une navigation (rôle *next*) pour explorer les autres périodes candidates à l'activation. Cette dernière version est plus générique, et donc plus évolutive, car le choix du nombre d'éléments dans la table peut être différé à l'implémentation. Du point de vue de la lisibilité et de la compréhensibilité, elle est peu satisfaisante : une personne ayant exprimé les besoins qui figurent dans le cahier des charges peut s'interroger sur l'oubli ou non du fait qu'il y a quatre (et pas autre chose) créneaux horaires pour les jours ouvrés ainsi que pour le week-end. La latitude d'interprétation que l'on laisse au programmeur avec la version du bas est à double tranchant. Finalement, la grande qualité des deux versions de la figure 5.4 est la largesse de choix des types de collection réutilisables en conception/implémentation alors que la solution en figure 5.2 suppose l'utilisation de collections ordonnées : *OrderedCollection* ou encore *SortedCollection* en Smalltalk par exemple, si l'application était codée avec ce langage de programmation.

Type d'objet Timer client

Ayant besoin de services de cadencement (pour scruter la table de programmation seconde par seconde pour éventuellement basculer de période), le type *Programmable thermostat* hérite d'une classe abstraite *Timer client* (à gauche de la figure 5.2). L'idée essentielle et bénéfique de la modélisation objet est d'extirper du modèle des besoins tout ce qui n'est pas spécifique du métier, ici la domotique. Un *Timer* est en informatique une notion universelle. Il est quasiment banal d'en avoir besoin. L'idée est donc de dire que le type *Programmable thermostat* se comporte comme un client de *Timer*. Ce comportement n'a rien à voir avec le métier. Il doit donc être spécifié indépendamment et réutilisable n'importe où, dans une autre application particulièrement comme dans le chapitre 6. Quant au comportement d'un *Timer*, la section « Patrons de spécification » de ce chapitre le décrit.

Le lien de la figure 5.2 qui fait que *Programmable thermostat* hérite de *Timer client* pose un problème épineux : faut-il ne pas gaspiller les liens d'héritage ? Nous sommes en phase d'analyse et rien ne dit que nous n'aurons pas en conception/implémentation la nécessité de faire hériter *Programmable thermostat* d'une autre classe venant d'une bibliothèque quelconque, cela pour tirer profit des services qu'elle offre. *Programmable thermostat* ne peut pas dans ce cas hériter de deux classes à la fois si le langage l'empêche. Ainsi, si nous utilisons un langage de programmation objet qui ne supporte pas l'héritage multiple, de qui alors hériter ? De cette classe venant d'une bibliothèque ou de *Timer client* ?

À cet égard, une alternative de modélisation est proposée en figure 5.5. Cela ne résout vraisemblablement rien car seul Java possède nativement une notion d'interface représentée par le stéréotype *«interface»* et la relation *realization* d'UML, cette dernière étant dessinée comme l'héritage mais avec un trait en pointillé au lieu d'un trait plein. Au-delà, la version de la figure 5.5 reste plus ouverte si l'on ne sait pas bien comment la connectivité avec un *Timer* peut s'opérer dans l'environnement d'exécution. La solution de la figure 5.2 elle, sous-entend la disponibilité d'une implémentation complète sur la base d'une classe préexistante. De manière plus générale, il y a donc peut-être lieu d'envisager d'autres possibilités que l'héritage ou la relation UML de réalisation pour bénéficier de services offerts par un *Timer*. Cela signifie qu'un fragment de modèle d'analyse faisant appel à *Timer* est transformable en différentes micro-architectures, qui se distinguent par la méthode choisie de réutilisation/implémentation des services de *Timer*.

Pour ne pas frustrer le lecteur sur cette discussion qui se perd un peu en conjectures et en anticipation de la spécification à venir du type d'objet *Timer*, nous pouvons signaler dès à présent que la bibliothèque *PauWare.Statecharts* supporte la classe abstraite *Timer_client* dont la réutilisation a deux modes possibles. Le premier est *public class Programmable_thermostat extends Timer_client* et nécessite la définition d'une méthode *time_out* pour faire perdre le caractère abstrait à *Programmable_thermostat*, celui-ci étant hérité de *Timer_client*. Le second mode fortement lié à Java est ce que nous appelons « la concrétisation à la volée » et permet l'économie d'un lien d'héritage. Il évite donc le dilemme évoqué ci-avant si la classe *Programmable_thermostat* doit hériter d'une autre classe. Le code qui suit est de notre point de vue pas très élé-

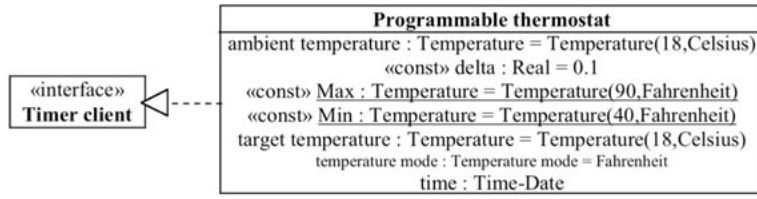


Figure 5.5 – Utilisation de la relation *realization* au lieu de la relation d'héritage pour disposer de services de *Timer*.

gant mais reste du Java « standard », si l'on veut permettre à *Programmable_thermostat* d'hériter d'autre chose que de *Timer_client*.

```
public class Programmable_thermostat {
   com.FranckBarbier.Java._PauWare.Timer_client timer_client =
     ➥ new com.FranckBarbier.Java._PauWare.Timer_client() {
   public void time_out(long delay,com.FranckBarbier.Java._PauWare.
        ➥ Statechart context) {
      // définition à la volée de la méthode time_out
      // traitement de l'événement ici
      }
   };
   …
   try {timer_client.to_be_set(5000L); // exemple d'armement}
   catch(Exception e) {…}
   …
}
```

Le point primordial est que l'implémentation du type UML *Timer client* reste ici masquée. Nous ne savons pas en effet comment le cadencement est réellement instrumenté à partir de couches logicielles « basses » (machine virtuelle Java, système d'exploitation, accès à des services d'un intergiciel…). Nous conservons donc l'abstraction nécessaire à l'analyse, le code précédent n'étant qu'à vocation didactique.

Type d'objet Relay

Il paraît utile de regrouper les trois relais de puissance que commande le thermostat au travers d'une classe abstraite *Relay*. L'effort est ici minime et se justifie par le fait que les trois possèdent un comportement assez semblable (voir ci-après). Nous faisons cela pour ébaucher une discussion sur l'appariement de *State Machine Diagram*. En effet, chacun des trois relais a son comportement spécifié par un *statechart* que nous utilisons pour chercher les ressemblances (factorisées dans le type *Relay* en l'occurrence) mais également les dissemblances (qui vont devenir des propriétés spécifiques à chaque sous-type de relais). Nous pouvons éventuellement faire de même pour les deux commutateurs *Season switch* et *Fan switch*. La généralisation d'une telle démarche ne se justifie néanmoins que si l'on a à faire de manière récurrente à ce genre d'objets. Le *State Machine Diagram* du type *Relay* présenté par la suite, montre donc ce qui est mis en commun.

Revenant aux trois relations d'héritage sur *Relay*, comme pour *Timer client*, des problèmes peuvent arriver en conception. Par exemple, une classe *Furnace_driver*

incarnant un pilote de dispositif physique pour le chauffage, peut être imposée par le fournisseur à qui l'on va acheter le système électronique qui commandera en réel ce chauffage. Faire hériter la classe d'analyse *Furnace relay* à la fois de *Relay* et de cette classe *Furnace_driver* est problématique en Java par exemple. De manière plus générale, l'intégration en conception de composants logiciels extérieurs a tendance à malmener (casser parfois) les architectures préconisées lors de l'analyse.

5.3.2 Invariants du modèle de la figure 5.2

Les invariants ci-dessous sont écrits en OCL. Nous avons délibérément choisi de les extraire du schéma proprement dit pour ne pas l'alourdir. Ils peuvent cependant apparaître graphiquement sous forme de notes UML. Commençons par des assertions simples comme le fait que la constante *delta* valorisée à l'instanciation du thermostat programmable, doit être strictement positive.

```
context Programmable thermostat inv: delta > 0
```

D'autres invariants moins triviaux que ce dernier sont possibles. Par exemple, pour les jours de la semaine, et symétriquement pour le week-end, les quatre valeurs de *time* doivent être différentes au risque de compliquer la méthode de détermination du changement de créneau horaire. En clair, si la table de programmation comporte deux instances de *Program* indiquant par exemple, 19 h-18 °C et 19 h-21 °C, il y a une contradiction sur la température cible à maintenir. Bien qu'il soit aisé d'exprimer un invariant empêchant cette possibilité d'erreur, la question plus profonde est : que signifie la violation de l'invariant et que faire à l'exécution ? Voici cet invariant :

```
context Programmable thermostat inv: programsubSequence(1,4)forAll(p1,p2 |
     ➥ p1  p2 implies p1.time  p2.time)
context Programmable thermostat inv: programsubSequence(5,8)forAll(p1,p2 |
     ➥ p1  p2 implies p1.time  p2.time)
```

Cet invariant n'est pas critique quant au fonctionnement du thermostat. En effet, s'il est violé, l'algorithme qui établit la température cible à suivre peut aléatoirement et/ou arbitrairement choisir une instance de *Program* au détriment d'une autre, par exemple, retenir 19 h-18 °C au lieu de 19 h-21 °C. L'algorithme n'a donc pas à lever une exception pour signaler un dysfonctionnement. Au contraire, le fait que la constante *delta* puisse être négative ou nulle, engendre un mode de commande du chauffage et du conditionneur d'air complètement aberrant d'où la plus grande importance du premier invariant, et donc sa présence ci-dessus. De manière générale, il n'est pas raisonnable de fixer de règles strictes. Selon la nature de chaque application, charge à l'analyste de savoir si des invariants, comme celui portant sur la table de programmation, doivent être écrits ou non. L'important est de définir une politique de défense en cas de violation des invariants « critiques », politique qui sera l'assurance fiabilité de l'application.

Nous pensons qu'un travail clé de l'analyse est de découvrir et d'exprimer cette criticité de manière à fournir un guide sûr aux concepteurs/programmeurs dans leur

gestion de la tolérance aux fautes. En déterminant les niveaux de criticité et en classant les invariants selon ces niveaux, il est possible par une méthode simple de mener une réelle politique de défense des programmes. Une remarque est cependant qu'UML, même dans sa version 2, apporte peu ou rien[1] sur le sujet.

5.3.3 Dynamique du système

Nous montrons ici que la modélisation des aspects dynamiques s'appuie seulement sur les *Statecharts* de Harel [4]. Les modèles présentés sont *nécessaires et suffisants* pour appréhender en totalité et de façon cohérente le système décrit dans le cahier des charges. Tout autre type de modèle UML mis en œuvre devient de fait redondant. Un modèle supplémentaire, un *Collaboration Diagram* par exemple, doit alors être géré comme une « vue » supplémentaire ou une façon différente de dire les choses, avec le risque de contradictions que cela suppose. Des vues de synthèse de toute la dynamique du système sont à ce titre intéressantes car le défaut des *State Machine Diagram* est de donner des vues parcellaires et locales de la dynamique globale. Un *Communication Diagram* qui est une sorte d'*Interaction Overview Diagram* dans la nouvelle liste des types de diagramme d'UML 2.x (voir chapitre 3), est proposé dans la suite de ce chapitre pour globaliser tous les flux de contrôle qui circulent entre les différents automates. Au préalable, l'idée de base est : soit un type d'objet du *Class Diagram* en figure 5.2, ce type voit ou non son comportement spécifié par un *statechart*. Seul le type *Program* n'a à cet égard pas de comportement décrit.

Comportement des commutateurs de saison et de ventilation

Les deux commutateurs en figure 5.6 ont des comportements triviaux à l'exception qu'il faille méthodiquement considérer le commutateur de saison comme le point d'entrée de tout le système. Dès qu'il bascule sur *Is off*, une émission d'événement a lieu à destination du thermostat programmable pour stopper de suite, selon le contexte, la commande du chauffage ou du conditionneur d'air. Pour le commutateur de ventilation, c'est cette fois le passage dans l'état *Is on* qui est remarquable car dès lors, le ventilateur va s'activer sans synchroniser par la suite son fonctionnement avec celui du chauffage ou du conditionneur d'air. Seule la restauration du mode automatique (*Is auto*) rétablit la situation. Le point le plus important est donc l'existence d'un flux de contrôle de ces deux types d'objet vers le thermostat, cela du fait de l'émission d'événement/message symbolisée par la notation ^[2].

Comportement des relais de puissance et du témoin d'activité

Les relais, tels qu'ils sont formalisés en figure 5.7, ont la particularité d'accroître le degré global de couplage du système. En effet, il existe une interaction entre *Furnace relay* et *Air conditioner relay* d'un côté, avec *Run indicator* (figure 5.9) de l'autre. Cha-

1. À l'exception bien sûr d'articles scientifiques et donc de contributions sur le sujet que l'on peut retrouver dans toute la série de conférences sur UML (voir section « Webographie » en fin de chapitre).

2. Rappelons que cette notation est native en UML 1.x alors qu'en UML 2.x, elle provient plus précisément d'OCL 2.

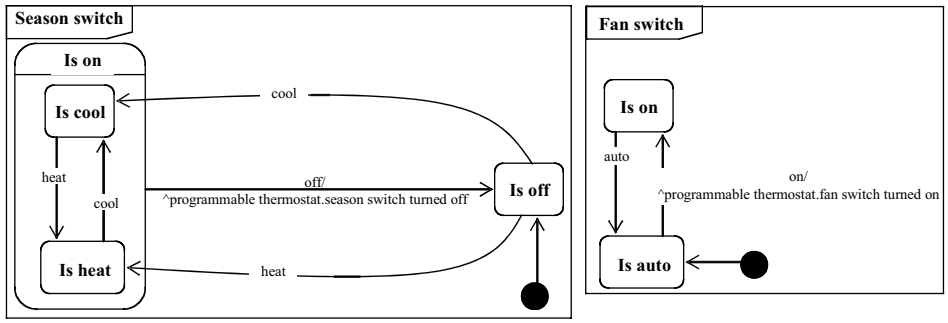


Figure 5.6 – Spécification du comportement des commutateurs de saison et de ventilation.

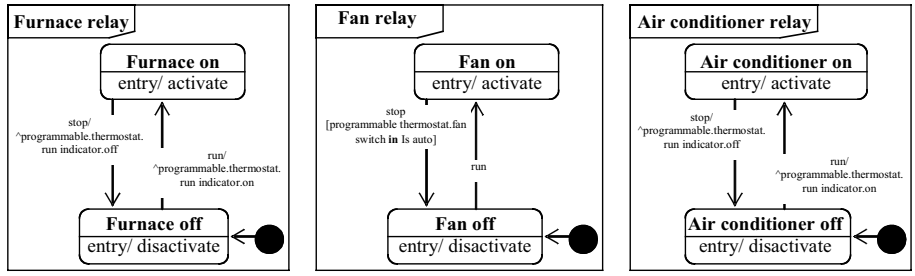


Figure 5.7 – Spécification du comportement des relais de puissance du chauffage, du ventilateur et du conditionneur d'air (1re proposition).

cun émet les événements *on* et *off* à destination de *Run indicator* pour que ce dernier s'allume puisque, d'après le cahier des charges, celui-ci s'illumine si quelque chose est actif (le chauffage ou le conditionneur d'air, la ventilation n'étant pas concernée).

Le choix de modélisation fait en figure 5.7 est critiquable au sens où la navigation qui autorise l'émission des événements *on* et *off* est *programmable thermostat.run indicator*. Une fois implantée, cette navigation renvoie *a priori* un objet de type *Run indicator* utilisé par les types *Furnace relay* et *Air conditioner relay* pour faire parvenir les consignes *on* et *off*. En clair, les types *Furnace relay* et *Air conditioner relay* connaissent le type *Run indicator*, d'où le couplage.

Une solution alternative est d'augmenter l'intelligence du thermostat (*i.e.* son niveau de responsabilité) en lui conférant le soin d'allumer ou d'éteindre le témoin d'activité lorsqu'il a préalablement mis en marche, respectivement arrêté, le chauffage ou le conditionneur d'air. L'avantage est que *Furnace relay* et *Air conditioner relay* deviennent complètement ignorants de leur environnement dans la mesure où ils n'émettent plus aucun événement (figure 5.8). L'inconvénient est la centralisation du travail dans le thermostat programmable, c'est-à-dire l'obtention d'un « gros » objet, complexe, qui est un peu l'antithèse de l'idée de simplicité, de partage harmo-

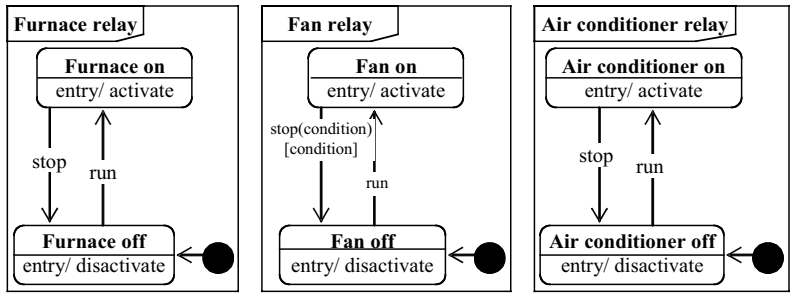


Figure 5.8 – Modèle de la figure 5.7 modifié pour diminuer le couplage.

nieux des responsabilités et de coopération qui prévaut dans les systèmes objet. Nous sommes donc ici sur un cas type d'arbitrage entre différents facteurs qualité. Le choix fait en figure 5.7 diminue le niveau de réutilisabilité de *Furnace relay* et d'*Air conditioner relay*. La spécification est cependant plus claire, voire plus naturelle, et donc intrinsèquement plus facile à modifier si le cahier des charges venait à évoluer.

Dans la figure 5.7, il est important d'observer que la connaissance de *Run indicator* par les deux types de relais se fait par l'intermédiaire de *Programmable thermostat*. En d'autres termes, la navigation directe *run indicator* n'est pas utilisée car elle supposerait de dessiner, en figure 5.2, une association dérivée entre *Run indicator* et *Furnace relay*, ainsi qu'entre *Run indicator* et *Air conditioner relay*. Au final, nous retenons la solution de la figure 5.8 au détriment de celle de la figure 5.7.

L'autre problème de couplage est la dépendance entre *Fan relay* d'un côté et *Fan switch* de l'autre. En effet, le relais de puissance du ventilateur recevant la consigne *stop* ne s'arrête que si le commutateur de ventilation est en mode automatique (*Is auto*, voir figure 5.7). Comment s'informer de l'état courant du commutateur de ventilation, mode automatique ou mode forcé ? La garde *programmable thermostat.fan switch* ***in*** *Is auto*[1] en figure 5.7 contraint *Fan relay* non seulement à connaître *Fan switch* mais à disposer d'une visibilité sur son état courant, qui est d'une manière ou d'une autre, une forme de transgression du principe d'encapsulation. Le niveau de couplage va néanmoins beaucoup dépendre de l'implémentation : doit-on implanter des services qui renseignent sur l'état des objets ou laisser des accès purs et simples à l'intérieur des objets ? Plus généralement, l'implémentation de l'opérateur *in* des *Statecharts* peut être fort coûteuse. Éviter ce problème, c'est retenir le modèle de la figure 5.8. Un couplage moindre est proposé car la garde devient tout simplement *condition*, c'est-à-dire une valeur booléenne véhiculée comme paramètre de l'événement *stop*. Cette manière de procéder ne nécessite donc plus pour *Fan relay*, la connaissance des états que possède à tout moment *Fan switch*.

Le message est toujours le même : le plus grand défaut d'UML est de supporter des constructions de modélisation qui sont mutuellement interchangeables ou dont la finalité d'utilisation est semblable. Pour un problème donné, comme le témoin

1. En OCL, cela s'écrit un peu différemment : *programmable thermostat.fan switch.oclInState(Is auto)*. Il est d'ailleurs regrettable que les deux formes d'écriture subsistent.

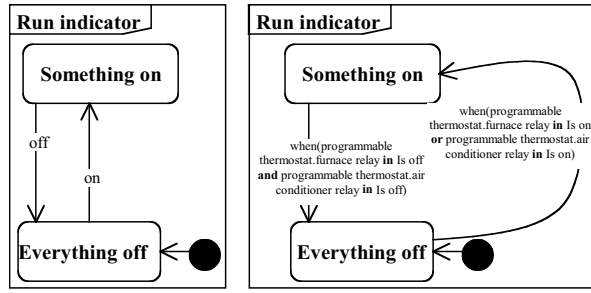


Figure 5.9 – Spécification du comportement du témoin d'activité et variante.

d'activité à spécifier, la difficulté est le choix du bon outil. Par exemple, on pourrait même imaginer que le témoin d'activité envoie des événements aux relais de puissance pour interroger leur état, cela au lieu d'utiliser *in* et *when*. Ces derniers renverraient alors un événement informant quel est leur état courant. Cette souplesse et cette variété sont extrêmement gênantes car à la complexité intrinsèque de l'application s'ajoute celle de la méthode de mise en œuvre d'UML.Une construction de modélisation délicate à utiliser est la notion de *ChangeEvent* d'UML 1.x qui a perduré dans UML 2.x. Bien que le métatype *ChangeEvent* reste dans le métamodèle, les notations associées comme le terme *when* de la figure 5.9 ont, au regard de la documentation [6], *a priori* disparu : cette notation n'est pas mise en œuvre dans la documentation. En UML 1.x, c'est un événement qui n'est pas explicitement généré mais dont une occurrence apparaît dès qu'une condition bascule de faux à vrai. Sur ce principe, le *statechart* du témoin d'activité en figure 5.9 (celui de gauche) peut être réécrit (figure 5.9 à droite). Il est évident que cette version de droite n'est pas source de simplicité et donc de compréhensibilité. De manière générale, les éléments de notation *in* et *when* doivent être utilisés avec parcimonie, particulièrement lorsqu'ils sont mêlés comme dans la partie droite de la figure 5.9.

Comportement du thermostat programmable

Pour compléter l'étude de la dynamique du système, il est nécessaire d'aborder le comportement du thermostat, véritable centre nerveux de toute l'application. Le *statechart* en figure 5.10 est à ce titre volumineux et complexe. Comme pour le *Class Diagram* en figure 5.2, de nombreuses itérations ont été nécessaires pour stabiliser le modèle. Le lecteur ne pouvant appréhender un tel schéma globalement, nous l'invitons à le regarder brièvement et à passer à la discussion qui suit. À mesure que nous expliquons partie par partie le schéma de la figure 5.10, nous montrons les difficultés de mise en œuvre des *State Machine Diagram* d'UML.

De plus, en fin de chapitre, l'implémentation basée sur la bibliothèque *PauWare.Statecharts* peut aider à la compréhension générale de la figure 5.10, cela si le lecteur est habituellement plus enclin à manipuler du code que des modèles. En bref, le conseil est d'occulter temporairement ce modèle pour y revenir après avoir lu le chapitre.

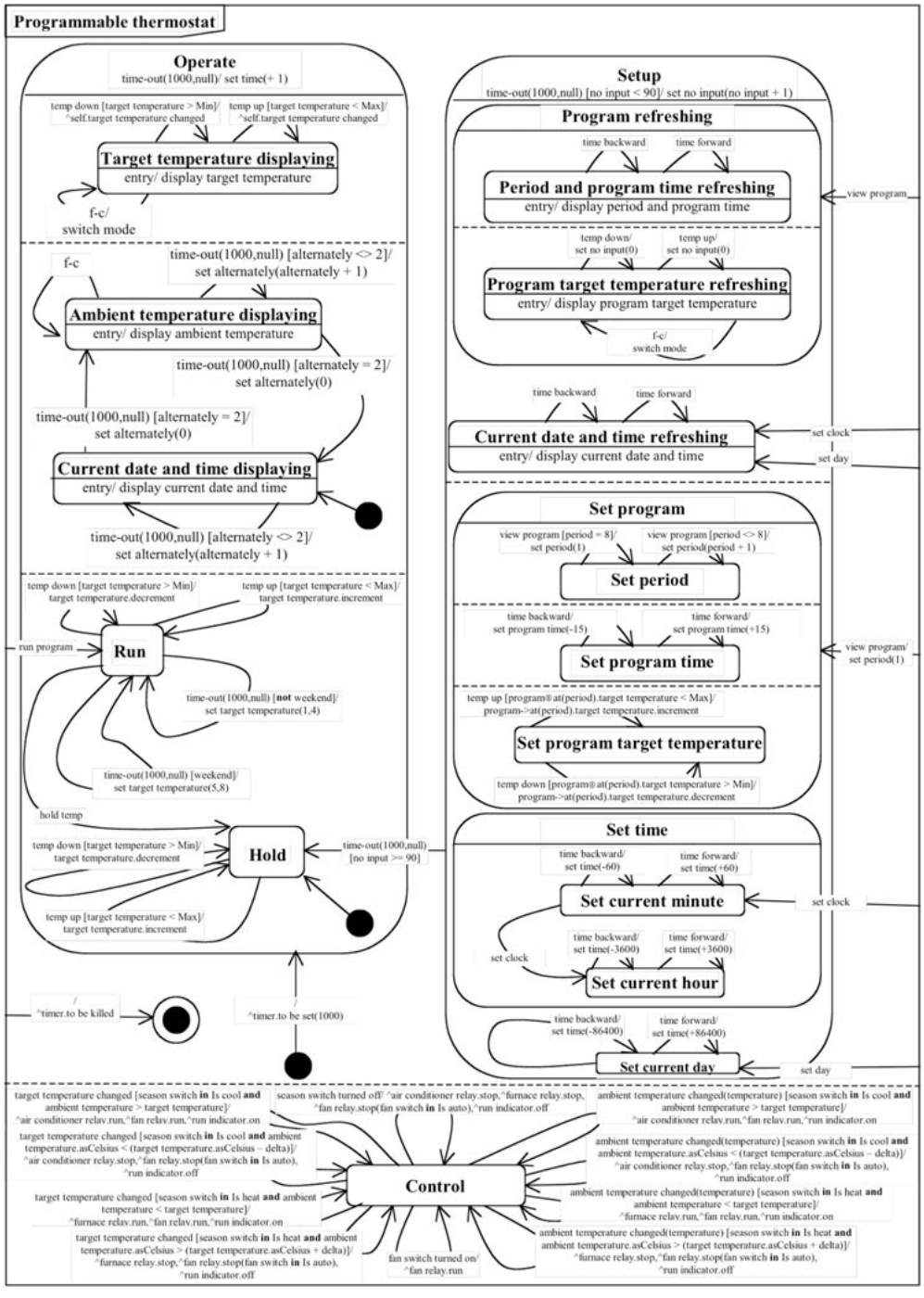


Figure 5.10 – Spécification du comportement du thermostat programmable.

Préconditions et post-conditions du modèle de la figure 5.10

Les préconditions et post-conditions décrites ici sont relatives aux événements, aux actions et aux activités. Nous les exprimons en OCL mais nous invitons le lecteur à consulter l'ouvrage de Cook et Daniels [2] pour comprendre la démarche suivie. En effet, attribuer des préconditions et post-conditions aux événements n'est pas préconisé dans la documentation officielle d'UML 2.x mais, en revanche, est tout à fait commun avec OCL 2. Les travaux de Cook et Daniels restent à ce titre une inspiration majeure d'UML. L'incorporation d'OCL dans UML en est évidemment la meilleure preuve. Au-delà, la sémantique des *Statecharts* de Harel ou de leur variante présentée dans [5], telle qu'elle existe à ce jour dans UML 2.x, demande des adaptations « à la demande », amenant justement à exprimer des préconditions et post-conditions assez différemment de ce qui est fait dans le chapitre 2.

Dans les post-conditions que nous allons décrire, beaucoup agissent sur trois variables locales au *statechart* en figure 5.10 :

- *alternately* dont le domaine de valeurs est *0..2*, permet de basculer toutes les deux secondes entre deux natures de données affichées, cela dans le mode d'activité de programme (état *Operate*). Cette variable est assignée en fonction des arrivées de l'événement *time-out* ;
- *no input* dont le domaine de valeurs est *0..90*, mémorise le nombre de secondes passées sans qu'aucune touche n'ait été pressée dans le mode de configuration/programmation (état *Setup*). Cette variable est aussi assignée en fonction des arrivées de l'événement *time-out* ;
- *period* finalement, dont le domaine de valeurs est *1..8*, représente le créneau horaire courant en mode de configuration/programmation. Cette variable évolue en fonction des appuis sur la touche *view program*.

Insistons sur le fait que placer ces propriétés en tant qu'attributs du type *Programmable thermostat* en figure 5.2 est *inapproprié* dans la phase d'analyse. Certes, l'implémentation Java que nous proposons fait que *Programmable thermostat* inclut au final ces trois variables comme attributs, mais cela résulte du besoin de données locales au *statechart* et non de propriétés intrinsèques de *Programmable thermostat*. En d'autres termes, ces trois variables sont des moyens d'arriver à nos fins : spécifier le plus finement et proprement possible le comportement du thermostat. Mais ce ne sont pas des données initiales du cahier des charges comme *time* peut l'être par exemple, puisque le thermostat doit disposer d'une horloge interne. En revanche, elles peuvent sans problème et de façon rationnelle, faire partie d'un modèle de conception.

Venons-en maintenant aux post-conditions :

```
context Programmable thermostat::ambient temperature changed(temperature :
     ➥ Temperature)
   post: ambient temperature = temperature -- rafraîchissement du champ ambient
-- temperature suite à une occurrence de l'événement ambient temperature changed
context Programmable thermostat::run program()
   post: alternately = 0
context Programmable thermostat::set alternately(i : Integer)
```

```
   post: alternately = i
context Programmable thermostat::set clock()
   post: no input = 0
context Programmable thermostat::set day()
   post: no input = 0
context Programmable thermostat::set no input(i : Integer)
   post: no input = i
context Programmable thermostat::set period(i : Integer)
   post: period = i
context Programmable thermostat::set program time(step : Integer)
   post: programat(period).time@pre + step = programat(period).time
context Programmable thermostat::set target temperature(min : Integer,max :
Integer)
   post: target temperature = programsubSequence(min,max)collect(p : Program |
➥ p.time = time)iterate(p : Program,t : Temperature = target temperature | p.target
➥ temperature) -- rafraîchissement du champ target temperature dans le cas et
-- uniquement dans le cas où l'heure courante (time) est égale à une heure
-- enregistrée dans la table de programmation
```

On remarque que la post-condition qui précède, compte tenu du fonctionnement standard de l'opération *iterate* d'OCL, prend la dernière valeur (dans l'ordre défini par la contrainte *{ordered}* en figure 5.2) de la température cible enregistrée dans la table. Les autres post-conditions sont :

```
context Programmable thermostat::set time(step : Integer)
   post: time@pre + step = time
context Programmable thermostat::switch mode()
   post: temperature mode@pre = Celsius implies temperature mode = Fahrenheit
and temperature mode@pre = Fahrenheit implies temperature mode = Celsius
-- la seconde partie de la post-condition après le and peut être déduite de la
-- première compte tenu que le domaine de valeurs de l'attribut temperature mode
-- est limité à deux valeurs
context Programmable thermostat::time backward()
  post: no input = 0
context Programmable thermostat::time forward()
   post: no input = 0
context Programmable thermostat::view program()
   post: no input = 0
context Programmable thermostat::weekend() : Boolean
   post: (result = "time.get(Calendar.DAY_OF_WEEK) == Calendar.SATURDAY ||
➥ _time.get(Calendar.DAY_OF_WEEK) == Calendar.SUNDAY;
-- avec _time de type GregorianCalendar en Java)
or (result = "time.tm_wday == 6 || _time.tm_wday == 0;
-- avec _time de type struct tm en C/C++)
```

Le choix de placer le service *weekend() : Boolean* dans le type *Programmable thermostat* n'est pas toujours bon. Cela aurait pu laisser la place à l'extension ou la réécriture d'un composant de gestion du temps comme *GregorianCalendar* en Java, *struct tm* en C/C++, *CTime* dans la bibliothèque C++ de Microsoft ou encore *Date* et *Time* en Smalltalk. Cependant, il peut être laborieux de spécialiser ce genre de composant élémentaire simplement pour le besoin d'un service aussi trivial que *weekend() : Boolean*.

Dans le chapitre 2, nous avons par exemple ajouté une fonction booléenne de calcul d'année bissextile (*leap year()*) sur le type *Time-Date*. Logiquement ainsi,

weekend() : Boolean aurait dû s'ajouter à *Time-Date*. Ne voulant délibérément pas étayer et spécifier en détail le type *Time-Date* puisqu'il est disponible partout, nous précisons juste dans la post-condition ci-dessus le critère d'obtention du résultat à l'implémentation. Par ailleurs, il est important de remarquer que le service *weekend() : Boolean* est sans effet de bord (c'est une obligation en OCL, vérifiable au niveau méta via l'opération *isQuery*). En effet, l'opération *weekend() : Boolean* est seulement une interrogation. Plus généralement, il est absolument fondamental de ne pas utiliser des opérations à effet de bord dans les gardes des automates surtout, au risque d'exprimer les choses de manière impérative au détriment d'une expression déclarative, seule forme véritablement acceptable en modélisation.

Aspects détaillés de modélisation en liaison avec le modèle de la figure 5.10

Nous passons ici en revue quelques points clés et pièges de modélisation d'un *State Machine Diagram* UML. Une gêne importante causée par ce type de diagramme est leur faiblesse à identifier et à tracer la manière et les moyens par lesquels les événements émis dans un *statechart* sont réceptionnés dans un autre. Cette absence de vision globale, que nous avons déjà soulignée, peut être en partie comblée par un *Sequence Diagram* ; mais cela reste insuffisant puisque ces derniers pêchent par leur inaptitude à traiter, entre autres, les exceptions aux cas normaux d'enchaînement des messages (explosion combinatoire). En revanche, les *Sequence Diagram* ont de bonnes qualités de synthèse. En ce sens, beaucoup d'informaticiens apprécient le genre de vue en figure 5.11 qui est, dans l'esprit, une synthèse de type *Sequence Diagram*. Ce type de document était central et intitulé *Object Communication Model* dans la méthode Shlaer et Mellor [8]. Dans OMT, il était appelé *Event Flow*. Nous l'avons rencontré dans le chapitre 3 sous le nom de *Communication Diagram* sous-type d'*Interaction Overview Diagram*. L'idée des *Communication Diagram*, au contraire des scénarios « traditionnels », est de faire une représentation *indépendamment* d'une quelconque échelle temporelle.

Sans être acyclique, le graphe de flux de contrôle de la figure 5.11 montre un couplage assez minime dès l'analyse. Ce schéma résume ce qu'il y a dans tous les *State Machine Diagram* décrits jusqu'à présent, sans rien apporter de nouveau en matière d'exigences supplémentaires à prendre en compte.

On observe que *Temperature sensor*, *Timer* et l'utilisateur via l'interface utilisateur (*User interface*, figure 5.11) sont vus comme des agents extérieurs au système (*i.e.* ils ne sont en particulier pas décrits dans la figure 5.2). Nous utilisons pour cela la notion d'*active class* d'UML 2.x en doublant le trait à gauche et à droite de la boîte matérialisant la classe. Conceptuellement et dans le pur respect de la définition donnée dans la documentation d'UML 2.x [7], cela signifie que les instances des classes actives ont « leur propre fil de contrôle » (*thread*). Ainsi, nous voyons ces trois types d'objet que sont *Temperature sensor*, *Timer* et *User interface* comme des entités dont les pannes de comportement n'ont pas d'incidence sur celui du thermostat programmable. En d'autres termes, et au pire, le non-fonctionnement des trois instances respectives de ces trois types arrête de façon probable le thermostat programmable (arrivée dans un contexte où tout message reçu et interprété par le thermostat pro-

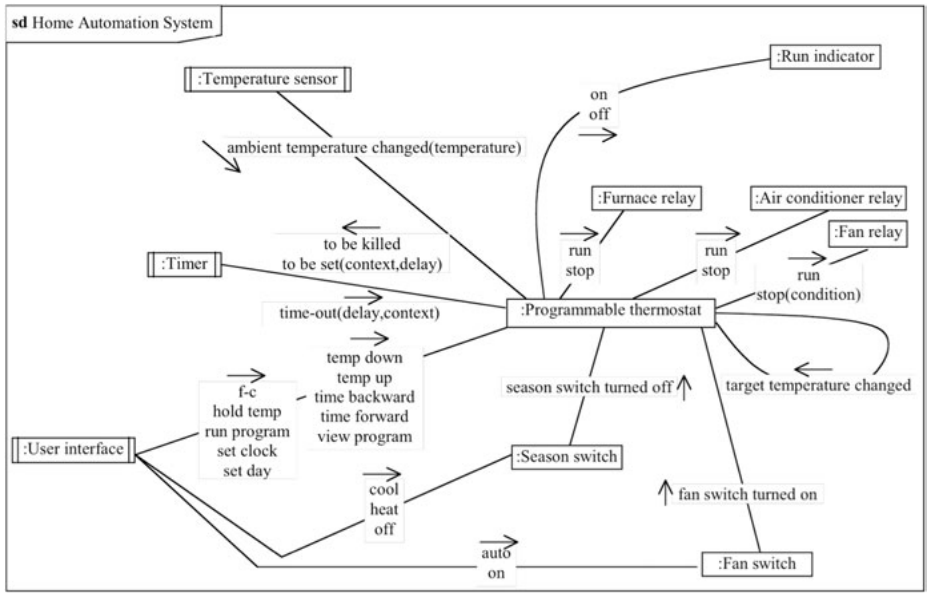


Figure 5.11 – Flux de contrôle global du système de domotique (*Communication Diagram*[a]).

a. L'intitulé *sd* pour *Sequence Diagram* en haut à gauche de la figure peut paraître incongru mais nous ne faisons que nous conformer à la documentation d'UML 2.x (qui contient un exemple du même type) bien que le chapitre 3 montre que les *Communication Diagram* sont parfois vus comme des *Sequence Diagram* et parfois comme des *Collaboration Diagram*.

grammable ne donne rien). Cela se produit néanmoins sans défaillance de l'application proprement dite.

État par défaut

La nécessité de définir des états par défaut est liée à un problème de déterminisme (voir [4] qui insiste sur ce point). Prenons l'exemple de l'événement *run program* qui force l'entrée dans le mode opératoire, et plus précisément dans l'état *Run* (figure 5.10 ou figure 5.12). Il y a deux façons d'écrire que l'on entre parallèlement dans l'état *Current date and time displaying*. La première, implicite, apparaît en figure 5.10 où un point noir suivi d'une flèche indique que *Current date and time displaying* est état par défaut : à la création d'une instance de thermostat programmable, cette dernière est dans l'état ou dans les états par défaut). L'état *Current date and time displaying* est donc « élu » au détriment d'*Ambient temperature displaying* chaque fois que l'on entre dans le mode opératoire, quelle qu'en soit la manière. En figure 5.12, le point noir ne se justifie plus (conservé, il est inutile mais n'introduit pas d'erreur dans la spécification) car est indiquée explicitement cette fois, la réactivité à *run program*. Attention quand même à cet exemple car s'il existe d'autres moyens que *run program* pour rentrer dans le mode opératoire, le point noir est nécessaire (cf. *time-out(1000,null) [no input >= 90]* en figure 5.10). Il faut donc veiller aux situations ambiguës, notamment celles d'états orthogonaux qui par essence s'activent en parallèle. Par ailleurs, les états par défaut concourent aussi à donner l'état des objets et

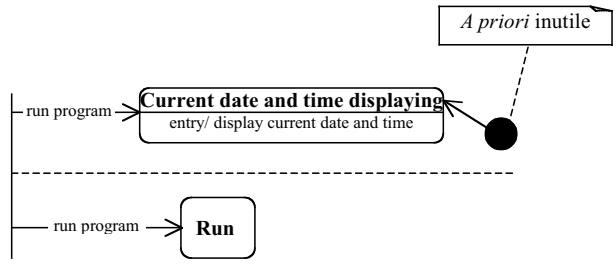


Figure 5.12 – Exemple d'alternative à l'état par défaut dans le mode opératoire.

donc du système à l'initialisation. Ils sont donc en même temps très pratiques et très intéressants pour bien documenter le démarrage d'une application.

Transition d'*Operate* à *Setup*

Les événements *set clock*, *set day* et *view program* (voir tout à droite de l'automate en figure 5.10) sont les trois moyens de se rendre dans l'état *Setup*. On remarque dès lors que l'horloge interne du thermostat programmable (symbolisée par le champ *time* du type *Programmable thermostat* en *figure* 5.2) n'est plus actualisée toutes les secondes grâce à la réactivité à l'événement *time-out* qui lance automatiquement l'action *set time* (concept d'*internal action* en UML associée ici à l'état *Operate* : voir tout en haut, à gauche de l'automate en figure 5.10). Selon la figure 5.10 donc, l'état *Setup*, au contraire d'*Operate*, occulte cette réactivité. Au regard des exigences, cette même réactivité devrait néanmoins perdurer. Comment la corriger[1] ?

Transition de *Setup* à *Operate*

L'événement *time-out* survenant dans l'état *Setup* est gardé par *[no input >= 90]* et *[no input < 90]*. La première de ces deux gardes spécifie la sortie de l'état *Setup* lorsque quatre-vingt-dix secondes se sont écoulées depuis que l'utilisateur n'a pas pressé sur *temp up*, *temp down*, *time forward*, *time backward*, *set clock*, *set day* ou *view program*. Si l'on sort de *Setup*, on se rend alors dans l'état *Hold* mais l'événement *time-out* avec la garde *[no input < 90]* fait rester dans *Setup* et déclenche l'action *set no input*. (voir tout en haut, à droite de l'automate en figure 5.10). Dans ce cas, on incrémente d'une unité la variable *no input* qui est locale au *statechart*. Cette dernière repasse à zéro dès que les boutons *temp up*, *temp down*, *time forward*, *time backward*, *set clock*, *set day* ou *view program* sont activés.

On remarque en figure 5.10 que *temp up* et *temp down* activent l'action *set no input(0)* dans l'état *Setup* alors que ce n'est pas le cas pour *time forward*, *time backward*, *set clock*, *set day* et *view program*. Ces cinq derniers types d'événement ont au contraire pour post-condition : *no input* = 0. Pourquoi ? Parce qu'écrire que la post-condition de *temp up* et *temp down* est *no input* = 0 est invalide dans l'état *Operate*. En effet, dans l'état *Operate*, on ne se préoccupe pas des quatre-vingt-dix secondes et

1. On laisse la correction ici triviale à titre d'exercice. De façon plus intéressante, on invite aussi le lecteur à modifier le source Java pour répercuter le changement fait dans le *statechart*, et surtout mesurer le temps de maintenance de ce code qui doit être court.

donc *no input* n'a pas à être manipulée. Cet exemple est une excellente illustration du partage de ce qu'il y a à faire entre actions associées aux événements et post-conditions associées à ces mêmes événements. On voit au moment de la conception que ceci permet une bonne segmentation des opérations d'une classe. En effet, il est toujours difficile de répartir logiquement les calculs à effectuer en services d'une classe donnée. La dissociation action/post-condition encouragée ici pour les *State Machine Diagram* d'UML, est un moyen de bien réussir. Finalement, l'autre grand moyen de transiter de *Setup* à *Operate* est de presser sur le bouton *run program* qui force l'entrée dans le mode d'activité de programme (exploration seconde par seconde de la table de programmation).

État *Control*

On note que l'état *Control* (figure 5.10, tout en bas) est parallèle à *Operate* et à *Setup* pour empêcher la coupure de la régulation continue de la température. Par ailleurs, *Control* ne travaille pas en direct sur la table de programmation, évitant dans cette spécification d'avoir des ressources partagées manipulées par des processus concurrents, ce qui peut entraîner des besoins d'exclusion mutuelle. Cette subtilité a des conséquences lors de l'implémentation car ainsi deux états orthogonaux n'auront pas forcément à devenir des tâches, *i.e.* de créer des instances de la classe *Thread* en Java par exemple, pour travailler.

Frontières du système

Booch dans [1, p. 294] évoque la nécessité de délimiter le système : « *Defining the Boundaries of the Problem* ». C'est une tâche difficile en général, et spécialement en UML, car certaines des entités d'une situation sont vues et incarnées par des types d'objet dans un modèle (figure 5.2 dans notre étude de cas) alors que d'autres sont rejetées ou alors représentées comme des acteurs au sens où ce concept est défini dans les *Use Case* (voir fin de chapitre 3). C'est ce dernier point qui nous intéresse ici car les limites du système que l'on construit vont être parfaitement et totalement caractérisées par le flux de contrôle entre types d'objet d'un modèle et « acteurs ». Dans l'étude de cas qui nous intéresse, ces derniers sont les entités *Temperature sensor*, *Timer* et *User interface* de la figure 5.11.

Par expérience, borner un système n'est pas une tâche inconnue : dans un système client/serveur simple par exemple, l'application cliente est bornée d'un côté par l'interface utilisateur (flux des événements provenant de l'interface utilisateur graphique) et de l'autre côté par la communication avec un SGBDR par exemple (flux des événements de communication selon une connectivité bien déterminée). Imaginons maintenant la spécification de la partie cliente en UML : toute la difficulté est de bien cadrer ce qui relève de la partie cliente (responsabilités) tout en montrant bien comment celle-ci interagit avec son environnement pour mener à bien ses calculs (collaborations).

Revenons à notre étude, le comportement du thermostat en figure 5.10 donne une bonne idée des communications de tout le système car le thermostat agit comme un centre de contrôle. Une grande partie des événements qu'il reçoit sont extérieurs

au système, c'est-à-dire qu'ils ne sont pas envoyés par des types d'objet en figure 5.2. *ambient temperature changed* est de ceux-là et, on peut naturellement le concevoir, cet événement est émis par un capteur de température. Pourquoi un type d'objet *Temperature sensor* n'est-il pas présent dans le *Class Diagram* en figure 5.2 alors que la situation de fonctionnement du thermostat l'impose de façon manifeste ? En poursuivant cette réflexion, pourquoi *Temperature sensor* est considéré comme un acteur en figure 5.11 ? Notons que dans une étude de cas du même genre concernant un système météorologique, Booch dans [1] manipule un capteur de température, et pour le besoin, le représente par une classe dans son modèle. En clair, il fait le contraire de ce que nous faisons.

Booch fait une remarque extrêmement pertinente dans [1] à laquelle nous adhérons totalement. Il écrit que le cahier des charges peut être directif sur le matériel et certaines parties, modules de logiciels à utiliser, ne serait-ce que pour des raisons de coût (reprise d'un existant par exemple). Dans un tel contexte, les propriétés d'un capteur de température peuvent être connues *a priori* (précision, accès à l'information produite…) et donc mentionnées dans un cahier des charges. Il n'y a ainsi pas forcément lieu de réaliser une spécification générique, comme la nôtre finalement, certains degrés de liberté étant figés dès le départ. Cette constatation faite, Booch dit que c'est de *l'analyse*, et non de la *conception*, que de s'intéresser et d'intégrer tout de suite de telles exigences dans un modèle. En d'autres termes, des contraintes techniques qui normalement mais surtout *idéalement* sont sa[1]tisfaites au plus tard, deviennent dans la logique de Booch, des objectifs de tout premier plan. À ce titre, leur prise en charge immédiate va de manière quasi certaine brider la spécification, c'est-à-dire que les modèles UML seront probablement plus difficiles à établir. À l'inverse, faire une spécification ouverte, et surtout en totalité ou par partie générique, c'est créer l'évolutivité. Cet investissement intellectuel a un coût qui, au risque de déranger beaucoup de puristes, ne peut pas toujours se justifier économiquement. En bilan, l'approche de Booch est difficilement contestable au regard de la réalité industrielle mais toute souplesse (sous la forme de contraintes techniques dont la satisfaction peut être différée) est bonne à prendre.Il n'y a pas pour ce problème de réponse standard, méthodique en l'occurrence. Pour notre application, le cahier des charges dit : « Le cycle et le mode de rafraîchissement de cette information ne sont pas connus car ils dépendent du dispositif retenu pour l'installation finale. En effet, en fonction de contraintes matérielles et logicielles ultérieurement définies (type de capteur, protocole de communication, etc.), la spécification doit être suffisamment flexible pour s'adapter à ces contraintes. » En fait, la valeur de la température ambiante à travers son mode de capture et de mise à jour encourage notre spécification à rester non pas vague mais ouverte. La fréquence d'apparition des occurrences de l'événement *ambient temperature changed* fixe par définition le cycle de rafraîchissement de la valeur de la température ambiante. Le mode d'acquisition, quant à lui, dépend de l'implémentation, elle-même dépendante du matériel et des couches logicielles mis en place.

1. Nous proposons en fait un simulateur et générons donc aléatoirement des valeurs de température ambiante au sein d'une classe Java *Temperature_sensor*.

Dans la problématique de délimitation du système, il y a donc intérêt, lorsque le cahier des charges le permet, à éviter des spécifications imposant des choix irréversibles. Par exemple, l'interface utilisateur est grossièrement maquettée en figure 5.1. À l'exception d'actions sommaires intitulées *display…* dans le *statechart* de la figure 5.10, aucun élément de notation UML (dans tous les diagrammes vus jusqu'à présent) n'est dédié et donc ne traite de l'interface utilisateur. Là encore, en figure 5.11, un acteur nommé *User interface* va juste mimer, montrer le positionnement de l'ensemble des entités de notre modèle UML par rapport à la nécessité d'entrer des commandes (pavé de dix boutons entre autres) et de visualiser des résultats. Nous voyons par la suite en implémentation qu'à l'exception de problèmes mineurs, ce choix est judicieux car la définition de l'interface utilisateur s'effectue simplement sur la base de la bibliothèque *Swing* de Java, cela indépendamment des classes codant les *State Machine Diagram*. De manière générale, faut-il modéliser en UML une interface graphique ? La réponse est non sauf si votre application est un jeu vidéo ou encore un système de visualisation élaboré (CAO, CFAO, SIG, etc.).

Patrons de spécification

Les schémas en figure 5.2 et en figure 5.10 sont incomplets si l'on ne précise pas ce qu'implique le fait que *Programmable thermostat* hérite de *Timer client*. Nous allons ici expliciter un protocole d'échange standard entre un *Timer* et un objet (*Timer client*) attendant de *Timer* des services de cadencement. Le but de cette manœuvre est triple :

- avoir des modèles UML prédéfinis et donc réutilisables dans des applications différentes (voir chapitre 6 où la même mécanique est mise en œuvre). C'est la notion de patron de spécification pour l'analyse (*Analysis Pattern*) qui est présentée dans [3]. L'idée est de capitaliser des modèles UML, éventuellement paramétrés, dont la différenciation d'une application à l'autre est insignifiante ou nulle. Le besoin de services de cadencement est, de manière évidente, indépendant de la domotique ;
- instrumenter le protocole au niveau code par une implémentation Java par défaut, en l'occurrence dans la bibliothèque *PauWare.Statecharts*. En clair, cette bibliothèque offre la classe abstraite *Timer_client* mais n'exclut pas d'autres implémentations comme celle disponible dans les EJB (interface *javax.ejb.TimerService* en particulier) ou dans CORBA (*Timer Event Service*). Néanmoins, dans *PauWare.Statecharts*, il existe une cohérence entre les services de cadencement, leur usage en particulier, et le fait qu'un objet de type *Timer client* a la plupart du temps son comportement décrit par un *statechart* ;
- simplifier une fois pour toutes les notations événementielles d'ordre temporel d'UML. Les annotations informelles dans les *Sequence Diagram* consistent à baliser certains intervalles entre deux événements avec du texte du genre *expression < 5 sec*. Dans les *State Machine Diagram*, les symboles *when* ou encore *after*[1] associés au métatype *TimeEvent* d'UML 2.x, rendent le niveau de

1. Alors que la notation *when* est maintenue dans UML 2.x, le terme *after* utilisé au chapitre 3 semble lui supprimé car jamais utilisé dans la documentation d'UML 2.x.

modélisation médiocre. En UML 2.x, les *Timing Diagram* par leur grande sobriété pour ne pas dire pauvreté, apportent peu. Notre approche gomme donc ces fioritures de notation au profit de l'utilisation canonique d'un couple *Timer/Timer client*.

Modélisation d'aspects temporels en UML

Avant d'en venir à notre approche, donnons brièvement, en supplément du chapitre 3, le cadre de modélisation des événements relatifs au temps en UML 2.x. Notre but est de montrer que, pour le cas qui nous intéresse, leur pouvoir d'expression est insuffisant. Pour cela, dans la figure 5.13, nous illustrons (sans conviction car nous pensons que ces notations sont bancales) la modélisation des aspects temporels « à la UML 2.x ». L'idée est d'imaginer, car ce n'est pas dans le cahier des charges initial, que le thermostat programmable a à charge le questionnement périodique du capteur de température, disons toutes les trente secondes (figure 5.13).

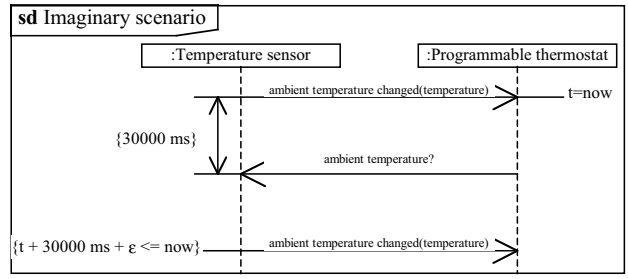


Figure 5.13 – Notations temporelles dans les scénarios en UML 2.x.

Différents métatypes UML 2.x sont à ce titre instanciés dont *DurationConstraint* (*e.g. {30 000 ms}*) et *TimeObservationAction* (*e.g. t=now*). Le scénario en figure 5.13 comporte des constructions de modélisation « propriétaire » au sens où elles sont propres aux *Sequence Diagram*. Cette approche est donc limitative. Sans aller au-delà en poursuivant les critiques, nous invitons le lecteur à reprendre et à « porter » toutes les notations temporelles de la figure 5.10 avec le formalisme offert par UML 2.x. Est-il réellement possible d'obtenir des modèles précis ? Le débat est ouvert… Nous préférons au lieu de répondre à cette question l'alternative de modélisation basée sur le patron *Timer/Timer client* qui suit.

Interaction entre *Timer* et *Timer client*

Un objet de type *Timer client* arme *Timer* via l'envoi de l'événement *to be set*. Le nombre de millisecondes est passé en paramètre comme par exemple dans l'envoi de *to be set(1 000)* (point noir en bas de l'automate, figure 5.10) pour faire des traitements toutes les secondes. Un autre argument (optionnel) est le contexte dans lequel l'armement s'opère (réception de *to be set(context,delay)* en figure 5.14). Ce contexte correspond à l'état courant (dans le *statechart*) de l'objet *Timer client* au moment où ce dernier requiert les services de cadencement. Ce contexte est utile à plusieurs titres. De nombreux *Timer* peuvent être lancés pour des raisons différentes

(cardinalité * vers *Timer* dans une deux associations en figure 5.14), ces dernières étant justement distinguées par le contexte, si précisé au démarrage. Au moment, du « claquage » du *Timer*, l'objet *Timer client* reçoit l'événement *time-out* avec en paramètre, le nombre de millisecondes qui avait été demandé et le contexte dans lequel la requête s'était faite. Ces deux paramètres en retour permettent à *Timer client* d'analyser pourquoi un *Timer* claque. Dans son cheminement (changement d'état dans son *statechart*), un objet *Timer client* peut ainsi fort bien ne plus être intéressé par l'information retournée par *Timer*. Pratiquement, un autre événement a pu entre-temps l'amener dans un nouvel état où les services de cadencement préalablement lancés deviennent caducs. Dans ce cas, il peut donc ne pas procéder à un traitement particulier de l'événement *time-out* qu'il reçoit néanmoins, sauf s'il avait désarmé *Timer* (événement *to be killed*) auparavant.

L'événement *to be killed* doit impérativement être envoyé pour stopper l'acquisition postérieure de services de cadencement. En l'absence de désarmement, outre une surcharge injustifiée, *Timer* continue d'envoyer l'événement *time-out*. Comme plusieurs armements associés à des contextes différents ont pu avoir lieu, il faut préciser comme argument de *to be killed*, le contexte que l'on veut tuer (figure 5.14). Pour des raisons de simplification, on tolère aussi les formes *to be set(1 000)* et *to be killed()* de la figure 5.10 par exemple, formes qui ne précisent pas de contexte, ce dernier étant nommé alors par *Timer*.

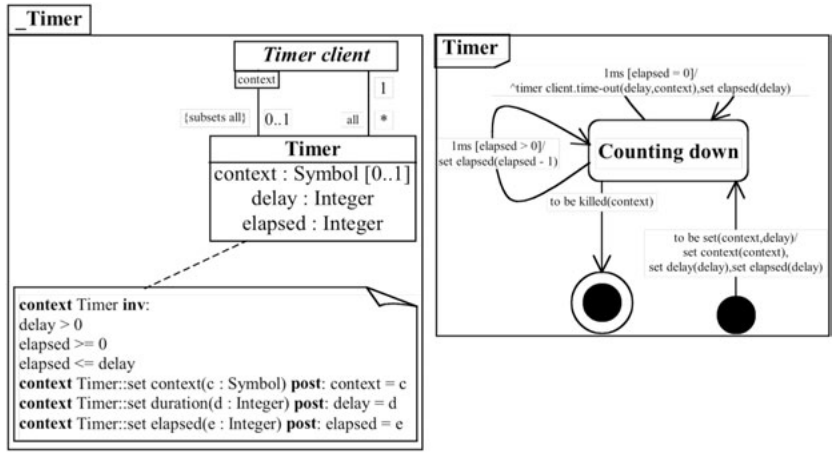


Figure 5.14 – Spécification du package *_Timer*.

Une spécification du même genre qu'en figure 5.14 se trouve dans [2] ainsi que dans [8]. En figure 5.14, la précision est le millième de seconde incarnée par l'événement *1ms* obtenu au niveau « système ». Par exemple, la machine virtuelle Java via sa partie *multithreading* offre les moyens d'une telle précision. À chaque apparition d'une occurrence de *1ms*, l'attribut *elapsed* est décrémenté de *1* jusqu'à atteindre la valeur *0* (garde *[elapsed = 0]*) qui fait que *Timer* invoque *time-out*. *Timer* est cyclique

car dans la foulée l'attribut *elapsed* est repositionné à la valeur *delay* (*set elapsed(delay)*) en enchaînement de l'envoi de *time-out*.

Sinon, divers invariants sont nécessaires au fonctionnement de l'automate en figure 5.14. Ceux qui sont triviaux sont insérés au sein d'une note attachée à *Timer*. Ajoutons les propriétés régissant l'évolution des deux associations entre *Timer* et *Timer client*. D'une part, tous les *Timer* d'un objet *Timer client* sont obtenus par une navigation nommée *all*. Pour un objet *Timer client* et un contexte donné, il y a ou n'y a pas de *Timer* associé (cardinalité *0..1*), celui étant inclus dans l'ensemble des *Timer* en cours (contrainte *{subsets all}*). D'autre part, au moment de l'armement, le lien qualifié par l'attribut *context* de *Timer*, est valorisé :

```
context Timer::to be set(context : Symbol,delay : Integer) inv:
-- Attention, bien distinguer le mot-clef context d'OCL du contexte d'un Timer!
   post: timer client[context] = self
```

De façon symétrique, l'événement *to be killed* met à jour le lien :

```
context Timer::to be killed(context : Symbol) inv:
   pre: timer client[context] = self
   post: timer client[context].oclIsUndefined
```

Subtilités de modélisation

Dans cette section, nous passons en revue quelques règles clés de modélisation concernant les *Statecharts* de Harel, testant pour cela ce qu'offre UML en termes de construction de modélisation avancées. Notre étude de cas est suffisamment complexe pour se rendre compte qu'UML n'est souvent qu'un moyen de notation et qu'il est opportun de bien vérifier ce que « dit » réellement un modèle. La notion de *statechart* exécutable [5] ou plus généralement de spécification exécutable permet de simuler[1] l'évolution des automates dans le temps. C'est un véritable support de test des *State Machine Diagram* qui existe dans des outils comme Rhapsody (voir section « Webographie » en fin de chapitre), outil auquel Harel contribue. UML développe une sémantique assez confuse des *State Machine Diagram* d'où la nécessité d'interpréter les automates bâtis le plus sûrement possible.

Complémentarité des gardes

Cette qualité est parfois laissée pour compte dans UML. Par exemple en figure 5.10, les deux transitions de sortie de l'état *Ambient temperature displaying* étiquetées par l'événement *time-out* sont gardées par *[alternately = 2]* et *[alternately <> 2]*. On a de façon rassurante *[alternately = 2]* $\vee$ *[alternately <> 2] = true*, levant l'ambiguïté quant aux traitements à réaliser lors de la réaction à *time-out* pour l'état *Ambient temperature displaying*. Néanmoins, il n'y a pas toujours, pour des raisons de simplicité et de clarté, besoin de s'assurer ou de garantir, pour état donné et un événement sur une transition sortant de cet état, que *= true*. Par exemple, dans l'état *Control* de la figure 5.10, on n'a délibérément pas écrit et donc ajouté une nouvelle garde en relation

1. À grande échelle, cela implique l'utilisation d'ateliers de génie logiciel.

avec l'événement *ambient temperature changed* pour vérifier = *true*. Si cette garde avait été introduite, elle aurait cette forme :

```
[season switch in Is off
or
season switch in Is cool and ambient temperature <= target temperature
   ➥ and ambient temperature.asCelsius >= (target temperature.asCelsius - delta)
or
season switch in Is heat and ambient temperature >= target temperature
   ➥ and ambient temperature.asCelsius <= (target temperature.asCelsius + delta)]
```

Cette garde, par sa longueur et sa relative inintelligibilité, parasite la compréhension de la situation au sens où elle n'a aucune pertinence dans la logique de commande des différents relais de puissance. Si elle est vraie (il suffit que le commutateur de saison soit dans l'état *Is off* par exemple) il n'y a en fait rien à faire. En UML, cela signifie que l'on n'utilise pas le / pour lancer des actions ou encore envoyer des événements.

De manière générale, il faut manipuler les gardes avec précaution. Si des expressions conditionnelles correspondent à des états critiques du système ou à des états d'erreur, peuvent-elles apparaître comme des gardes dans la spécification ? Rien n'est précisé en UML alors que la réponse est à notre avis « non ». En effet, l'évaluation à « vrai » d'une garde établit un chemin à suivre dans le graphe dont les sommets sont les états du *statechart*. L'occurrence de l'événement est consommée, et la garde vérifiée fait changer d'état. Au pire, la garde n'est pas avérée et l'on aboutit à une consommation de l'occurrence de l'événement sans changement d'état. Cela n'est pas à considérer comme une erreur. Comment alors spécifier que des événements ne doivent pas se produire dans certaines circonstances ? Cela est traité avec le mécanisme d'*allowed event* dans [2]. Ainsi, sous l'hypothèse que la garde ci-dessus n'est pas écrite, l'événement *ambient temperature changed* déclaré *allow* dans l'état *Ambient temperature displaying* stipule que l'arrivée de l'événement alors qu'aucune garde n'est vraie, a un effet *neutre*. Au contraire, l'omission de la déclaration *allow* caractérise un état d'erreur. L'arrivée de l'événement alors qu'aucune garde n'est vraie, sous-entend donc un dysfonctionnement : « *Guards, then, can be used for two purposes. When the logical or of the guards for an event leaving a state is true, the guards are selecting a path through the machine. When the logical or is other than true (for example, where there is only one guarded transition for the event), the guards are acting as pre-conditions as well.* » [2, p. 99]. Comment faire en UML ? L'utilisation des points de jonction (*junction points*) et de la condition prédéfinie *[else]* peut être une solution. Cette dernière peut servir à sélectionner un chemin dans le graphe lorsque l'évaluation à « vrai » de toutes les gardes a échoué. La garde que nous avons écrite et jugée non pertinente, correspond justement en l'occurrence à un cas type de *[else]*.

Différenciation événement et action

Cela nous amène naturellement aux préconditions et post-conditions des événements ainsi que celles des actions éventuelles exécutées à l'issue des événements. La différence entre le traitement par post-conditions, par nature déclaratif, et le traitement par actions, par nature impératif, gagne à se fonder sur des critères de fiabilité.

En fait, les calculs (au sens général du terme) à assurer peuvent être représentés naturellement par les opérations (partie basse des boîtes dans les *Class Diagram*). Celles-ci sont considérées comme des actions si elles sont ininterruptibles ou des activités (syntaxe : *do/*) si elles sont interruptibles. Une autre manière est d'associer des post-conditions aux événements pour signifier l'avancement des calculs à l'issue de l'apparition des événements. Comment alors méthodiquement choisir entre ces deux outils pour modéliser les traitements en général ?

L'idée qui nous paraît séduisante est de considérer que le déroulement d'une opération, et par là même un quelconque échec, ne doit pas être préjudiciable au fonctionnement global du système. Le contenu des opérations est décrit par ailleurs ou plus tard dans le processus de développement, c'est selon, sous forme de pseudo-code par exemple. Ces contenus sont des algorithmes complexes au sens transformationnel (transformation de données en l'occurrence) mais n'ont pas de nécessité de contrôle extérieur : ils doivent disposer de toutes les données en local pour calculer les résultats. En revanche, les post-conditions sont des instantanés et renseignent en partie sur l'état du système. Une violation de post-condition doit donc mieux être considérée comme une erreur à corriger impérativement pour que le système soit en mesure de continuer. Les post-conditions des événements, des actions et des activités, lorsqu'elles existent pour ces dernières, doivent donc guider la politique d'amélioration de la robustesse à l'implémentation, via le mécanisme de gestion des exceptions, à ce jour assez standard d'un langage de programmation objet à l'autre.

Communication intra-objet

En figure 5.10, les événements *temp up* et *temp down* ont trois types de conséquences majeures dans l'état *Operate*. Tout d'abord, ils incrémentent et décrémentent la température cible courante et ont un impact direct sur l'affichage car l'utilisateur s'attend à voir les valeurs des températures se modifier à l'écran. À cet effet, une réactivité est prévue pour l'état *Target temperature displaying* lors d'occurrences de *temp up* et de *temp down*. Dans le *statechart*, l'état *Run* est orthogonal à l'état *Target temperature displaying* et *Run* présente aussi une réactivité à *temp up* et à *temp down*. Le choix fait est que les actions de montée et de descente de la température cible sont lancées sur apparition de *temp up*, respectivement *temp down*, dans l'état *Run*. Le contrôle de l'affichage est pris en compte lors de l'arrivée de ces deux mêmes événements dans l'état *Target temperature displaying*. Une occurrence de *temp up*, par exemple, a donc le pouvoir d'exciter deux traitements. Une erreur de spécification aurait notamment été d'incrémenter la température cible par réaction à *temp up* dans l'état *Target temperature displaying* alors que c'est déjà fait dans l'état *Run*. La troisième et dernière conséquence est d'agir sur le contrôle du chauffage ou du conditionneur d'air car la température cible ayant *a priori* augmenté ou diminué, il y a lieu de réguler l'atmosphère. L'événement *target temperature changed* est une communication intra-objet. Il est émis en réaction à *temp up* et à *temp down* dans l'état *Target temperature displaying* (voir l'expression : ^*self.target temperature changed*). Le caractère intra-objet résulte du fait que *Programmable thermostat* s'envoie cet événement à lui-même : il est en l'occurrence reçu et donc traité dans l'état *Control*.

Le principe de la communication intra-objet est donc fondamentalement basé sur des échanges entre états et seulement entre états orthogonaux d'un même *statechart*. Une alternative aurait consisté à établir une communication entre les processus *Run* et *Hold* d'une part, à destination du processus *Target temperature displaying* d'autre part. En d'autres termes, cela revient à générer un événement comme *target temperature changed* pour réactualiser l'affichage uniquement si la température a *effectivement* augmenté ou diminué. Ce n'est pas ce qui a été modélisé réellement dans la figure 5.10.

La communication intra-objet, même si elle ajoute un surcoût en communication, peut faire gagner en efficacité et parfois en rationalité. De manière générale, l'arrivée d'un événement dans un *statechart* est soit partagée sur les états orthogonaux candidats à son interprétation, soit réceptionnée à un endroit précis pour être ensuite réorientée via une communication intra-objet.Une autre alternative aurait été de faire réagir l'état *Control* aux événements *temp up* et *temp down*, c'est-à-dire réguler le chauffage ou le conditionneur d'air dès lors que la température cible bouge. L'intérêt de la modélisation retenue en figure 5.10 est que *Control* n'est véritablement « dérangé » que si la température cible a réellement changé. En effet, une garde telle que *[target temperature > Min]* évaluée à « faux » va empêcher l'envoi de l'événement *target temperature changed*.

Masquage de la réactivité

Soit un état *S* présentant une réactivité à un événement *e*. En d'autres termes, si *e* se produit, on quitte *S* pour y revenir ou pour aller dans un autre état. Ce comportement peut être qualifié de « niveau général » si *S* possède en plus des sous-états et qu'il est nécessaire d'affiner ce qu'il y a à faire pour tout ou partie de ces sous-états. Le principe est que la réactivité décrite à un niveau supérieur (comportement généraliste attaché à *S*) est masquée par la réactivité, si elle est spécifiée, pour les sous-états.

Par exemple en figure 5.15 (tout comme en figure 5.10), l'événement *set clock* a pour origine le contour de *Programmable thermostat* et pour destination l'état *Set current minute*. Cela signifie que si l'on est dans l'état *Set current minute* et que *set clock* se produit, on boucle sur *Set current minute*. De plus, si l'on est dans l'état *Set current hour* et que *set clock* se produit, on transite dans *Set current minute*. En fait, ce comportement ne respecte pas le cahier des charges, car si l'on est dans l'état *Set current minute* et que *set clock* se produit, on devrait aller dans *Set current hour*. Pour corriger ce problème, il faut ajouter en figure 5.15 (tout comme en figure 5.10), une réactivité qui annule et remplace le comportement par défaut. Cette réactivité est une transition de *Set current minute* à *Set current hour* étiquetée par *set clock*. En clair, il y a deux traitements préconisés pour *set clock*, dont un plus profond qui masque celui de niveau directement supérieur aussi appelé « général ».

Le principe de restriction et/ou d'extension de la réactivité devient délicat si des actions sont associées à un même événement à la fois aux niveaux supérieur et inférieur. Par exemple en figure 5.10, à chaque occurrence de *view program* il faut changer de période (créneau horaire de la table de programmation). L'expression *view*

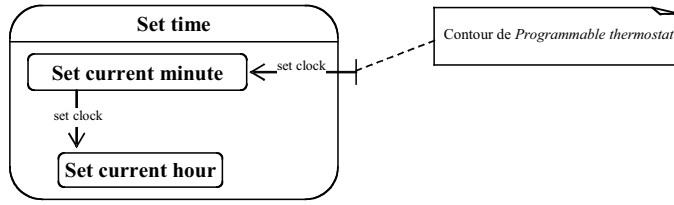


Figure 5.15 – Raffinement de la réactivité.

program/ set period(1) est le comportement général. Elle se trouve au milieu, tout à droite de la figure 5.10 sur une transition partant du contour de *Programmable thermostat*. Elle est néanmoins insuffisante. En effet, si l'occurrence de *view program* arrive alors que l'on est déjà précisément dans l'état *Set period*, il faut affiner le comportement en ajoutant des conditions comme suit : *view program [period = 8]/ set period(1)* et *view program [period <> 8]/ set period(period + 1)*. Ces deux dernières réactivités masquent celle de niveau supérieur mais inhibent aussi l'action qui lui est associée, *i.e. set period(1)* n'a pas lieu si l'on est déjà dans *Set period* : il faut tester la valeur de *period* au préalable avant de déterminer quelle nouvelle valeur lui est assignée. Deux autotransitions sur *Set period* assurent donc ce job en figure 5.10. En aparté, on remarque néanmoins que l'expression *view program [period = 8]/ set period(1)* étiquetant une des deux autotransitions peut être omise. Le maintien de sa présence n'introduit pas d'erreur mais pose plus volontiers un problème de redondance syntaxique. On la garde néanmoins pour une meilleure compréhension car d'un point de vue sémantique, la spécification reste correcte.

Redondance

Il existe une réactivité à l'événement *view program* quel que soit le sous-état de *Programmable thermostat* dans lequel on se trouve. Dans un tel contexte, une flèche, transition en l'occurrence, avec le label *view program* a pour origine le contour de *Programmable thermostat* (figure 5.16). Ainsi, dans tout sous-état de *Programmable thermostat*, à l'apparition de *view program*, on transite entre autres dans *Period and program time refreshing*. Il faut faire attention que dans la figure 5.16 (à droite), la flèche ait pour fin le superétat *direct* de *Period and program time refreshing*, c'est-à-dire *Program refreshing*. Il n'y a pas d'ambiguïté concernant les états à activer lors d'une l'occurrence de *view program* car *Program refreshing* n'a que deux sous-états orthogonaux (*i.e.* parallèles).

Dans la figure 5.16, l'autre transition étiquetée par *view program* de l'état *Period and program time refreshing* sur lui-même paraît donc inutile puisqu'elle dit sémantiquement la même chose. Au demeurant, les actions déclenchées sont les mêmes, *i.e.* la remise à zéro de la variable *no input* (modélisée par la post-condition de *view program*), ainsi que le rafraîchissement à l'écran de la période et du créneau horaire associé à cette période (action en entrée intitulée *display period and program time*).

L'idée est donc de simplifier le *statechart* en figure 5.16 de manière à aboutir à la figure 5.17 : l'autotransition sur *Period and program time refreshing* étiquetée par *view program* a disparu. Il est bien évident que toute simplification est salutaire car elle

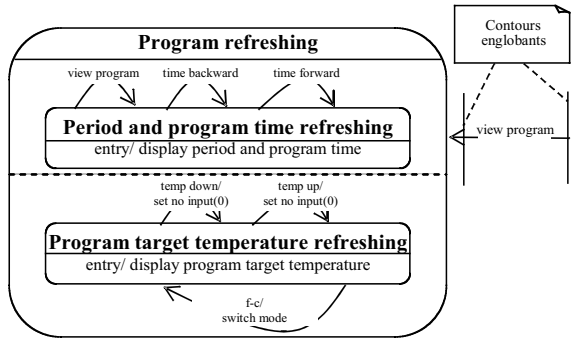


Figure 5.16 – Redondance possible.

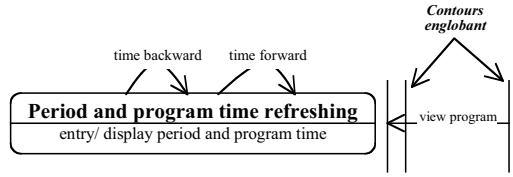


Figure 5.17 – Suppression de la redondance de la figure 5.16.

améliore la lisibilité et engendre moins de travail de conception et d'implémentation. *A contrario*, si la redondance perdure jusqu'au code, la maintenance se compliquera d'autant.

Il existe néanmoins un danger non négligeable quant à la bonne compréhension du fragment de modèle en figure 5.17. En d'autres termes, le passage de la figure 5.16 à la figure 5.17 demande de regarder en détail partout où apparaît l'événement *view program* dans la figure 5.10 avant de procéder à des simplifications. Une analyse soigneuse et en totalité du *statechart* en figure 5.10 montre ainsi que la réactivité à l'événement *view program* est aussi redéfinie au sein de l'état *Set program* (figure 5.18), qui est en parallèle de l'état *Program refreshing*. Dans le sous-état de *Set program* appelé *Set period*, une garde est posée sur la variable *period*. Le comportement général décrit relatif à l'événement *view program* est donc inhibé (masqué) dès lors que l'on se trouve déjà dans *Set program* (*Set period* en l'occurrence) et donc dans *Program refreshing* puisqu'ils sont tous deux en parallèle.

En clair, si l'on est dans *Set period*, on active l'action *set period* en fonction de la valeur courante de la variable *period*. Si l'on n'est pas dans cet état (on arrive d'*Operate* par exemple), on active l'action *set period* avec le paramètre *1* sans poser de garde sur la variable *period* : c'est l'expression *view program/ set period(1)* que l'on voit tout à droite de la figure 5.18. Contrairement à la logique de passage de la figure 5.16 à la figure 5.17, on ne peut donc pas cette fois supprimer ladite expression car elle ne fait pas référence à un simple événement mais à un événement et une action associée (*set period*), dont le lancement et les paramètres varient en fonction du con-

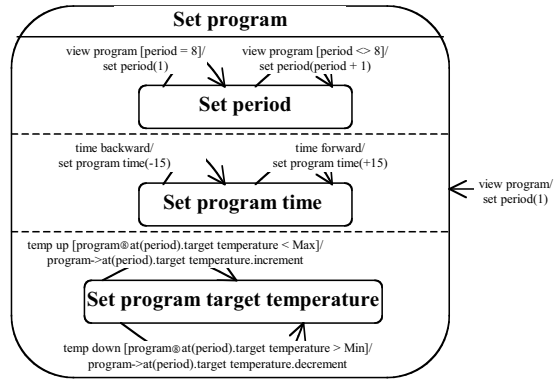


Figure 5.18 – Redondance cohabitant avec masquage.

texte. Il y a donc en figure 5.18 une redondance d'apparence, syntaxique disons, mais pas sémantique !

Anticipant la partie implémentation de cette étude de cas, voyons comment l'usage de la bibliothèque *PauWare.Statecharts* et plus précisément l'intérieur de la fonction *view_program* dans la classe *Programmable_thermostat* précisent notre propos global :

```
public void view_program() throws Statechart_exception {
   Object[] args = new Object[1];
   args[0] = new Integer(1);
   Programmable_thermostat.fires(_Programmable_thermostat,_Set_program,true,
     ➥ this,"set_period",args);
   args[0] = new Integer(1);
   Programmable_thermostat.fires(_Set_period,_Set_period,_period ==
     ➥ 8,this,"set_period",args);
   args[0] = new Integer(_period + 1);
   Programmable_thermostat.fires(_Set_period,_Set_period,_period !=
     ➥ 8,this,"set_period",args);
   Programmable_thermostat.fires(_Programmable_thermostat,_Program_refreshing);
   Programmable_thermostat.fires(_Program_refreshing,_Program_refreshing);
   Programmable_thermostat.run_to_completion();
   // post-condition(s)
   no_input = 0;
}
```

Dans le code qui précède, les transitions sont décrites via la méthode *fires* de la classe *Statechart_monitor* dont *_Programmable_thermostat* est une instance. La méthode *run_to_completion* a à charge le franchissement d'une phase partant d'un état global stable du *statechart* pour arriver à un autre : c'est un cycle d'interprétation ininterruptible au sens où aucun événement ne peut arriver et donc être traité dans cette phase.

La redondance est liée à la présence concomitante dans le code des transitions *_Programmable_thermostat/_Program_refreshing* et *_Program_refreshing/_Program_refreshing*. En regard de la simplification salutaire de la figure 5.16, la ligne de code

_Programmable_thermostat.fires(_Program_refreshing,_Program_refreshing); semble donc être l'implémentation de la figure 5.16 alors que l'on s'attend à celle de la figure 5.18, qui verrait sa disparition. Cette ligne de code doit néanmoins être préservée en liaison avec le fonctionnement actuel de la bibliothèque *PauWare.Statecharts*. C'est certes difficile à comprendre et peut-être contradictoire mais nous maintenons le modèle de la figure 5.18 comme un « meilleur » modèle que celui de la figure 5.16. Le défaut de la bibliothèque est qu'il faut forcer une sortie puis une entrée sur *Program refreshing* de manière à forcer le lancement de l'opération *display period and program time*. En l'absence de la ligne de code *_Programmable_thermostat.fires(_Program-_refreshing,_Program_refreshing);*, le moteur d'exécution va déterminer que l'on ne change pas d'état : on reste dans *Program refreshing* si on y était déjà ce qui est différent d'en sortir pour y revenir ! En revanche, les deux transitions *_Set_period/_Set_period* distinguées par leur garde s'ajoutent à la plus générale *_Programmable_-thermostat/_Set_program* qui sera inhibée (masquée) si l'automate est déjà dans *Set period*.

Communication interobjet et couplage

La figure 5.11 donne par rapport à la figure 5.2 un degré de couplage entre classes plus strict et plus précis. Par exemple, le thermostat programmable connaît le relais de puissance du conditionneur d'air (il émet à son endroit les messages *run* et *stop*) alors qu'il n'y a pas de communication inverse. Cependant, les flux de contrôle/communication retenus auraient pu être différents. Le degré de couplage synthétisé dans la figure 5.11 peut donc s'accroître sans raison si l'on n'y prête pas attention.

Il n'est ainsi, par exemple, pas souhaitable de coupler *Air conditioner relay* avec *Fan relay* (figure 5.19). La motivation présidant à la création d'un tel lien est qu'en mode automatique, le ventilateur se lance et s'arrête au gré du lancement et de l'arrêt du conditionneur d'air (c'est aussi vrai et symétrique pour le chauffage). La synchronisation entre les deux dispositifs, puisqu'elle figure dans le cahier des charges, est actuellement gérée via le thermostat programmable qui joue le rôle d'intermédiaire (figure 5.10). L'association dérivée (rôle *my collaborating fan relay* préfixé de /) en figure 5.19 est l'ajout d'un couplage supplémentaire. Elle est par définition dynamiquement calculable (*i.e.* ***context*** *Air conditioner relay* ***inv****: fan relay = programmable thermostat.fan relay*). Elle pourrait servir à écrire les expressions ^*my collaborating fan relay.run* ou ^*my collaborating fan relay.stop* dans l'automate d'*Air conditioner relay* ce qui n'est pas le choix que nous avons fait (figure 5.8). En faisant cela, on déplace la responsabilité de commande du ventilateur dans le relais de puissance du conditionneur d'air. À grande échelle, une telle démarche accroît les coûts de maintenance : une dépendance logicielle entre les deux classes est créée qui fait qu'elles doivent être maintenues l'une en fonction de l'autre. L'évolution globale de l'application via par exemple une autre politique de commande oblige plus généralement à gérer l'évolution de plusieurs classes au lieu de se limiter à *Programmable thermostat*.

Un autre exemple est le commutateur de ventilation (*Fan switch*) qui activé en mode forcé, événement reçu *on*, informe le thermostat programmable (envoi de *fan switch turned on* à l'objet *programmable thermostat*, figure 5.6). Le thermostat pro-

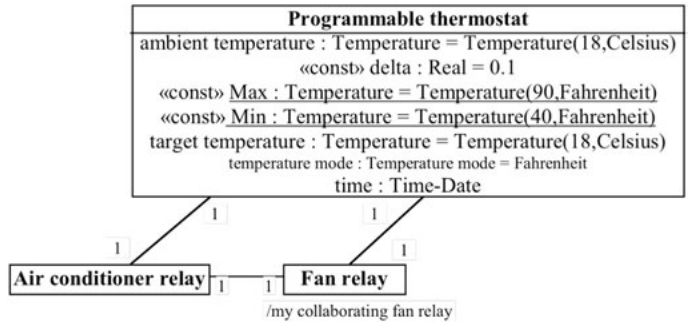


Figure 5.19 – Couplage fort.

grammable réagit à cet événement pour donner une consigne au relais de puissance qui commande le ventilateur. En effet, en figure 5.10, dans l'état *Control* du thermostat programmable, la réception de l'événement *fan switch turned on* est suivie de l'expression ^*fan relay.run*. Dans la solution actuelle, il n'y a donc pas de lien avéré entre *Fan switch* et *Fan relay* bien que le cahier des charges sous-entende que l'activation du premier doit immédiatement être suivie d'un ordre pour le second. On peut imaginer une communication directe entre le commutateur de ventilation et le relais de puissance du ventilateur, *i.e.* dans le *statechart* du commutateur de ventilation, on verrait alors *on/ ^my fan relay.run*. Cette expression de navigation OCL n'est valide qu'à la condition de fournir l'association dérivée en figure 5.20. Quelle que soit la solution choisie, nous préconisons dans l'ensemble d'expliciter dans les *Class Diagram* (sous forme d'associations de base ou dérivées) les supports des flux de communications/contrôle des *State Machine Diagram*, *Sequence Diagram* et *Collaboration Diagram*.

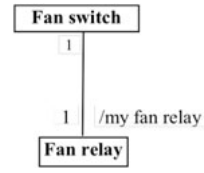


Figure 5.20 – Autre couplage fort.

Une autre façon de procéder, autre forme de modélisation, est en l'occurrence d'écrire *on/^programmable thermostat.fan relay.run* dans le *statechart* du commutateur de ventilation. Elle a l'avantage de ne pas demander la description de l'association additionnelle en figure 5.20 mais reste facteur de complication des communications.

En résumé, plusieurs communications interobjets sont donc possibles mais pas toujours souhaitables De manière générale, un système hautement coopératif fera appel à de nombreuses communications interobjets et donc à des objets souvent simples mais avec un graphe d'interaction plutôt complexe. À l'opposé, ce système de domotique tel que nous l'avons construit est centralisé. Il est simple quant au cou-

plage (figure 5.11) mais conduit à un objet très complexe en figure 5.10 dont la manipulation n'est pas aisée tant au niveau de la modélisation que de l'implémentation. Un autre défaut est sa compréhension. L'automate de la figure 5.10 est difficile à ingérer puis digérer ! Encore une fois, tout n'est qu'une question d'équilibre dans la répartition des responsabilités entre objets et conséquemment, des collaborations qui en résultent ou justement qui causent les devoirs de responsabilité. Une lecture ou relecture de [9] est ici salutaire.

Factorisation des *State Machine Diagram*

La factorisation des *State Machine Diagram* (voir par exemple [5] pour une ébauche du problème) ou de manière générale leur utilisation conjointe avec l'héritage et le polymorphisme, est un problème épineux. Tout en évoquant le problème dans le chapitre 3, nous n'avons pas fourni d'automate de classe abstraite (la classe *Account* en l'occurrence).

Ici, à titre d'illustration, nous avons en figure 5.2 la classe abstraire *Relay* dont héritent trois sous-types. Les automates de ces sous-types sont en figure 5.8 et se ressemblent de manière évidente. En figure 5.21, nous donnons le *statechart* de la classe *Relay*. En fait, si l'on compare la figure 5.21 à la figure 5.8, la topologie est la même : mêmes sommets au nom près, mêmes arcs. La réactivité aux événements est la même (mêmes transitions et même noms d'événement déclencheur) à l'exception de l'événement reçu *stop* sur *Fan relay* qui est porteur d'un attribut booléen (*condition*) utilisé comme garde. On peut voir cette spécificité comme une réactivité spécialisée, une sorte de masquage de la réactivité standard de la figure 5.21 où *stop* mène sans condition de l'état *Is on* à l'état *Is off*. Les actions mises en œuvre en entrée d'état (mot-clef *entry*) en l'occurrence, sont aussi les mêmes : *activate* pour l'activation du relais de puissance physique incarné par la classe et *disactivate* pour la désactivation du même dispositif.

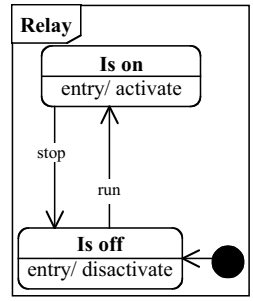


Figure 5.21 – *Statechart* de la classe abstraite *Relay*.

La démarche suivie ici est purement didactique au sens où elle n'est pas réaliste au plan industriel. Même s'il y a des résultats scientifiques de qualité sur le sujet, ils restent à notre sens difficilement transférables. Pour preuve, à notre connaissance, aucun AGL ne traite correctement le problème, *i.e.* l'appariement de *State Machine Diagram* pour mieux factoriser les comportements d'objet et donc réutiliser en géné-

ral. Dans l'utilisation de la bibliothèque *PauWare.Statecharts*, nous sommes cependant allé jusqu'au code avec un niveau de factorisation correct :

```
abstract public class Relay {
   // les états sont créés à partir de la classe Java prédéfinie Statechart
   protected Statechart _Is_off;
   protected Statechart _Is_on;
   // l'automate est incarné par une variable de type Statechart_monitor
   // on lui donne par convention, le nom de la classe préfixée d'un _
   protected Statechart_monitor _Relay;
   // la création consiste à s'intéresser aux attributs et autres propriétés
   // (associations…) trouvées dans les Class Diagram
   protected void init_structure() throws Statechart_exception {
   }
   // la création consiste aussi à instancier et configurer les états,
   // i.e. les dépendances entre eux : xor (exclusion), and (parallélisme)
   //  et emboîtement
   protected void init_behavior() throws Statechart_exception {
      _Is_off = new Statechart("Is off");
      _Is_on = new Statechart("Is on");
      // la relation d'emboîtement s'opère via les constructeurs,
      // i.e. _Is_off et _Is_on sont des sous-états de _Relay
      _Relay = new Statechart_monitor(_Is_off.xor(_Is_on),"Relay");
   }
   protected Relay() throws Statechart_exception {
      init_structure();
      init_behavior();
   }
   // activités
   abstract public void activate();
   // actions
   abstract public void disactivate();
   // le traitement des événements qui suivent ne sont pas abstraits
   // car le comportement est parfaitement déterministe dans la figure 5.21
   public void run() throws Statechart_exception {
      _Relay.fires(_Is_off,_Is_on);
      _Relay.run_to_completion();
   }
   public void stop() throws Statechart_exception {
      _Relay.fires(_Is_on,_Is_off);
      _Relay.run_to_completion();
   }
}
```

La classe *Relay* n'étant jamais instanciée directement puisqu'elle est abstraite, les déclenchements en entrée d'état (*entry/*), sortie d'état (*exit/*) et durant l'état (*do/*) sont différés dans les sous-types.

Historique

Le principe d'historique (cf. chapitre 3) consiste à, quand on retourne dans un état S, de retrouver le sous-état de S dans lequel on était lors du précédent passage dans S. Cette convention est extrêmement pratique pour simplifier les modèles. Pour ce qui

est de la notation, on introduit une construction de modélisation (un *H* cerclé en l'occurrence) que l'on associe graphiquement à *S* pour dire qu'il y a une mémoire.

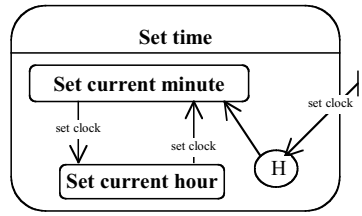


Figure 5.22 – Utilisation des possibilités d'historique.

L'intérêt ici ergonomique (au niveau de l'interface utilisateur de l'application finale) du *H* est de retrouver le réglage des minutes, si c'est cela que l'utilisateur faisait la dernière fois qu'il réglait l'horloge interne du thermostat programmable (idem pour les heures). Sinon, la spécification du basculement de *Set current hour* à *Set current minute* et de son inverse reste nécessaire. Cette spécification complète le *H* qui donne le comportement si l'on arrive depuis l'extérieur de *Set time* au moment où une occurrence de *set clock* se produit. Elle donne le comportement si l'on est déjà dans *Set time*. Pour information, notons que la bibliothèque *PauWare.Statecharts* n'offre pas de fonctionnalité d'historisation.

5.4 CUEILLETTE DES OBJETS, INGÉNIERIE DES BESOINS

Au contraire de l'étude de cas du chapitre 4, nous n'avons pas exposé une démarche par laquelle les objets sont trouvés (« cueillis »). Mais dans cet exemple en domotique, comment sommes-nous passés du cahier des charges aux modèles, et plus précisément comment les objets ont-ils été déterminés ? Il n'y a pas eu véritablement de « cueillette » (*i.e.* tout type d'objet établi à un moment de la réflexion qui préside à l'élaboration d'un *Class Diagram* reste à jamais élément de ce *Class Diagram*). De nombreuses itérations ont donc été nécessaires avant de stabiliser les modèles. En d'autres termes, on peut pressentir un type X, le représenter dans un *Class Diagram*, le lier éventuellement à d'autres types et au moment de la construction de son *statechart*, le rayer finalement du *Class Diagram* par le fait que son comportement se résume à un seul état, par exemple.

Nous abordons l'aspect démarche d'ingénierie des besoins plutôt que l'aspect langage de modélisation et entrons donc dans un cadre moins rationnel. En ce sens, le choix d'un objet plutôt qu'un autre ainsi que le choix des propriétés des objets ne relèvent pas d'une quelconque méthode mais véritablement de l'empirisme. En outre, cette étude de cas ne s'appuie pas sur la technique des *Use Case* — dédiée justement à l'ingénierie des besoins — pour aboutir à des modèles complets et cohé-

rents, donnant une implémentation Java concise, claire et, nous l'espérons, robuste et évolutive, respectant les critères de qualité logicielle évoqués au chapitre 1. Néanmoins, quelques heuristiques intéressantes peuvent contribuer à la fabrication des modèles et il peut être bénéfique de discuter et donc reconsidérer l'absence ou la présence de tel ou tel type d'objet.

Si l'on procède intuitivement, les types d'objet de la situation sont : le thermostat programmable, la table des températures cibles et leurs créneaux horaires associés, le relais de puissance du chauffage, le relais de puissance du conditionneur d'air, le relais de puissance du ventilateur, le témoin d'activité, le commutateur de saison, le commutateur de ventilation et pourquoi pas le capteur de température, l'interface utilisateur, l'utilisateur et le pavé de dix boutons, voire même le type « bouton-poussoir » avec dix instances (*temp up*, *temp down*...). Où s'arrêter dans la décomposition et le détail des objets ? Nous n'avons pas en figure 5.2 le type *Button pad* (pavé de boutons). Par ailleurs, puisque c'est l'intuition qui nous guide, en aucun cas la liste d'« entités perçues » qui précède ne peut être considérée exhaustive ou tout simplement adéquate.

Les types d'objet du système logiciel (en objet, les classes de l'architecture logicielle) sont rarement les types d'objet de la situation (objets conceptuels des modèles d'analyse), plus exactement, leur intersection est non vide ; mais comment la caractériser ? À titre d'illustration, Rumbaugh *et al.* s'appliquent méthodiquement à cette tâche dans [7, p. 153-169] ou encore Wirfs-Brock *et al.* dans [9]. En essayant de doter de propriétés les types d'objet de la situation, prenons dans la liste ci-avant « l'utilisateur » par exemple, qui possède la propriété « âge ». Telle est la situation mais cette propriété est-elle pertinente pour le système logiciel ? Non, bien évidemment. L'utilisateur n'est caractérisable que par le fait qu'il appuie sur le bouton *Temp up* par exemple. Ce phénomène est plus intéressant. Est-ce l'utilisateur qui génère l'événement *Temp up* ou le pavé de dix boutons ? L'utilisateur reçoit-il des événements, par exemple : l'affichage a changé, la valeur de la température ambiante n'est plus la même ? N'est-ce pas plutôt l'interface utilisateur qui reçoit cet événement depuis le thermostat programmable, lui-même l'ayant reçu d'un capteur de température (événement *ambient temperature changed(temperature)* de *Temperature sensor* à *Programmable thermostat* en figure 5.11) ? L'interface utilisateur le génère plutôt (choix fait en figure 5.11), non ? Quel imbroglio ! Comment s'en sortir ?

Ce que nous voulons mettre en lumière en accentuant délibérément la confusion, est que répondre à ces questions va affirmer ou infirmer la présence d'un type d'objet *User* (l'utilisateur) et/ou un type d'objet *User interface* (l'interface utilisateur) dans un *Class Diagram*, puis conséquemment dans d'autres types de modèle. Dans les faits, ni *User*, ni *User interface* ne font partie du modèle de la figure 5.2. *User interface* intervient juste comme objet « extérieur » (*i.e.* « aux frontières du système ») ayant son propre processus de contrôle (notion d'objet actif) en figure 5.11. Pouvons-nous démontrer de manière définitive que c'est « le » bon choix, le choix unique ? Non, certainement pas.

Les grands gurus des méthodes de développement logiciel n'ont eux-mêmes pas la réponse alors que beaucoup d'industriels pensent bénéficier, spirituellement,

métaphysiquement, grâce à l'approche objet, des réponses à ces questions d'ingénierie des besoins. À cet égard, il est assez remarquable de constater que sur une étude de cas bancaire où un utilisateur insère une carte dans un distributeur d'argent, Rumbaugh *et al.*, au travers la spécification présentée dans [7, p. 168] récuse *User* comme type d'objet du système logiciel, là où Wirfs-Brock *et al.*, dans [9] voient *User* comme un type d'objet. Dans le même esprit, la spécification originale OMT de ce système de domotique récuse le capteur de température comme type d'objet d'un *Object Model* (*i.e.* un *Class Diagram* en UML) [7, p. 107], là où Booch dans [1], sur une étude de cas de même nature, considère *Temperature sensor* comme un type d'objet. La conclusion à tirer, mais ce n'est pas une surprise car nous l'avons avancé déjà plusieurs fois dans cet ouvrage, est que la modélisation objet et UML sont loin d'être bâtis sur des sciences exactes.

Pour le cas qui nous intéresse, une solution revient à considérer l'utilisateur, le capteur de température, le pavé de dix boutons… comme des éléments extérieurs à la « situation [1]» d'étude. Le système analysé n'est pas le système logiciel (ou système conçu) et en ce sens voir le capteur de température comme agent extérieur en analyse (figure 5.11) n'empêche pas par la suite son existence en conception et implémentation. Pourquoi évincer les objets similaires au capteur de température ? Parce que d'un point de vue fonctionnel, ils n'assurent finalement aucune fonction (aucune opération) sinon des fonctions de génération d'événements. Même si l'utilisateur reçoit des événements, quel est l'intérêt de ce phénomène général ? Peu importe l'opération réalisée par l'utilisateur à l'issue de la réception de ces événements. Il faut juste distinguer et caractériser, en entrée du système analysé, un groupe de dix types d'événement (les dix types de pression possible sur le pavé de dix boutons) qui circulent par un même canal. La matérialisation de l'agent extérieur *User interface* en figure 5.11 est suffisante pour préciser ce canal et le type d'objet chargé du traitement des événements : le thermostat programmable.

Continuons dans cette logique. Ainsi, bien que le capteur de température ne reçoive *a priori* aucun événement et ne génère, également *a priori*, qu'un seul événement : *ambient temperature changed(temperature)*, qu'en est-il si le cahier des charges parle de la gestion des pannes du capteur de température, de sa précision, de son type technique en général ? Il se peut alors que celui-ci intègre le *Class Diagram* comme un type d'objet à part entière. Finalement, quel est le critère, relativement informel, qui permet de retenir ou d'évincer un type d'objet des différents diagrammes UML ? Ce sont des propriétés attribuées (événement reçu par..., événement généré par..., état de...) qui conditionnent le fonctionnement global du système. L'ensemble des états de l'utilisateur ne prête pas d'intérêt. Il en est de même pour le pavé de dix boutons et le capteur de température. En appliquant ce critère, la table des températures cibles n'est pas retenue mais elle apparaît pourtant en figure 5.2. On lui attribue cependant des propriétés pertinentes autres que des états et des géné-

1. Nous avons « volé » ce terme à Cook et Daniels dans [2]. Cet ouvrage peut être consulté avec profit pour son étude de la différence entre situation et système logiciel, et donc distinction entre, encore une fois, analyse et conception.

rations/réceptions d'événements. Ainsi, il y a exactement huit créneaux de réglage des températures cibles scindés entre *weekend* et *weekday*.

Voici donc la cueillette des objets achevée. En conformité avec l'avant-propos de cet ouvrage, douter, c'est poser et reposer les bonnes questions et donc converger vers la qualité, totale espérons-nous.

5.5 CONCEPTION

La frontière entre analyse et conception reste en développement objet la quadrature du cercle. Il est regrettable de ne pas trouver dans le langage UML une distinction claire entre notations d'analyse et notations de conception. Les experts qui liront ces lignes diront que cela relève du processus de développement et non du langage de modélisation, certes. Mais il n'y a pas besoin d'un processus de développement objet hautement codifié pour classer, confiner des notations, dans « le thème conception/implémentation », telles par exemple les classes génériques au sens des *template* C++ (voir chapitre 2).

Le parti pris ici pour traiter ce cas de domotique jusqu'à l'implémentation, est de proposer cette distinction, et donc d'aborder des aspects « processus », même si d'une part on trouve dans la littérature des opinions inverses et si d'autre part, toute application ayant ses spécificités, les règles que nous énonçons peuvent être inadéquates dans un autre contexte. Nous tentons donc d'établir une démarche minimale de développement en vue de « mener » une ingénierie des modèles la plus correcte possible.

L'application Java réalisée est un simulateur. Trois classes Java spécifiques à la simulation, *Season*, *Air_conditionner* et *Furnace* miment respectivement, la saison (de type européen : printemps, été, automne et hiver) établie en fonction de l'horloge système, l'abaissement (de façon aléatoire grâce à la classe *java.util.Random*) de la température ambiante et son accroissement (même méthode que pour l'abaissement). La température ambiante est créée sous forme d'une instance unique de *Temperature*. Cette dernière est donc partagée dans le logiciel par l'unique instance de la classe Java *Temperature sensor*, l'instance unique d'*Air_conditionner* et l'instance unique de *Furnace*. *Temperature sensor* se charge de simuler des variations de température dues à l'environnement atmosphérique alors qu'*Air_conditionner* et *Furnace*, en marche, la diminue, respectivement l'augmente, suite aux commandes d'activation venant des relais (opération *activate* en figure 5.21).

Le choix fait, en toute flexibilité, est que le capteur de température communique la température ambiante toutes les secondes au thermostat programmable. Ce choix confirme la grande souplesse autorisée d'une part par la modélisation en figure 5.11 et d'autre part par le cahier des charges, sans contrainte sur le sujet.

5.5.1 Grandes règles de conception

Les événements reçus par un type d'objet et identifiés en tant que tels dans son *statechart* (en surimpression des transitions ou vus comme des actions internes dans les états), deviennent des opérations publiques de la classe qui implémente le type d'objet et son automate. Par exemple, les événements *run* et *stop* (figure 5.21) du type *Relay* deviennent deux méthodes publiques (voir code ci-avant). Dans un souci de rigueur, le canal de transmission de l'événement doit être explicitement précisé dans les *Class Diagram* ou implicitement via des navigations possibles ou des éléments de modèle accessibles dans l'espace de nommage du type. Par exemple, un événement peut être porteur d'un attribut typé offrant dans l'espace de nommage du *statechart* un point d'entrée d'une navigation.

Les événements provenant de l'extérieur du système, c'est-à-dire émis par aucun des types d'objet apparaissant dans le modèle de la figure 5.2, doivent être regroupés logiquement dans des interfaces idoines (*Temperature sensor client*, *Button pad*, *Season switch user interface* et *Fan switch user interface* dans la figure 5.23) et assignés à des composants logiciels « frontière » qui incarnent les entrées/sorties. Dans la figure 5.23, ce sont les composants logiciels *User interface* et *Temperature sensor*. *Timer* est délibérément occulté par la réutilisation quasi normée (héritage de *Timer client*) du patron de spécification présenté avant. De tels composants opèrent souvent la liaison avec le système d'exploitation qui, par ses possibilités, va contraindre la manière dont l'application va réagir aux phénomènes extérieurs et elle-même influer sur son environnement comme un système sous contrôle.

Dans la figure 5.23, nous utilisons la notation expansée des *Component Diagram* (l'explication des interfaces et de leur contenu par exemple). Les associations sont orientées en cohérence avec le flux de contrôle établi en figure 5.11. Des contraintes opératoires font cependant que des canaux s'ajoutent comme la navigation bidirectionnelle entre *Programmable thermostat* et *User interface* (pour information, le flux de contrôle est monodirectionnel en figure 5.11) pour effectuer les rafraîchissements à l'écran.

La spécification en figure 5.10 oblige pour la grande majorité des autotransitions sur l'état *Control* (bas du statechart) à connaître l'état du commutateur de saison (*season switch* ***in*** *Is cool* ou *season switch* ***in*** *Is heat*). Au lieu d'utiliser l'opérateur *in* natif dans les *Statecharts* de Harel, on peut modéliser le besoin de connaissance de tout ou partie des états d'un autre objet par un envoi d'événement « dans quel état es-tu ? » et d'un envoi d'événement par le receveur, en tant que réponse, « je suis dans l'état … ». C'est laborieux, mais point de vue du puriste, voire de l'intégriste objet, l'encapsulation est respectée : l'état est par définition une notion privée et donc invisible de l'extérieur. Dans le protocole d'échange exposé, le questionné répond donc s'il le veut bien. Nous avons aussi vu précédemment (figure 5.7 et figure 5.8) que des navigations sophistiquées peuvent faire référence à des états d'objets en transgression justement d'une certaine encapsulation.

Nous pensons que si transgression il y a, c'est en implémentation uniquement. En spécification, l'utilisation de l'opérateur *in* ou celui d'OCL, *oclInState*, est naturelle

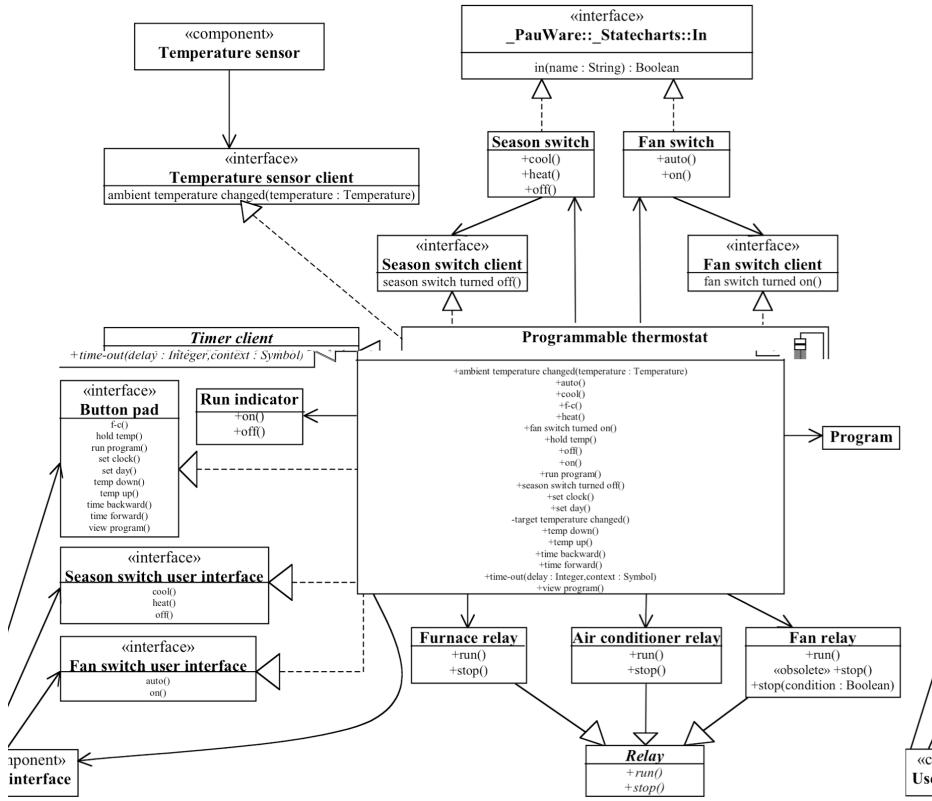


Figure 5.23 – Architecture logicielle du cas de domotique.

et non gênante. Le problème est donc d'implanter correctement ces opérateurs. En haut de la figure 5.23 par exemple, les classes *Season switch* et *Fan switch* implémentent simplement l'interface Java prédéfinie *_PauWare::_Statecharts::In* qui exploite leur automate respectif pour déterminer leur état courant. La classe *Programmable thermostat* accède alors à cet état, au besoin, en l'occurrence dans l'état *Control* (figure 5.10). L'implémentation réelle s'appuie sur la méthode disponible *in_state(name : String) : Boolean* offerte par la classe *_PauWare::_Statecharts::Statechart_monitor* :

```
public class Season_switch implements In {
   …
   protected Statechart _Is_cool;
   protected Statechart _Is_heat;
   protected Statechart _Is_off;
   protected Statechart _Is_on;
   protected Statechart_monitor _Season_switch;
   …
   public boolean in(String name) {
   return _Season_switch.in_state(name);
   }
   …
}
```

5.5.2 Design de l'interface utilisateur

Et l'interface utilisateur ? Il est presque anecdotique que de la construire, c'est-à-dire de la maquetter (figure 5.24) puis de la coder à l'aide de la bibliothèque Java *Swing*.

Figure 5.24 – Apparence de l'interface utilisateur.

Cet avis volontairement provocateur a une fonction d'alarme. En aucun cas, l'application ne doit être pensée en fonction de l'interface utilisateur. C'est le contraire qu'il faut toujours[1] faire : l'interface utilisateur, son design, sa programmation, se plient aux modèles métier, cœur du problème (application) et de la problématique (domaine). À notre sens, la catastrophe actuelle est que la plupart des environnements de développement (*Integrated/Interactive Development Environment*, IDE), par leurs outils (*Interface Builder*), guident, voire forcent, la naissance et l'évolution d'une application avec l'interface utilisateur comme point d'appui. Le résultat est, par expérience, l'antithèse de la réutilisation. Les types d'objet construits voient leurs fonctionnalités et donc leur nature profonde s'adapter à l'interface homme/machine au détriment d'une forme plus transversale, plus détachée de l'application, répondant à des critères métier ou domaine.

5.5.3 Intégration

Nous ébauchons ici l'immersion de notre architecture dans un canevas d'application (*framework*, voir chapitre 1) particulier, en l'occurrence le modèle de composants technologiques *JavaBeans*.

L'état *Something on* du témoin d'activité (figure 5.9) indiquant si le chauffage ou le conditionneur d'air est en marche, est destiné à être connu de l'utilisateur. Dans les faits, une partie de l'interface utilisateur informe donc de l'état de ce témoin : *Something on* ou *Everything off*. Les événements *on* et *off* réceptionnés dans l'automate du type *Run indicator* (figure 5.9) sont devenus en conception les méthodes publiques *on()* et *off()* du même type en figure 5.25. C'est le thermostat programmable qui envoie les événements *on* et *off*. L'interface utilisateur et le témoin d'activité sont en fait deux types qui s'ignorent, ce qui est sain pour conserver un couplage fai-

1. Sauf dans le cas d'un jeu vidéo par exemple, où l'interface utilisateur n'est pas un élément de l'application, c'est l'application.

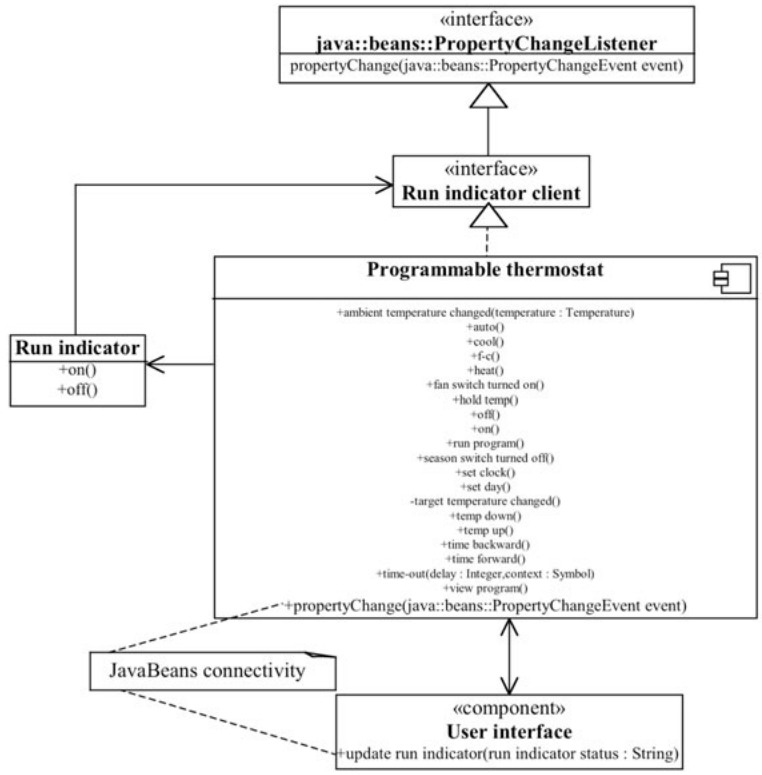


Figure 5.25 – Utilisation de *JavaBeans* pour « remonter » vers l'interface utilisateur, l'état du témoin d'activité.

ble. Il faut donc mettre en place un mécanisme qui « remonte » les changements d'états du témoin d'activité au niveau du thermostat, ce dernier se chargeant ensuite de les propager en direction de l'interface utilisateur, pour rafraîchissement.

L'interface Java *Run indicator client* est ainsi créée en héritant de l'interface *java::beans::PropertyChangeListener* (figure 5.25). En implémentant l'interface *Run indicator client* (relation UML de réalisation en figure 5.25), le thermostat programmable fournit un code de traitement d'une instance du type *java::beans::PropertyChangeEvent*. Le témoin d'activité créé cette instance qui incarne le changement d'état, *i.e.* l'ancien état et le nouveau :

```
public class Run_indicator implements java.io.Serializable {
   // la propriété dont le changement est appelé à être observé, est nommée :
   public static final String Status="Run indicator status";
   // propriété transmise à l'interface utilisateur,
   // une chaîne de caractères :
   private String status;
   // outillage de base associé à JavaBeans :
   private java.beans.PropertyChangeSupport propertySupport;
   // implantation du modèle à gauche de la figure 5.9
   protected Statechart _Everything_off = new Statechart("Everything off");
```

```
    protected Statechart _Something_on = new Statechart("Something on");
    protected Statechart_monitor _Run_indicator;
    // création
    protected void init_structure() throws Statechart_exception {
    }
    protected void init_behavior() throws Statechart_exception {
    _Run_indicator = new Statechart_monitor(_Everything_off.xor(_Something_on),
      ➥ "Run indicator");
    }   _Everything_off.inputState();
    public Run_indicator(Run_indicator_client programmable_thermostat)
      ➥ throws Statechart_exception {
    init_structure();
    init_behavior();
    status = getStatus();
    propertySupport = new java.beans.PropertyChangeSupport(this);
    // le thermostat programmable est ajouté à la liste des écouteurs
    // du changement d'état :
    addPropertyChangeListener(programmable_thermostat);
    }
    // le modèle de composant  JavaBeans est canonique en ce sens
    // qu'il incite à la construction des getters et setters
    // sur toutes les propriétés écoutées getter :
    private String getStatus() {
    return _Run_indicator.current_state();
    }
    // setter :
    private void setStatus(String newValue) {
    String oldValue = status;
    status = newValue;
    // émission du changement d'état aux écouteurs,
    // ici le thermostat programmable :
    propertySupport.firePropertyChange(Status,oldValue,newValue);
    }
    // ajout dynamique d'écouteur, inutile ici donc private
    private void addPropertyChangeListener(java.beans.
      ➥ PropertyChangeListenerlistener) {
    propertySupport.addPropertyChangeListener(listener);
    }
    // suppression dynamique d'écouteur, inutile ici donc private
    private void removePropertyChangeListener(java.beans.
      ➥ PropertyChangeListenerlistener) {
    propertySupport.removePropertyChangeListener(listener);
    }
    // fin du statechart, le traitement des événements reçus :
    public void off() throws Statechart_exception {
    _Run_indicator.fires(_Something_on,_Everything_off);
    _Run_indicator.run_to_completion();
    setStatus(_Run_indicator.current_state());
    }
    public void on() throws Statechart_exception {
    _Run_indicator.fires(_Everything_off,_Something_on);
    _Run_indicator.run_to_completion();
    setStatus(_Run_indicator.current_state());
    }
}
```

Le composant logiciel *Programmable thermostat* se charge alors de propager le changement d'état à l'interface utilisateur de la manière suivante :

```
public void propertyChange(final java.beans.PropertyChangeEvent event) {
   if(event.getPropertyName().equals(Run_indicator.Status))
_user_interface.update_run_indicator_status((String)event.getNewValue());
}
```

5.5.4 Bibliothèque *PauWare.Statecharts*

Le but de la bibliothèque de classes Java *PauWare.Statecharts* est d'instrumenter l'implantation des *Statecharts* de Harel. La configuration de l'automate d'un composant logiciel s'opère via la description de ces états possibles, comme par exemple pour le thermostat programmable :

```
protected Statechart _Run;
protected Statechart _Hold;
…
```

Les états sont ensuite mis en liaison, préférablement dans une fonction *init_behavior()* ayant vocation d'être outrepassée (*overriding*, visibilité *protected*) et appelée dans le constructeur du composant logiciel. Les fonctions de mise en relation sont *xor* (états exclusifs comme *Run* et *Hold* ci-dessous) et *and* (états parallèles comme *Operate* et *Control*, voir les toutes dernières lignes de code ci-dessous). L'emboîtement se met en œuvre via le constructeur de la classe Java *Statechart* : l'instance de *Statechart* passée en argument est un sous-état direct de l'objet *Statechart* construit. Voici donc le code correspondant à la figure 5.10 :

```
_Run = new Statechart("Run");
_Hold = new Statechart("Hold");
_Current_date_and_time_displaying = (new
Statechart("Current_date_and_time_displaying")).entryAction(this,
     ➥ "display_current_date_and_time",null);
_Ambient_temperature_displaying = (new
Statechart("Ambient_temperature_displaying")).entryAction(this,
     ➥ "display_ambient_temperature",null);
_Target_temperature_displaying = (new
Statechart("Target_temperature_displaying")).entryAction(this,
     ➥ "display_target_temperature",null);
_Operate = (((_Run.xor(_Hold)).and(_Current_date_and_time_displaying.xor(
     ➥ _Ambient_temperature_displaying)).and(_Target_temperature_displaying)).na
me("Operate"));
_Set_current_minute = new Statechart("Set_current_minute");
_Set_current_hour = new Statechart("Set_current_hour");
_Set_time = (_Set_current_minute.xor(_Set_current_hour)).name("Set_time");
_Set_current_day = new Statechart("Set_current_day");
_Set_period = new Statechart("Set_period");
_Set_program_time = new Statechart("Set_program_time");
_Set_program_target_temperature = new
Statechart("Set_program_target_temperature");
_Set_program =
(_Set_period.and(_Set_program_time).and(_Set_program_target_temperature)).
     ➥ name("Set_program");
```

```
_Current_date_and_time_refreshing = (new Statechart("Current_date_and_time_
    ➥ refreshing")).entryAction(this,"display_current_date_and_time",null);
_Program_target_temperature_refreshing = (new
Statechart("Program_target_temperature_
    ➥ refreshing")).entryAction(this,"display_program_target_temperature",null);
_Period_and_program_time_refreshing = (new Statechart("Period_and_program_time_
    ➥ refreshing")).entryAction(this,"display_period_and_program_time",null);
_Program_refreshing = (_Program_target_temperature_refreshing.
    ➥ and(_Period_and_program_time_refreshing)).name("Program_refreshing");
_Setup = ((_Set_time.xor(_Set_current_day).xor(_Set_program).
    ➥ and(_Current_date_and_time_refreshing.xor(_Program_refreshing))).
    ➥ name("Setup"));
_Control = new Statechart("Control");
_Programmable_thermostat = new Statechart_monitor((_Operate.xor(_Setup)).
    ➥ and(_Control),"Programmable_thermostat");
_Hold.inputState();
_Current_date_and_time_displaying.inputState();
```

Des fonctions utilitaires diverses et variées permettent d'ajouter les actions lancées en entrée (*entry/*), en sortie (*exit/*)… La classe Java *Statechart_monitor* qui hérite de *Statechart* incarne l'automate tout entier. On en crée donc une instance (code ci-avant), nommée par convention du nom du composant logiciel préfixé d'un « _ » :

```
protected Statechart_monitor _Programmable_thermostat;
```

En mémoire, le *statechart* est organisé comme un arbre binaire (figure 5.26) avec pour chaque père, deux fils et leur relation : *xor* ou *and*.

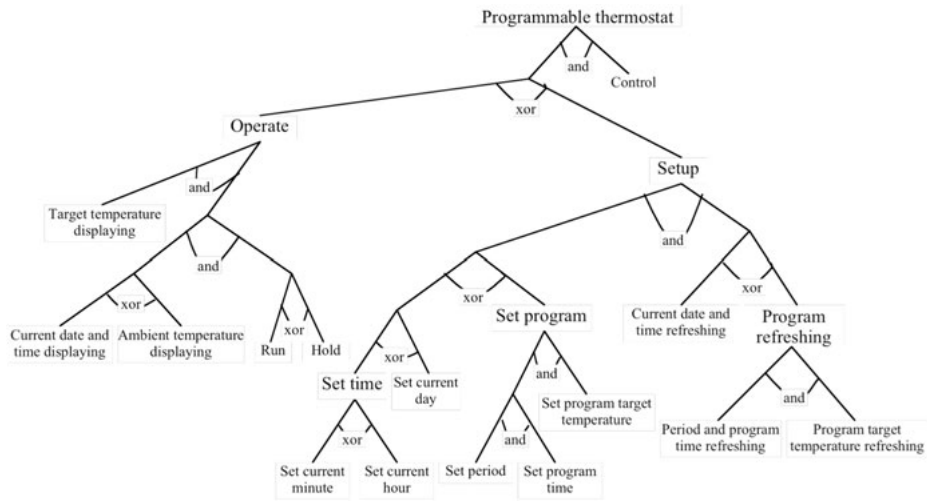


Figure 5.26 – Arbre binaire représentant en mémoire un *statechart*.

Notons pour terminer qu'à ce jour, la bibliothèque a été testée pour construire des EJB avec des facilités spécifiques de gestion de la réentrance. Comme les EJB, les composants logiciels construits avec *PauWare.Statecharts* ne sont pas naturellement

réentrants. Il faut utiliser la constante *Statechart_monitor.Reentrance* lors d'un franchissement de transition pour autoriser la réentrance. Voici l'exemple du traitement de l'événement *temp down* :

```
public void temp_down() throws Statechart_exception {
   boolean guard = _target_temperature.greaterThan(_Min);
   _Programmable_thermostat.fires(_Target_temperature_displaying,_
     ➥ Target_temperature_displaying,guard,this,"target_temperature_changed",
     ➥ null,Statechart_monitor.Reentrance);
   …
```

Dans le code qui précède, il y a envoi à lui-même (voir aussi figure 5.11, tout à droite, au milieu) de l'événement *target temperature changed*. La constante *Statechart_monitor.Reentrance* assure que le traitement de cet événement aura lieu à la fin complète de celui de *temp down* pour ne pas déréguler l'automate : ce n'est jamais que le mode *run-to-completion* qui a toujours été préconisé dans UML. On reste donc bien dans la sémantique originelle d'exécutabilité.

Les EJB ne sont pas naturellement réentrants (on peut les déclarer réentrants dans leur descripteur de déploiement mais l'expérience montre que cela est à éviter…). En cas de traitement inachevé d'occurrence d'événement au moment de la réception d'une autre occurrence, il faut utiliser les *Message-Driven Beans* et *Java Message Service* (JMS) pour s'assurer que le récepteur ne traite les messages entrants que lorsqu'il est « libre » ou bien notre technologie à base de *Stateful Session Bean* uniquement (paramètre *Statechart_monitor.Reentrance*) qui rend les EJB réentrants. Rappelons brièvement que les contraintes de non-réentrance imposées par des modèles de composants logiciels technologiques comme EJB, viennent de la gestion des transactions, qui ne peuvent être emboîtées en EJB.

5.6 CONCLUSION

Ce chapitre tente de démontrer qu'il faut privilégier les types de diagramme « essentiels » (voir classification de Mellor au chapitre 2) au détriment d'un usage inconsidéré et exhaustif de tous les types de diagrammes possibles d'UML 2.x. Avec les *Class Diagram*, plus modérément les *Communication Diagram* et les *Component Diagram*, et intensivement les *State Machine Diagram*, nous obtenons une spécification plutôt solide de l'étude de cas en domotique sans néanmoins aller vers une spécification formelle. Les *Timing Diagram* par exemple, alors que l'application utilise abondamment des services de cadencement, ne sont pas mis en œuvre. Sous un autre angle, les *Activity Diagram* d'UML 2.x, malgré leur primauté affichée, s'avèrent inutiles puisque nous nous en passons totalement. De même en ingénierie des besoins, il n'y pas de recours obligé aux *Use Case Diagram*.

Nous pensons qu'un effort de concentration sur le sens précis et exact que l'on donne aux modèles construits est largement préférable à la multiplication de modèles dans tous les formalismes permis à ce jour par UML. Nous soulevons de nom-

breux problèmes épineux dont la résolution est délicate : masquage de réactivité ou encore héritage d'automate(s). À la différence de l'étude de cas traitée dans le livre de Rumbaugh *et al.* [7], nous allons jusqu'au code en mettant l'accent sur une traçabilité totale, dans l'esprit du *Model-Driven Engineering* (MDE). Une question reste néanmoins posée : UML est-il destiné à faire des spécifications aussi pointilleuses et détaillées que les nôtres ? Rien n'est moins sûr. Doit-il à l'opposé être cantonné à une phase d'ébauche ? Pointe alors le syndrome touchant les développeurs de terrain (cf. chapitre 1) : « Y'a des boîtes et y'a des flèches mais on ne voit pas bien ce qu'il y a dans les programmes. » À méditer…

5.7 BIBLIOGRAPHIE

1. Booch, G. : *Object-Oriented Analysis and Design with Applications*, Second Edition, Benjamin/Cummings (1994)
2. Cook, S., and Daniels, J. : *Designing Object Systems – Object-Oriented Modelling with Syntropy*, Prentice Hall (1994)
3. Fowler, M. : *Analysis Patterns – Reusable Object Models*, Addison-Wesley (1997)
4. Harel, D.: Statecharts : "A Visual Formalism for Complex Systems", *Science of Computer Programming*, 8, (1987) 231-274
5. Harel, D., and Gery, E. : "Executable Object Modeling with Statecharts", *IEEE Computer*, 30(7), (1997) 31-42
6. Object Management Group : *UML 2.0 Superstructure Specification* (2003)
7. Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., and Lorensen, W. : *Object-Oriented Modeling and Design*, Prentice Hall (1991)
8. Shlaer, S., and Mellor, S. : *Object Lifecycles – Modeling the World in States*, Prentice Hall (1992)
9. Wirfs-Brock, R., Wilkerson, B., and Wiener, L. : *Designing Object-Oriented Software*, Prentice Hall (1990)

5.8 WEBOGRAPHIE

AGL Rhapsody : www.ilogix.com
Série de conférences UML : www.umlconference.org
Sources de l'étude de cas : www.PauWare.com

6

Système bancaire

6.1 INTRODUCTION

Eh oui, revoilà le cas du distributeur automatique bancaire ! Tout ce que vous auriez toujours voulu savoir sur ce célèbre cas d'école est dans ce chapitre, mais avec un plus : l'implémentation complète en Java mode client/serveur, avec en particulier l'usage de JDBC.

Cette étude de cas, telle qu'elle est traitée ici, est une libre adaptation de l'étude figurant dans le livre sur OMT [5] ou de nombreux autres ouvrages, puisqu'il s'agit, du cas standard pour tester un langage de spécification. Ainsi, les auteurs de ces ouvrages sur les méthodes de développement logiciel, la modélisation et la spécification ont usé et abusé de ce cas d'étude. Malheureusement, à notre connaissance, aucun n'est véritablement allé jusqu'au terme de l'étude en fournissant la totalité des modèles et du code. En effet, cette étude de cas réputée simpliste, voire naïve, devient subitement plutôt compliquée si l'on veut bien prendre la peine de l'étudier dans sa globalité, et surtout de l'implémenter : les différents sources ainsi que leurs mises à jour et éventuels *patchs* (« rustines » en français) sont téléchargeables sur Internet (voir section « Webographie » en fin de chapitre).

La réalisation de l'étude de cas est basée sur la bibliothèque Java *PauWare.Statecharts*, réutilisée et expliquée au chapitre 5. Encore une fois, les *State Machine Diagram* priment mais nous montrons aussi la mise en œuvre des *Activity Diagram* en soulevant des critiques sur la qualité des modèles obtenus, leur précision en particulier. Dans ce chapitre, un exercice de maintenance de l'application est également disponible. C'est le meilleur moyen de l'aborder puis de la comprendre car les modèles sont nombreux. La mise à jour de ces derniers puis conséquemment celle du code, dans le cadre de cette maintenance, a une vocation pédagogique. Une lecture passive pourrait être inefficace au vu de tous les détails de modélisation à intégrer.

6.2 PRÉSENTATION GÉNÉRALE

Cette section a pour but de donner un descriptif d'ensemble de l'infrastructure bancaire.

6.2.1 Éléments constitutifs du système

Un ordinateur central et un ensemble de distributeurs automatiques bancaires (ATM, *Automated Teller Machine*) sont la propriété d'un consortium de banques. Chaque banque dispose de son propre parc d'ordinateurs pour gérer les comptes de ses clients. Cependant, il existe pour chaque banque un ordinateur appelé « ordinateur de banque » qui communique avec l'ordinateur central pour traiter les transactions effectuées via les ATM à l'aide de cartes bancaires. Cette infrastructure globale est exposée en figure 6.1.

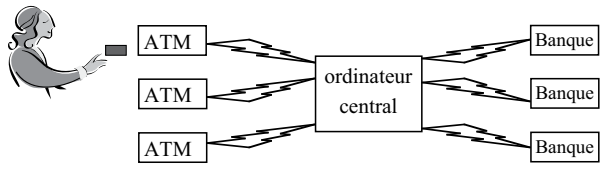


Figure 6.1 – Synoptique d'organisation du système bancaire.

Le consortium incarné par l'ordinateur central, gère le réseau permettant de propager le contrôle d'autorisation de l'ATM à la banque appropriée, ainsi que le traitement d'opérations bancaires toujours de l'ATM vers la banque appropriée. Le consortium n'effectue à proprement dit aucun contrôle, traitement ou plus généralement calcul. Il gère la concurrence de réception des requêtes provenant des ATM (contrôle d'autorisation, traitement d'opération bancaire, annulation d'autorisation, annulation d'opération bancaire) ainsi que des réponses provenant des banques (résultat de contrôle d'autorisation, résultat de traitement d'opération bancaire). Il gère également la concurrence d'émission des réponses destinées aux ATM (contrôle d'autorisation « OK », contrôle d'autorisation « non OK », traitement d'opération bancaire « OK », traitement d'opération bancaire « non OK ») ainsi que des requêtes destinées aux banques (autorisation à contrôler, opération bancaire à traiter, opération bancaire à annuler).

Un ATM est constitué des parties suivantes :

- une interface utilisateur (figure 6.2) ;
- un lecteur de carte (*Card reader*) qui est chargé de l'avalement de la carte, sa lecture et éventuellement son stockage (sa consignation) dans le cas d'un oubli ou d'une tentative de fraude et finalement de son éjection ;

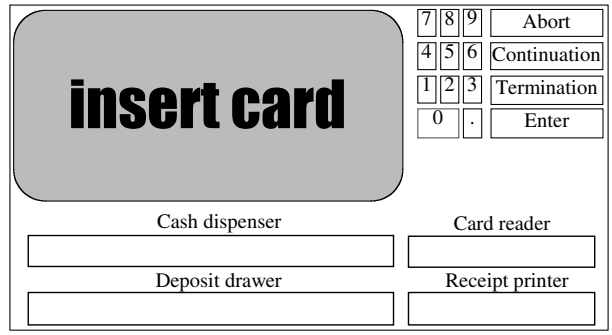


Figure 6.2 – Interface utilisateur et dispositifs d'un ATM.

- un récupérateur d'argent (*Deposit drawer*) qui est chargé, lors d'une opération bancaire de dépôt, de la récupération de l'enveloppe contenant l'argent et/ou le(s) chèque(s) ;
- un distributeur d'argent (*Cash dispenser*) qui est chargé, lors d'une opération bancaire de retrait, de la distribution de l'argent. Il dispose à tout moment d'informations sur le montant d'argent initialement déposé ainsi que sur le montant d'argent déjà délivré ;
- un système d'impression de ticket (*Receipt printer*) qui est chargé de l'impression et de la distribution du ticket à l'issue de la transaction.

6.2.2 Fonctionnement général de l'ATM

Un utilisateur effectue une transaction sur un ATM à l'aide d'une carte. Cet utilisateur est nécessairement reconnu et enregistré en tant que client d'au moins une des banques du consortium.

La transaction débute lorsque le *Card reader* a lu la carte et communiqué à l'ATM les informations stockées sur la carte. La transaction se termine lorsque la carte a été retirée par l'utilisateur ou a été stockée (consignée) par le *Card reader*. Toute transaction est datée (seconde, minute, heure, jour, mois, année) à l'instant où elle débute. Toute transaction est archivée à l'instant où elle se termine.

L'insertion de la carte donne lieu à son avalement et à sa lecture. Si la carte est illisible (les informations stockées sur la carte ne peuvent pas être lues), le *Card reader* informe l'ATM de la non-lisibilité de la carte : c'est la notion d'autorisation. L'éjection ou le stockage (consignation) de la carte résulte toujours d'une demande de l'ATM. De manière générale, à l'issue de l'éjection, si l'utilisateur prend la carte, le *Card reader* informe l'ATM de la prise de la carte. Au-delà de trente secondes, si l'utilisateur n'a pas pris la carte, le *Card reader* avale et stocke (consigne) la carte (cas d'oubli) et informe l'ATM du stockage (consignation) de la carte, sinon le *Card reader* informe l'ATM de la prise de la carte.

6.2.3 Fonctionnement général de l'ordinateur central

Un utilisateur effectuant une transaction sur un ATM à l'aide d'une carte fait l'objet d'un contrôle d'autorisation. Outre le fait que le mot de passe stocké sur la carte doit être celui entré au clavier, l'autorisation est établie par la banque où l'utilisateur est reconnu et enregistré comme client. Les informations stockées sur la carte permettent à l'ATM de définir une requête de contrôle d'autorisation en vue de l'envoyer au consortium pour routage à la banque appropriée (autorisation à contrôler).

Le contrôle d'autorisation effectué par la banque où l'utilisateur est reconnu et enregistré comme client permet de vérifier que l'identité de la carte, l'identité du client ou encore l'attribution d'une certaine identité de carte à une certaine identité de client, est bien conforme aux informations dont dispose la banque.

L'autorisation peut être refusée par la banque pour des motifs divers : le client n'a plus le droit d'utiliser la carte, le parc d'ordinateurs de la banque n'est pas en mesure d'assurer le contrôle d'autorisation, et donc la transaction, etc. La banque retourne au consortium le résultat de contrôle d'autorisation avec le portefeuille (*portfolio* en anglais) du client (ensemble des comptes actifs) ainsi que l'erreur de vérification d'autorisation : « pas d'erreur » ou « texte d'erreur » pour routage à l'ATM concerné (contrôle d'autorisation « OK » avec portefeuille du client ou contrôle d'autorisation « non OK » avec erreur de vérification d'autorisation).

Si le contrôle d'autorisation est « OK », l'utilisateur peut effectuer un certain nombre d'opérations bancaires qui pour chacune donne lieu à une requête de demande de traitement de cette dernière. Les informations entrées au clavier (montant de l'opération par exemple, dans le cas où celle-ci n'est pas simplement une interrogation) permettent à l'ATM de définir la requête « traitement d'opération bancaire » en vue de l'envoyer au consortium pour routage à la banque appropriée (opération bancaire à traiter).

L'opération peut être refusée par la banque pour des motifs divers : dans le cas d'un retrait par exemple, le client n'a plus suffisamment d'argent sur le compte où il effectue le retrait ; toujours dans le cas d'un retrait, la somme des retraits effectués par le client est supérieure à la limite hebdomadaire de retrait de la carte, etc.

Remarque

Lors d'une opération bancaire de retrait, l'ATM est chargé d'empêcher que le montant du retrait soit supérieur à la limite hebdomadaire de retrait de la carte. Cependant, la banque est chargée d'empêcher que la somme des retraits effectués par le client soit supérieure à la limite hebdomadaire de retrait de la carte.

La banque retourne au consortium le résultat du traitement d'opération bancaire avec le compte du client à afficher ainsi que l'erreur de vérification d'opération bancaire : « pas d'erreur » ou « texte d'erreur » pour routage à l'ATM concerné (trai-

tement d'opération bancaire « OK » avec compte à afficher ou traitement d'opération bancaire « non OK » avec erreur de vérification d'opération bancaire).

6.2.4 Panne et réparation de l'ATM

L'ATM est considéré être hors service si le *Card reader*, le *Deposit drawer*, le *Cash dispenser* ou encore le *Receipt printer* tombe en panne. L'ATM passe alors lui-même dans l'état hors service.

Dans l'état hors service, l'ATM affiche un message approprié. La réparation de l'ATM consiste en la réparation d'un ou plusieurs des dispositifs précités. À l'issue de la réparation, l'ATM affiche alors la demande d'insertion de la carte.

Si le *Cash dispenser* ou le *Deposit drawer* tombe en panne au moment, respectivement, de la distribution de l'argent ou de la récupération de l'enveloppe, l'ATM doit envoyer une requête d'annulation d'opération au consortium pour routage à la banque appropriée (opération bancaire à annuler). En effet, si la banque a préalablement traité l'opération bancaire en rendant positif le résultat de traitement d'opération, il y a lieu de l'informer de la non-réalisation effective au niveau de l'ATM.

6.2.5 Panne de l'ordinateur central

La panne de l'ordinateur central peut occasionner la perturbation de la transaction en cours sur l'ATM. Cependant, l'ATM dispose d'une autonomie lui permettant, au-delà de soixante secondes, d'abandonner (ignorer) la requête de contrôle d'autorisation ou de traitement d'opération bancaire qu'il vient d'envoyer au consortium.

Ainsi, la panne de l'ordinateur central matérialisée au niveau de l'ATM par l'absence de réponse aux deux types de requête précités n'empêche donc pas l'ATM de terminer dans des conditions correctes la transaction en cours. L'inconvénient du mécanisme est que, dans des conditions normales (absence de panne), la réponse à la requête doit arriver avant que soixante secondes se soient écoulées. De la réception de la requête, le routage à la banque, la réception de la réponse de la banque, au routage de la réponse de la banque à l'ATM, l'ordinateur central dispose de soixante secondes. Si plus de soixante secondes s'écoulent, il ne doit donc pas envoyer la réponse à la requête car l'ATM considère que l'ordinateur central a rencontré des problèmes et n'a pas traité la requête (l'ATM abandonne dans ce cas la requête).

Remarque

Le mécanisme précité n'est réaliste que si le temps de communication entre l'ATM et l'ordinateur central ainsi qu'entre l'ordinateur central et l'ATM est négligeable par rapport au temps imparti au traitement de la requête par l'ordinateur central. La pratique montre cependant que le choix de soixante secondes est un choix pertinent.

6.3 PRÉSENTATION DÉTAILLÉE

Cette section a pour but de donner un descriptif logique de l'infrastructure bancaire notamment via la caractérisation de son système d'information.

6.3.1 Éléments informatifs du système

ATM : c'est un dispositif qui permet aux clients d'effectuer des transactions en utilisant une carte bancaire. L'ATM interagit avec le client pour recueillir les informations de la transaction, envoyer ces informations à l'ordinateur central pour contrôle et/ou traitement par l'ordinateur de banque. Un ATM est reconnu dans le consortium par son code station.

Banque : c'est une institution financière identifiée au sein du consortium par son code banque. Une banque attribue ou non à un de ses clients, une ou plusieurs cartes, et permet ou non l'accès par un de ses clients, à un ou plusieurs de ses comptes via le réseau du consortium. Un des clients d'une banque possédant plusieurs cartes dispose en fait de plusieurs copies de la carte (*cf. Carte* juste après) ayant même numéro de carte, même mot de passe et même limite hebdomadaire de retrait.

Carte : c'est une carte de crédit d'un client d'une banque qui permet au client de la banque l'utilisation d'un ATM. Chaque carte possède un numéro de carte établi en fonction des standards internationaux en matière de carte bancaire : le numéro de carte identifie la carte au sein du consortium. Une carte n'accède pas nécessairement à tous les comptes d'un client d'une banque. Chaque carte est la propriété d'un unique client d'une banque, mais plusieurs copies de la carte (supports plastiques) peuvent coexister. Un support plastique est identifié par un numéro de série. Tous les supports plastiques (copies) d'une carte ont le même numéro de carte, le même mot de passe et la même limite hebdomadaire de retrait.

Client : c'est le possesseur d'un ou plusieurs comptes dans une banque à qui on a attribué une carte. Les comptes que possède un client, et dont l'accès par le client via le réseau du consortium est autorisé, constituent le portefeuille du client. Le portefeuille d'un client n'est jamais vide. En effet, un compte dit « par défaut » utilisé pour supporter toutes les opérations bancaires que ce soient des retraits, des interrogations, des dépôts ou des transferts, constitue la partie minimale d'un portefeuille. Un client a un nom et une adresse et consiste en une ou plusieurs personnes et/ou établissements ; la correspondance entre personnes et/ou établissements constituant le même client n'a cependant ici pas besoin d'être prise en compte. La même personne et/ou établissement possédant un ou plusieurs comptes dans une banque différente et à qui on a attribué une carte, est considéré(e) à chaque fois comme un client différent. Chaque banque affecte à un de ses clients un numéro d'identification (PIN, *Personal Identification Number*) qui est unique au sein de la banque mais qui ne l'est pas nécessairement au sein du consortium. Le numéro d'identification client a la particularité de déterminer le type du client.

Compte : un compte est propre à une banque et fait l'objet d'opérations via les ATM. Les comptes possèdent un solde, peuvent être de différents types et un client peut posséder plus d'un compte. Chaque banque affecte à un compte un numéro d'identification compte qui est unique au sein de la banque mais qui ne l'est pas nécessairement au sein du consortium. Ce numéro d'identification a la particularité de déterminer le type du compte.

Consortium : c'est un groupement de banques qui financent et mettent en œuvre le réseau. Le réseau ne supporte des transactions que pour les banques du consortium.

Opération bancaire : une opération bancaire est valide dès qu'un ATM a procédé à son traitement, c'est-à-dire reçu une réponse positive du consortium suite à une requête « traitement d'opération bancaire » ainsi que, dans le cas d'un retrait, distribué l'argent (sans panne) à l'aide du *Cash dispenser*, et dans le cas d'un dépôt, récupéré l'enveloppe (sans panne) à l'aide du *Deposit drawer*. Une opération bancaire n'existe que dans le cadre d'une transaction. Cependant, seules les opérations bancaires valides sont archivées lorsque la transaction est elle-même archivée. Une opération bancaire est soit un retrait, soit un dépôt, soit un transfert, soit une interrogation. Une opération bancaire a un montant (sauf une interrogation) et s'applique sur un compte, sauf si c'est un transfert, elle s'applique alors sur deux comptes.

Transaction : une transaction débute dès qu'un ATM a pu lire la carte en déterminant le numéro de série de la carte ainsi que les autres informations stockées sur la carte dont en particulier le numéro de carte et le mot de passe. Une transaction est composée de zéro, une ou plusieurs opérations bancaires. Une transaction ne donne lieu à une impression et à une distribution d'un ticket que si elle est composée d'au moins une opération bancaire et si l'utilisateur prend la carte avant trente secondes — si le *Card reader* stocke (consigne) la carte, il n'y a pas d'impression et de distribution d'un ticket.

6.3.2 Scénario ATM normal

L'ATM affiche la demande d'insertion de la carte (voir message d'accueil *insert card* en figure 6.2). L'utilisateur insère la carte.

Le *Card reader* lit la carte en déterminant le numéro de série qui identifie un exemplaire parmi les supports plastiques ayant même numéro de carte et en déterminant les autres informations stockées sur la carte, en particulier le mot de passe.

Dans l'hypothèse où la carte est lisible, l'ATM affiche la demande du mot de passe et permet la saisie du mot de passe. L'utilisateur entre par exemple « 1234 ». L'utilisateur a droit à deux tentatives pour entrer le mot de passe et quinze secondes à chaque tentative. L'ATM teste le mot de passe en déterminant si le mot de passe entré est valide ou invalide.

Dans l'hypothèse où le mot de passe entré est valide, l'ATM traite la transaction en envoyant une requête au consortium pour contrôler l'autorisation. Le consortium

répond « OK » ou « non OK » à l'ATM en attente d'une réponse à la requête. Le consortium retourne avec la réponse, soit le portefeuille (« OK »), soit le texte concernant l'erreur de contrôle d'autorisation qui s'est produite (« non OK »).

Dans l'hypothèse où le consortium répond « OK », l'ATM demande à l'utilisateur de choisir le type d'opération bancaire (retrait, dépôt, transfert ou interrogation). L'utilisateur choisit par exemple « retrait ».

L'ATM affiche la demande du montant et permet la saisie du montant. L'utilisateur entre par exemple « 200 ». L'utilisateur a droit à deux essais pour entrer le montant et soixante secondes à chaque essai. Lorsque le type d'opération bancaire est « retrait », l'ATM contrôle le montant en :

- vérifiant que le montant entré est inférieur ou égal à la limite hebdomadaire de retrait stockée sur la carte ;
- vérifiant au niveau du *Cash dispenser* que le montant entré est inférieur ou égal à ce qui a été initialement déposé moins ce qui a été déjà délivré.

Dans l'hypothèse où le montant entré est valide, l'ATM traite le retrait en demandant à l'utilisateur de choisir le compte. L'utilisateur choisit le compte dans le portefeuille. L'ATM envoie une requête au consortium pour traiter le retrait. Le consortium répond « OK » ou « non OK » à l'ATM en attente d'une réponse à la requête. Le consortium retourne avec la réponse, soit le compte sur lequel le retrait s'est effectué (« OK »), soit le texte relatif à l'erreur de traitement d'opération bancaire qui s'est produite (« non OK »).

Dans l'hypothèse où le consortium répond « OK », l'ATM sollicite le *Cash dispenser* pour distribuer l'argent, affiche durant quinze secondes l'état du compte sur lequel s'est effectué le retrait et demande à l'utilisateur de se saisir de l'argent.

L'ATM demande à l'utilisateur de choisir *continuation* ou *termination* (voir figure 6.2 en haut à droite). L'utilisateur choisit par exemple *continuation*.

L'ATM demande à l'utilisateur de choisir le type d'opération bancaire (retrait, dépôt, transfert ou interrogation). L'utilisateur choisit par exemple *interrogation*.

L'ATM traite l'interrogation en demandant à l'utilisateur de choisir le compte. L'utilisateur choisit le compte dans le portefeuille. L'ATM envoie une requête au consortium pour traiter l'interrogation. Le consortium répond « OK » ou « non OK » à l'ATM en attente d'une réponse à la requête. Le consortium retourne avec la réponse, soit le compte sur lequel l'interrogation s'est effectuée (« OK »), soit le texte d'erreur (« non OK »).

L'ATM demande à l'utilisateur de choisir *continuation* ou *termination*. L'utilisateur choisit par exemple *continuation*.Dans l'hypothèse où le consortium répond « OK », l'ATM affiche durant quinze secondes l'état du compte sur lequel s'est effectuée l'interrogation.

L'ATM demande à l'utilisateur de choisir le type d'opération bancaire (retrait, dépôt, transfert ou interrogation). L'utilisateur choisit par exemple *transfert*.

L'ATM affiche la demande du montant et permet la saisie du montant. L'utilisateur entre par exemple « 1000 ». L'utilisateur a droit à deux essais pour entrer le montant et soixante secondes à chaque essai.

L'ATM traite le transfert en demandant à l'utilisateur de choisir le premier compte (par convention, sur ce compte s'effectue le crédit). L'utilisateur choisit le premier compte dans le portefeuille. L'ATM demande à l'utilisateur de choisir le second compte (par convention, sur ce compte s'effectue le débit). L'utilisateur choisit le second compte dans le portefeuille.

Dans l'hypothèse où le premier compte et le second compte choisis à l'écran ne sont pas les mêmes, l'ATM envoie une requête au consortium pour traiter le transfert. Le consortium répond « OK » ou « non OK » à l'ATM alors en attente d'une réponse à la requête. Le consortium retourne avec la réponse, soit le compte (le second) sur lequel le transfert s'est effectué (« OK »), soit le texte d'erreur (« non OK »).

Dans l'hypothèse où le consortium répond « OK », l'ATM affiche durant quinze secondes l'état du compte (le second) sur lequel s'est effectué le transfert.

L'ATM demande à l'utilisateur de choisir *continuation* ou *termination*. L'utilisateur choisit par exemple *termination*. L'ATM envoie une requête d'annulation d'autorisation au consortium.

L'ATM sollicite le *Card reader* pour éjecter la carte et affiche la demande de prise de la carte puis sollicite le *Receipt printer* pour imprimer et distribuer le ticket ; l'utilisateur prend la carte et prend le ticket.

L'ATM affiche la demande d'insertion de la carte.

6.3.3 Scénario ATM anormal

Le comportement dit normal de l'ATM décrit ci-avant peut être perturbé de diverses manières. Il est illusoire et même impossible d'exprimer littéralement toutes les exceptions au processus normal à cause de l'explosion combinatoire qui s'en suit. Cette section s'attache donc à classifier les grands dysfonctionnements qu'il est fondamental de prendre en charge dans la spécification puis l'implémentation. Cela ne présume pas cependant d'autres dysfonctionnements plus subtils ou plus techniques car découverts *a posteriori*, qui doivent être maîtrisés dans le système logiciel final pour en assurer sa sûreté de fonctionnement.

Mot de passe

Lors d'une tentative d'entrée du mot de passe, s'il s'écoule plus de quinze secondes entre le moment où l'ATM affiche la demande du mot de passe et le moment où l'utilisateur entre le mot de passe, l'ATM sollicite le *Card reader* pour éjecter la carte et affiche la demande de prise de la carte.

Lors de la première et de la seconde tentatives d'entrée du mot de passe, si le mot de passe entré est invalide, l'ATM affiche durant quinze secondes que le mot de passe

est invalide. Si c'est la première tentative, l'utilisateur recommence la saisie du mot de passe. Si c'est la seconde tentative (cas de tentative de fraude), l'ATM sollicite le *Card reader* pour stocker (consigner) la carte.

Contrôle d'autorisation

Lors du contrôle d'autorisation, s'il s'écoule plus de soixante secondes entre le moment où l'ATM envoie la requête au consortium et le moment où le consortium répond, l'ATM sollicite le *Card reader* pour éjecter la carte et affiche la demande de prise de la carte.

Lors du contrôle d'autorisation, si le consortium répond « non OK », l'ATM affiche durant quinze secondes l'erreur de contrôle d'autorisation retournée par le consortium puis sollicite le *Card reader* pour éjecter la carte et affiche la demande de prise de la carte.

Traitement d'opération bancaire

Lors du traitement d'opération bancaire, s'il s'écoule plus de soixante secondes entre le moment où l'ATM envoie la requête au consortium et le moment où le consortium répond, l'ATM sollicite le *Card reader* pour éjecter la carte et affiche la demande de prise de la carte.

Lors d'un retrait, d'un dépôt ou d'un transfert, s'il s'écoule plus de soixante secondes entre le moment où l'ATM affiche la demande du montant et le moment où l'utilisateur entre le montant, l'ATM envoie une requête au consortium pour annuler l'autorisation en cours puis sollicite le *Card reader* pour éjecter la carte et affiche la demande de prise de la carte.

Lors d'un retrait, au premier essai, si le montant entré est supérieur à la limite hebdomadaire de retrait ou si, au niveau du *Cash dispenser*, le montant entré est supérieur à ce qui a été initialement déposé moins ce qui a été déjà délivré, l'ATM affiche durant quinze secondes que le montant est invalide et l'utilisateur recommence la saisie du montant. Au second essai, dans les mêmes conditions, l'ATM affiche durant quinze secondes l'erreur de traitement d'opération bancaire puis demande à l'utilisateur de choisir *continuation* ou *termination*.

Lors du traitement d'opération bancaire, si le consortium répond « non OK », l'ATM affiche durant quinze secondes l'erreur de traitement d'opération bancaire retournée par le consortium puis demande à l'utilisateur de choisir *continuation* ou *termination*.

Touche abort

Les touches *continuation* et *termination* ne peuvent être utilisées que lorsque l'ATM demande à l'utilisateur de s'en servir. À tout autre moment, l'ATM ne réagit pas si l'on presse sur l'une de ces deux touches. En revanche, la touche *abort* peut être utilisée à n'importe quel moment excepté du début de l'éjection de la carte à la fin de l'insertion de la carte. Durant cette période, l'ATM ne réagit pas si l'on presse sur cette touche. Il est à noter que le consortium peut être en train de contrôler une

autorisation pour l'ATM ou encore traiter une opération bancaire pour l'ATM alors que l'utilisateur presse sur la touche *abort*. Deux cas de figure sont à prendre en compte. Dans le premier cas, l'utilisateur peut presser sur la touche *abort* alors que la requête de contrôle d'autorisation n'a pas encore été envoyée par l'ATM au consortium (au moment du test du mot de passe par exemple). Dans ce cas, il n'y a pas lieu d'envoyer une requête d'annulation d'autorisation. Dans le second cas, l'autorisation est en cours. Dans ce cas, il faut envoyer une requête d'annulation d'autorisation (*i.e.* l'utilisateur interrompt la transaction).

Sécurité du système

L'ordinateur central mémorise à tout instant les autorisations à contrôler dans une liste des autorisations non encore contrôlées. Il ne doit pas y avoir dans cette liste deux autorisations ou plus en cours sur le même ATM ou encore deux autorisations ou plus ouvertes avec le même numéro de série de carte plastique bien que cela soit possible avec le même numéro de carte « logique ». Lorsqu'un contrôle d'autorisation effectué par une banque est positif, l'autorisation concernée quitte la liste des autorisations non encore contrôlées pour une liste des autorisations déjà contrôlées satisfaisant aussi les contraintes de sécurité précitées. Les deux listes concernant les autorisations font l'objet de verrouillage du fait de leur accès concurrent par les processus d'émission et de réception du consortium.

Dans le même ordre d'idées, l'ordinateur central mémorise à tout instant les opérations bancaires à traiter dans une liste des opérations bancaires non encore traitées. Une opération bancaire de cette liste doit être associée à une autorisation figurant dans la liste des autorisations déjà contrôlées. Lorsqu'un traitement d'opération bancaire effectué par une banque est positif, l'opération bancaire concernée quitte la liste des opérations bancaires non encore traitées lors du routage à l'ATM concerné.

L'ordinateur central mémorise aussi à tout instant les opérations bancaires annulées dans une liste des opérations bancaires annulées. Une opération bancaire de cette liste doit aussi être associée à une autorisation figurant dans la liste des autorisations déjà contrôlées. Une opération bancaire de cette liste la quitte lors du routage à la banque concernée. Les deux listes concernant les opérations bancaires font aussi l'objet de verrouillage du fait de leur accès concurrent par les processus d'émission et de réception du consortium.

6.4 INGÉNIERIE DES BESOINS

Au regard de la technique souvent préconisée des *Use Case*, le système bancaire pose le problème d'une multitude de cas d'utilisation si l'on veut bien considérer tous les cas d'erreur possibles : l'utilisateur oublie sa carte sur l'ATM, la carte est défectueuse, etc. Outre le fait d'identifier chaque *Use Case* avec éventuellement une description textuelle préformatée, il est aussi souhaitable d'en donner le contenu à l'aide de

Sequence Diagram. Comme nous l'avons déjà mentionné, la combinatoire inhérente aux scénarios en général ne nous permet pas de représenter correctement et de façon exhaustive le système décrit dans le cahier des charges, à moins d'une approche plus cadrée telle que celle exposée en fin de chapitre 3 pour le cas télécoms.

Identifier les *Use Case* est donc uniquement un travail d'introspection à fin de capture, appréciation et éventuellement validation des besoins à prendre en compte. À notre sens, parce que cela n'est pas de la modélisation, nous parlons plus volontiers ici et encore une fois d'ingénierie des besoins.

Les modèles complets et rigoureux qui vont défiler dans la section 6.5 sont cependant suffisamment difficiles à élaborer pour qu'une phase amont ou préparatoire existe. En ce sens, les *Use Case* sont un outil de réflexion pouvant épauler une phase d'investigation, préalable à la spécification proprement dite. En termes de formalisme néanmoins (relations entre cas d'utilisation, points d'extension…), les *Use Case* d'UML 2.x restent ici aussi inintéressants.

En clair, chercher les grandes fonctionnalités du système est une voie de réflexion naturelle et efficace, sans avoir à se soucier d'une notation sophistiquée telle que peut l'être celle des *Use Case* d'UML 2.x. Examinons dans ce cadre ce qui paraît plus essentiel : les *Sequence Diagram* qui peuvent en effet servir d'outil de « décantation » des besoins. Comprenons bien alors que ce qui apparaît en figure 6.3 n'est pas un modèle (un schéma contractuel) mais un support de réflexion ou encore de discussion en amont de la fabrication des modèles proprement dite. Le but est donc de substituer au texte des sections 6.2 et 6.3 des graphiques qui « parlent ». Le problème clé de l'élaboration d'un *Sequence Diagram* est de savoir si ce qui y apparaît « peut » se produire ou « doit » se produire ? « Peut se produire » revient à documenter le système sur des cas remarquables (l'exhaustivité étant irréaliste) de fonctionnement, cas auxquels les utilisateurs sont sensibles et réactifs. L'ingénieur des besoins qui réalise ces représentations a alors en main un outil d'évaluation de sa compréhension personnelle du système en regard des détails du *Sequence Diagram*. « Doit se produire » atténue de façon significative le nombre de combinaisons de fonctionnement possibles sans pour autant simplifier les choses. En effet, cela revient indirectement à poser le problème de fiabilité, en s'obligeant à déterminer une stratégie de défense si ce qui apparaît dans le *Sequence Diagram* « déraille ». Autrement dit, on n'aborde pas les cas exceptionnels ou d'erreur mais le problème arrivera tôt ou tard à l'implémentation.

En résumé, on préconise ici une utilisation douce d'un formalisme de scénario, détachée de toute la batterie conceptuelle inhérente aux *Use Case Diagram* dans le but de décortiquer le cahier des charges. Soyons néanmoins honnêtes, nous avons triché pour construire le *Sequence Diagram* en figure 6.3 car nous avons utilisé les résultats de l'analyse détaillée du système qui apparaissent section 6.5. Dans la réalité, on aurait obtenu des scénarios moins précis, voire entachés de manques ou de défauts. Mais c'est l'esprit de l'ingénierie des besoins, qui définitivement n'est pas de la modélisation !

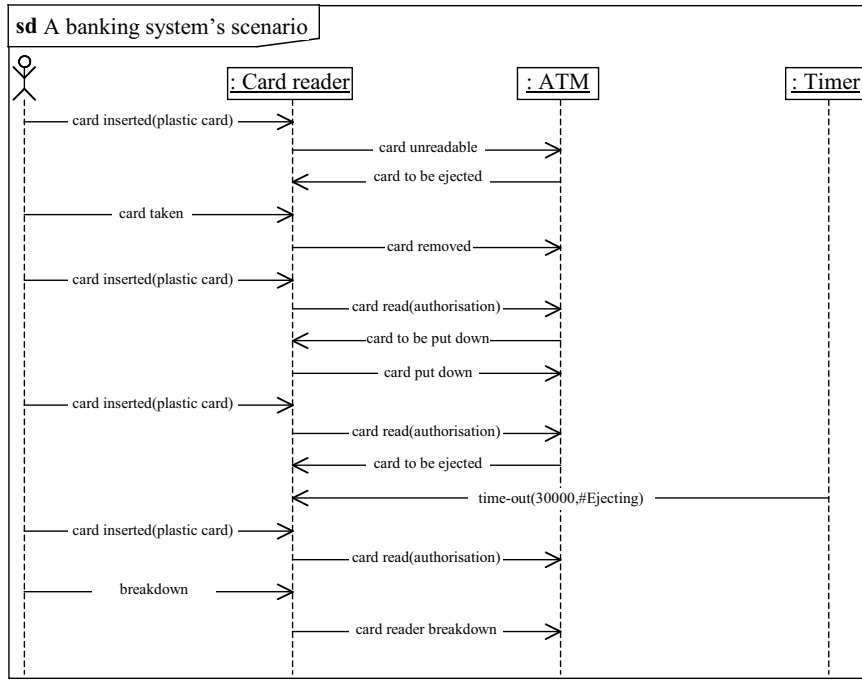


Figure 6.3 – Exemple de *Sequence Diagram* du système bancaire.

Le formalisme en figure 6.3 est délibérément épuré en regard de tout ce qu'offre UML. Il donne juste dans le temps une interaction possible entre une instance d'ATM, une de *Card reader* et une de *Timer*. En fait, on « paraphrase » le texte des sections 6.3.2 et 6.3.3 au moyen d'un graphique. On renvoie le lecteur à l'article de Harel sur les scénarios « vivants » (ou *Live Sequence Charts* [3]) pour une discussion critique sur le formalisme des scénarios dans UML. Rapidement, la principale déficience est le caractère anonyme des instances. En effet, n'importe quel ATM ne dialogue pas avec n'importe quel *Card reader*. Nous revenons dans ce chapitre sur ces problèmes ainsi que plus globalement sur l'ensemble des événements de la figure 6.3.

6.5 ANALYSE

L'architecture générale du système au niveau analyse fait apparaître quatre packages (figure 6.4) dont deux spécifiques ou métier : *_ATM* et *_Consortium*. Les deux autres, comme nous allons le voir par la suite, ont vocation de réutilisation. Les relations de dépendances UML entre ces packages montrent la forte cohésion de chacun d'eux et le couplage lâche de l'ensemble. C'est le but recherché lorsque l'on décrit des dépendances entre packages que de préciser comment s'opère la modularité au niveau système : les sous-systèmes logiques[1] et leurs interdépendances. Le schéma en figure 6.4 résulte de toute l'analyse qui suit dans ce chapitre. C'est donc un schéma *de syn-*

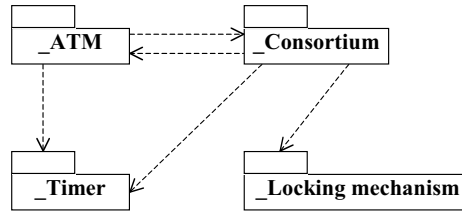


Figure 6.4 – Architecture d'analyse.

thèse puisqu'on ne peut l'obtenir que lorsque chaque type est bien spécifié et que les relations avec les autres types sont définies, notamment le flux de contrôle (événements) circulant qui aide au découpage (voir figure 6.5).

Notre objectif est de raffiner cette architecture d'analyse pour aller vers une description plus explicite et surtout justifiée des packages. La figure 6.5 montre pour les deux packages métier l'interdépendance en matière de flux de contrôle. Grâce à deux notes UML, on montre les types d'événements censés passer d'un package à l'autre, en l'occurrence, *start of* et *end of* pour les débuts et fins de transaction bancaire. *to be processed* et *to be canceled* pour les demandes de traitement et d'annulation d'opérations bancaires au sein des transactions. Dans l'autre sens, on a *response to -start of- not ok* et *response to -start of- ok* concernant l'échec ou le succès du contrôle d'autorisation, ainsi *response to -to be processed- ok* et *response to -to be processed- not ok* concernant le succès ou l'échec de traitement d'opération bancaire.

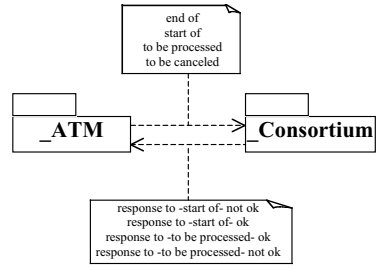


Figure 6.5 – Packages avec flux de contrôle.

Le flux d'événements circulant dans les deux directions représentées en figure 6.5 doit en effet être minimal en comparaison des flux de contrôle entre types d'objet de chaque package. Cette règle simple et de bon sens donne en spécification l'architecture logique. En conception et implémentation, elle doit guider la fabrication de composants logiciels en tant qu'entités autonomes de déploiement, offrant par exemple des stratégies plus variées d'équilibrage de charge réseau si les composants sont déployés sur des nœuds différents.

1. Le terme « sous-système » faisant usuellement référence à des architectures physiques, on précise ici le caractère logique de la segmentation.

En anticipant, on peut d'ores et déjà dire que la segmentation des packages est gouvernée par le fait que les types d'objet au sein des packages s'interrogent à propos de, ou « voient », leurs états respectifs. Cette propriété incite à les grouper dans le même package en rapport aussi à la fréquence des interrogations sur ces états. Seules les spécifications individuelles des types d'objet données par la suite mettent ou non en exergue ce genre de propriété.

6.5.1 Package *_ATM*

Ce package peut être considéré comme le côté client du système logiciel. En d'autres termes, les ATM en tant que dispositifs physiques embarquent un programme assurant l'interface entre usagers et le système central incarné en UML par un autre package (*_Consortium*), étudié dans la section 6.5.2.

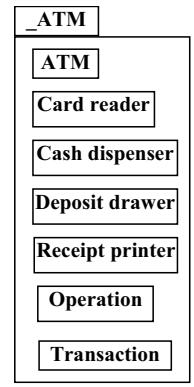


Figure 6.6 – *Class Diagram* de niveau le plus abstrait (package *_ATM*).

Package _ATM*,* Class Diagram *détaillés*

On a en figure 6.7 le type ATM composé de quatre périphériques. La relation d'agrégation ici utilisée n'a pas de sémantique rigide au sens où elle exprime intuitivement une dépendance physique des quatre dispositifs nécessaires et suffisants au fonctionnement de l'ATM. L'association avec le type d'objet *Consortium* marque l'interdépendance entre les deux packages métier de la figure 6.4. Les deux relations d'héritage sont plus intéressantes et caractérisent le fait que l'*ATM* et le *Card reader* vont être utilisateurs (clients) de services de cadencement. Ces services par essence « non-métier» sont décentrés dans le package *_Timer* apparaissant en figure 6.4 et expliqué au chapitre 5. L'idée évidente est de préparer des modèles UML assez généraux et donc réutilisables pour d'autres spécifications.

Le type *Cash dispenser* a deux attributs lui permettant de connaître à tout moment l'argent en stock (*cash on hand*) et l'argent jusqu'à présent délivré (*dispensed*). Le type *ATM* a l'attribut *station code* qui est son identifiant. Rappelons ici que

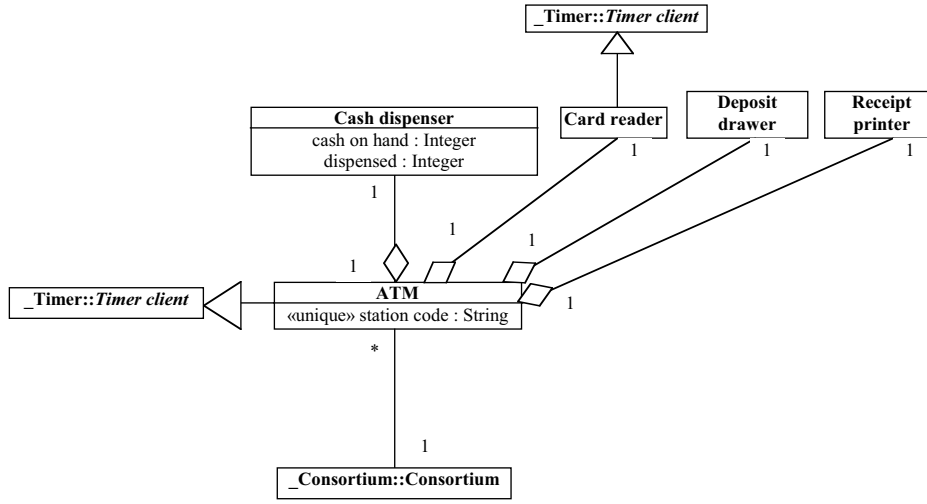


Figure 6.7 – Spécification des dispositifs nécessaires au distributeur automatique bancaire.

le stéréotype *«unique»* fait référence à une contrainte OCL utilisant l'opérateur *isUnique* sur les collections (voir chapitre 2).

On procède via différentes vues de manière à ce que chaque diagramme mette l'accent sur un aspect particulier du système. Alors que la figure 6.7 donne les liens plutôt physiques, la figure 6.8 fournit des liens logiques avec les types d'objet *Transaction* et *Operation*, lui-même relié au type *Account* du package *_Consortium*. Un *ATM* va gérer ou non (cardinalité *0..1* vers *Transaction*) une transaction courante (rôle *current* vers *Transaction*). L'autre association décrit les transactions passées (cardinalité * vers *Transaction*). Les opérations bancaires composent alors les transactions (agrégation et ordre : contrainte prédéfinie *{ordered}*) et concernent les comptes. La seconde association d'*Operation* vers *Account* dont le rôle est *secondary* est spécifiquement dédiée aux transferts qui touchent deux comptes alors que les autres types d'opération n'en concernent qu'un. Le type d'une opération est connu via l'attribut *type* dont le type UML est énuméré. On remarquera à ce titre que l'attribut *amount* a une cardinalité *0..1* car il peut être indéfini dans le cas où *type = query* (interrogation de compte). De manière plus générale, il y a beaucoup de contraintes qui s'appliquent sur les diagrammes de la figure 6.7 et de la figure 6.8. Avant d'y venir, voici en figure 6.9, un diagramme simplifié d'instances qui permet de mieux comprendre, voire de valider, le modèle de la figure 6.8.

Contraintes relatives au Package _ATM

Nous abordons ici un aspect clé d'UML à savoir l'usage conjoint et indispensable d'OCL. En effet, il y a lieu de supprimer les ambiguïtés et les imprécisions des modèles en spécifiant formellement des propriétés remarquables du système. Soit ces propriétés sont explicitées dans le cahier des charges, soit elles se signalent naturellement au moment de la modélisation. Dans les deux cas, il faut les écrire et si possible les implémenter de manière à accroître la cohérence globale et la fiabilité du logiciel final.

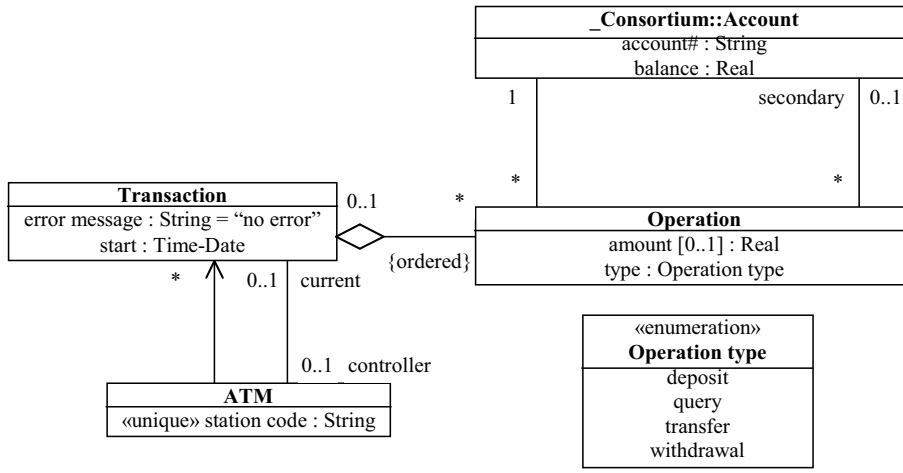


Figure 6.8 – Spécification des types *Transaction* et *Operation* du package *_ATM*.

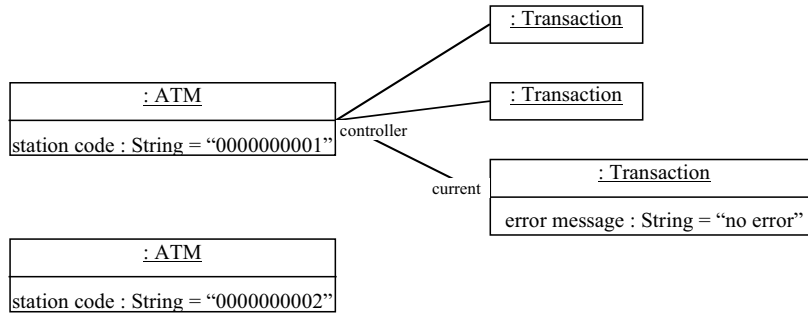


Figure 6.9 – Spécification d'instances en liaison avec le modèle de la figure 6.8.

Les transactions passées d'un *ATM* sont une sorte d'historique, et la transaction courante, si elle existe, montre que l'*ATM* est actif. La transaction courante n'est pas membre de l'historique ; elle intégrera donc cet historique (à modéliser par la suite dans les modèles dynamiques) à sa fermeture. Statiquement pour l'instant, on a donc :

```
(current→intersection(transaction))→isEmpty()context ATM inv:
```

Le montant de l'argent en billets stocké dans un *Cash dispenser* est toujours supérieur ou égal au « déjà délivré » :

```
context Cash dispenser inv:
   cash on hand >= dispensed
```

Le montant d'une opération est toujours strictement positif ou indéterminé dans le cas d'une interrogation. Par ailleurs, l'association entre *Operation* et *Account* dont le rôle est *secondary* côté *Account* est valorisée seulement, et toujours, dans le cas

d'un transfert. Pour en finir avec *Operation*, un transfert concerne deux comptes différents et appartenant au *même* client, *i.e.* accessibles par la même carte de crédit dite « logique » (voir plus loin). Voici l'ensemble :

```
context Operation inv:
   amount > 0
   type = query implies amount→Empty()
   type = transfer implies secondary→notEmpty()
   type <> transfer implies secondary→isEmpty()
   account <> secondary
   type = transfer implies account.logical card = secondary.logical card
```

La toute dernière navigation ci-avant s'appuie sur le modèle de la figure 6.25. Finalement, l'*ATM* sur lequel est archivée une transaction est l'*ATM* sur lequel elle s'est déroulée :

```
context Transaction inv:
   atm = controller
```

Type d'objet Transfer*, variations*

Proposer des modèles « qui marchent » sans étudier le processus de leur obtention n'a qu'un intérêt minime. Par ailleurs, des variantes de modèle existent avec autant de qualités ou des qualités différentes. Nous soumettons ici au lecteur un débat sur la spécification du type *Transfer*, c'est-à-dire trois autres modèles disons « corrects » mais en concurrence avec celui de la figure 6.8. L'idée est d'en extraire un parmi les quatre par le fait qu'il a des qualités supérieures ou en cas d'égalité, des qualités à privilégier qui sont ici l'extensibilité et une certaine rationalité.

Dans la figure 6.10, on note tout d'abord qu'*Operation* est une classe abstraite (en italique). Ensuite, chaque sous-type d'opération bancaire est matérialisé par une classe UML (suppression donc de l'attribut *type* dans *Operation* proposé en figure 6.8) avec des propriétés ciblées. Par exemple, le type *Query* ne dispose pas d'attribut *amount*. Seul *Transfer* entretient une seconde association avec *Account*. En consé-

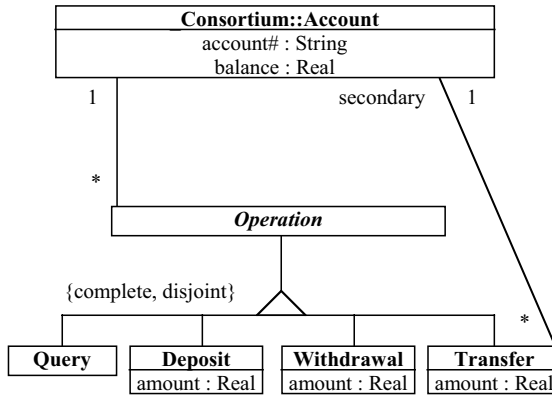


Figure 6.10 – Spécification du type *Transfert* à l'aide de l'héritage.

quence, les contraintes relatives au type *Transfer* dans la figure 6.10 sont plus limitées :

```
context Transfer inv:
   amount > 0
   account <> secondary
   account.logical card = secondary.logical card
```

Le modèle de la figure 6.10 est plus compréhensible que celui de la figure 6.8 car plus intuitif via l'usage de l'héritage, fondement de l'approche objet. En revanche, il est moins « ramassé » ou concis et introduit des sous-types avec des propriétés non discriminantes. Par exemple, *Deposit* et *Withdrawal* n'ont pas de différence. D'autres versions plus « originales » sont données en figure 6.11.

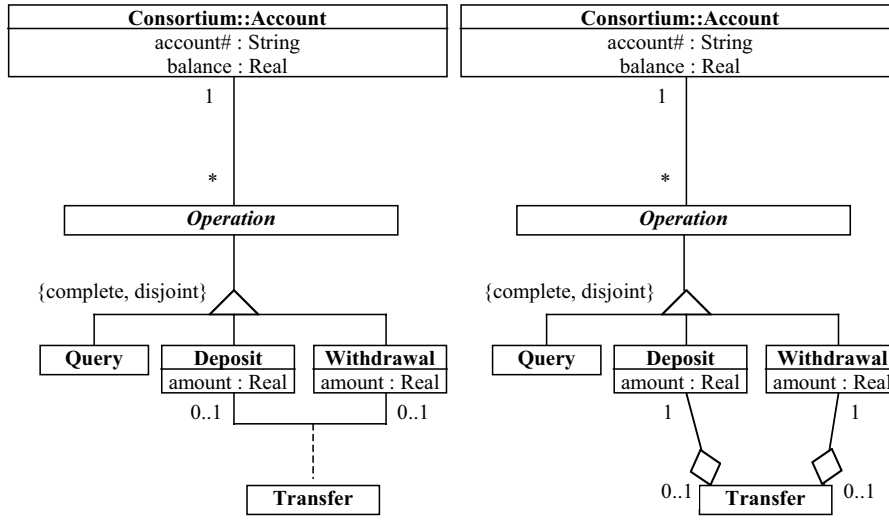


Figure 6.11 – Spécifications éclectiques du type *Transfert*.

Les deux modèles de la figure 6.11 partagent le même esprit : un transfert, c'est un retrait *et* un dépôt. Cela implique qu'il y a des « vrais » dépôts et des « vrais » retraits ainsi que des instances de retrait et de dépôt créées uniquement pour matérialiser des transferts. Les deux configurations sont sous-entendues par l'usage des cardinalités *0..1*. Cher lecteur, qu'en penser ? Les deux modèles de la figure 6.11 paraissent « tirés par les cheveux »… Par ailleurs, le maintien de la cohérence impose d'autres contraintes, comme le fait qu'un dépôt et un retrait d'un transfert ont le même montant :

```
context Transfer inv:
   deposit.amount = withdrawal.amount
   deposit.account <> withdrawal.account
   deposit.account.logical card = withdrawal.account.logical card
```

Notre position est que le modèle de la figure 6.10 est le plus apte à l'évolution si les besoins se développent. La mise en exergue des sous-types est un bon point pour augmenter leur spécificité. Dans la figure 6.8, l'accentuation de la différentiation entre *Query*, *Deposit*, *Withdrawal* et *Transfer* amènerait à casser la structuration choisie. Les modèles de la figure 6.8 et de la figure 6.10 sont eux plus rationnels que ceux de la figure 6.11. La qualité du modèle de la figure 6.8 est sa concision même s'il induit plus de contraintes que celui de la figure 6.10. Il permet de ne pas développer une « usine à gaz » si dans le futur les anticipations sur l'évolution des besoins s'avèrent exagérées. En d'autres termes, produire des modèles à vocation d'extensibilité a un coût, lui-même associé à un risque de non-retour sur investissement.

Package _ATM, Behavior Diagram

Les modèles dynamiques présentés sont pour l'essentiel des *State Machine Diagram*. Ils ont pour but de préciser la dynamique, si elle existe, de chacun des types d'objet du package *_ATM*. Ils sont par ailleurs et bien entendu essentiels à l'implémentation.

Distributeur d'argent

Le *Cash dispenser* dialogue avec l'ATM dont il fait partie. Outre les phases d'initialisation (état initial point noir), de réinitialisation (événement : *reset*) et de réparation éventuelle (événement : *repairing*), le *Cash dispenser* procède à l'évaluation de l'argent en stock (*enough?*) dans l'état « disponible » (*Idle*) et répond à l'ATM (actions : ^*atm.enough cash* ou ^*atm.not enough cash*) puis dans une autre phase délivre l'argent dans l'état *Working* en activant le dispositif physique (activité : *activate device*).

Les contraintes relatives au comportement de *Cash dispenser* dans la figure 6.12 sont :

```
context Cash dispenser::enough?(amount : Integer)
   pre: amount > 0
```

Ci-dessus il apparaît tout simplement que le montant requis est strictement positif. C'est au travers d'une précondition à la requête de demande s'il y a assez d'argent en stock (*enough?*) que la propriété est spécifiée.

```
context Cash dispenser::set cash on hand(cash : Integer)
   pre: cash > 0
   post: cash on hand = cash
```

La contrainte précédente reflète le réapprovisionnement en argent.

```
context Cash dispenser::set dispensed()
   post: dispensed = 0
```

À l'initialisation du *Cash dispenser*, l'assertion qui précède dit que le « délivré » (attribut *dispensed*) est nul.

```
context Cash dispenser::to be dispensed(amount : Integer)
   pre: amount > 0
   post: dispensed = dispensed@pre + amount
```

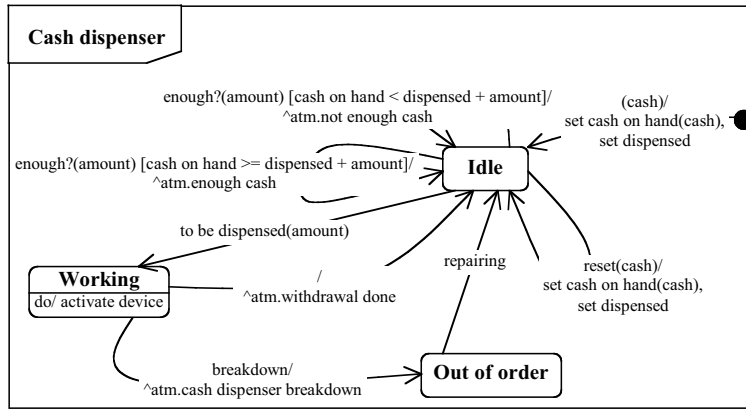


Figure 6.12 – Spécification du comportement du dispositif de distribution d'argent.

Cette dernière contrainte prouve que la nouvelle valeur de l'attribut *dispensed* est son ancienne valeur (*dispensed@pre*) plus le montant délivré.

Le *State Machine Diagram* en figure 6.12 peut paraître « mal fait ». En effet, de manière tout à fait évidente, la somme à distribuer mentionnée lors de l'arrivée de l'événement *to be dispensed* doit être systématiquement égale à la dernière somme soumise au contrôle « reste-t-il assez d'argent disponible ? » incarné par l'événement *enough?*. En allant plus loin, on pourrait imaginer qu'il n'y a pas à spécifier la somme à distribuer (enlèvement de l'attribut *amount* à l'événement *to be dispensed*) car c'est par définition la toute dernière somme contrôlée. Voici en conséquence en figure 6.13, une variante du *State Machine Diagram* de la figure 6.12, variante qui va produire une solution plus robuste sans néanmoins supprimer l'attribut *amount* de l'événement *to be dispensed*.

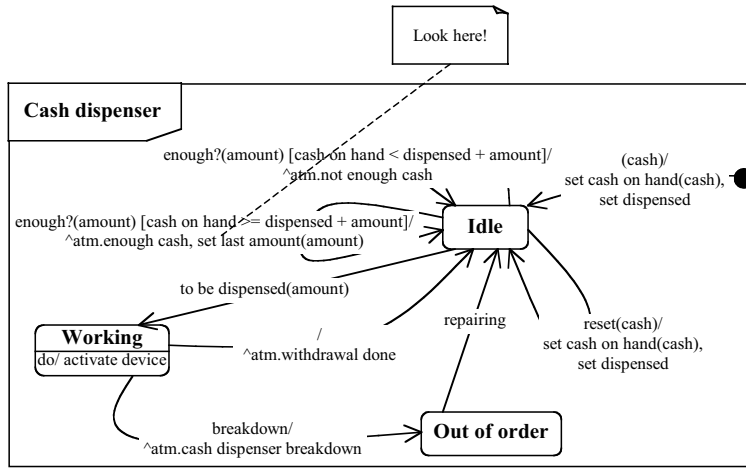


Figure 6.13 – Spécification du comportement du dispositif de distribution d'argent, première variante.

La différence en figure 6.13 est l'exécution d'une nouvelle action (*set last amount*) régie par la contrainte suivante :

```
context Cash dispenser::set last amount(amount : Integer)
   pre: amount > 0
   post: last amount = amount
```

En conséquence, le processus de délivrance d'argent tel que spécifié en figure 6.13 doit être revu via l'introduction d'une contrainte plus forte à satisfaire (seconde précondition) qui porte sur la variable locale au *statechart* appelée *last amount* :

```
context Cash dispenser::to be dispensed(amount : Integer)
   pre: amount > 0
   pre: last amount = amount
   post: dispensed = dispensed@pre + amount
```

La contrainte OCL qui précède garantit qu'on délivre la somme pour laquelle on avait préalablement reçu une requête de test de l'argent en stock. Le protocole de dialogue entre l'ATM et le *Cash dispenser* est donc :

- y a-t-il assez d'argent en stock pour distribuer « x € » (événement reçu *enough?(x)*) ? Oui (événement envoyé *enough cash*) ou non (événement envoyé *not enough cash*) ;
- « y € » à distribuer (*to be dispensed(y)*) est égal au tout dernier montant testé ($x = y$).

Le bilan est que le modèle de la figure 6.13 est plus robuste que celui de la figure 6.12. Pourquoi néanmoins cette complication en apparence ? En l'occurrence, en figure 6.13, il n'y a pas nécessité que l'événement *to be dispensed* porte l'attribut *amount* puisqu'on connaît le dernier montant testé (variable *last amount*) et que c'est celui-là qui doit être distribué. La surspécification réside dans la gestion de pannes (interruption de liaison/communication avec le consortium de banques, panne du *Cash dispenser* lui-même ou autres) qui impose une trace plus fine et une possibilité de reprise sur le contexte d'arrêt. Dans [1], il est intéressant de noter qu'en abordant le même cas d'étude sous l'angle de la spécification formelle, on tend à simplifier l'automate (pas d'attribut porté par l'événement *to be dispensed*) alors que cela n'est pas satisfaisant du point de vue de la qualité de service.

La dernière variante du *Cash dispenser* touche un autre problème, celui des transitions non étiquetées qui sont autorisées dans UML mais qui sont corrigées dans la figure 6.14 (introduction de l'événement *withdrawal realised*). Dans la figure 6.12, le passage de l'état *Working* à l'état *Idle* n'est pas le résultat de l'occurrence d'un événement. Dans la figure 6.14, le *Cash dispenser* s'envoie à lui-même l'événement *withdrawal realised*. Le mode d'interprétation d'événement *run-to-completion* assure que l'activité *activate device* n'est pas interrompue. Étiqueter toutes les transitions, c'est garantir une approche uniforme, plus formelle et plus à même de produire des spécifications exécutables, c'est-à-dire vérifiables, dans des ateliers de génie logiciel. C'est aussi, pour l'implémentation, un moyen plus aisé de dériver du code lorsque toutes les transitions sont déclenchées par des événements. En revanche, le seul

inconvénient est la lourdeur des spécifications, en l'occurrence des modèles moins intuitifs dès lors que des événements « fictifs » sont introduits.

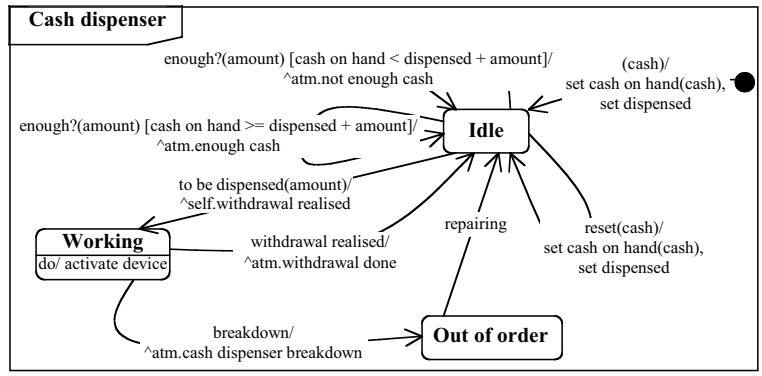


Figure 6.14 – Spécification du comportement du distributeur d'argent, seconde variante.

Lecteur de carte

Le *Card reader* a un comportement légèrement plus compliqué que le *Cash dispenser*. L'état de travail ou *Working* a trois sous-états caractéristiques qui sont *Busy* pour la phase où la carte plastique reste dans le lecteur le temps de la transaction, *Storing* pour la consignation de la carte lors de son oubli par l'utilisateur par exemple, et finalement *Ejecting* pour la restitution (figure 6.15). *Idle* est exclusif de *Working* et correspond à la période de non-activité. *Out of order* comme pour le *Cash dispenser* modélise l'état hors service. L'utilisateur est responsable de l'émission des événements *card inserted* lorsqu'il insère la carte et *card taken* lorsqu'il la récupère. L'ATM lui dispense les ordres *card to be ejected* pour entamer l'éjection et *card to be put down*

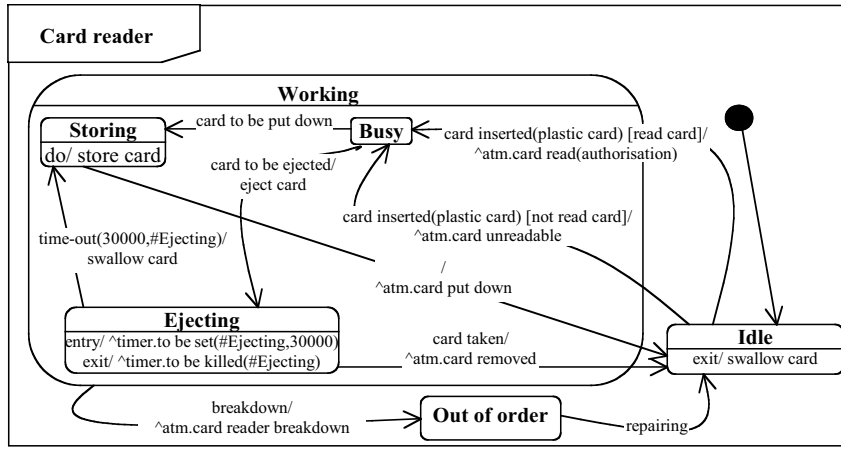


Figure 6.15 – Spécification du comportement du lecteur de carte.

to démarrer la consignation de la carte (en cas de tentative malveillante ou autre). Le *Card reader* est comme l'ATM cadencé, d'où le traitement d'un événement *time-out* provenant d'un *Timer* (voir aussi le lien d'héritage entre *Card reader* et *Timer client* en figure 6.7). Finalement les événements *breakdown* et *repairing* représentent la mise hors service et le dépannage.

Le lecteur de carte entretient dans le *State Machine Diagram* en figure 6.15, une relation avec un objet de type *Plastic card*. Cet objet entre en contact avec le lecteur de carte à réception de l'événement *card inserted* (voir attribut *plastic card* de cet événement). De plus, le lecteur de carte génère un objet de type *Authorisation* à destination de l'ATM (action : *^atm.card read(authorisation)*). En outre, l'état *Storing* correspond à la consignation de la carte en cas d'oubli par son utilisateur ou sur ordre de l'ATM (événement *card to be put down*). Aucun des *Class Diagram* déjà présentés dans ce chapitre (figure 6.7 en particulier) ne mentionne de relation entre *Card reader* et *Plastic card*. Cette relation volatile peut cependant être spécifiée (figure 6.16) pour améliorer la compréhension et éclairer l'implantation.

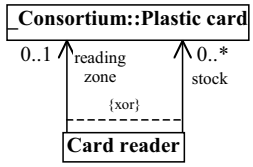


Figure 6.16 – Spécification détaillée du lecteur de carte.

Dans le modèle de la figure 6.16, on voit la présence d'une carte (rôle *reading zone*), éventuellement (cardinalité *0..1*), dans le lecteur. L'autre association à droite est le stock de cartes définitivement bloquées (rôle *stock*). On veut signifier que l'intersection entre le singleton (ou l'ensemble vide) des cartes dans la zone de lecture et, les cartes à jamais consignées, est vide. La contrainte *{xor}* dit formellement que le sous-ensemble du produit cartésien *Card reader x Plastic card* obtenu par l'association de gauche est disjoint du sous-ensemble du produit cartésien *Card reader x Plastic card* obtenu par l'association de droite. Certes, cela est vrai mais dans sa forme actuelle la contrainte *{xor}* n'autorise pas l'expression de l'exclusion sur deux extrémités d'association. Elle se prononce sur les produits cartésiens uniquement. La solution existe dans Syntropy par la formalisation *{Card reader::xor}* qui indique qu'on déclare une exclusion sur des sous-ensembles de *Plastic card* (*i.e.* on précise donc une direction de navigation qui part de *Card reader*) et pas sur des produits cartésiens *Card reader x Plastic card*.

En outre pour compléter la spécification, il est nécessaire d'avoir une post-condition au processus de stockage, post-condition qui précise que la carte en zone de lecture va s'ajouter aux cartes bloquées, cela dès que l'ATM détecte un problème (fraude, oubli de la carte après transaction…) :

```
context Card reader::store card()
   post : reading zone→isEmpty() and stock = stock@pre→union(reading zone)
```

Les autres contraintes sont la libération de la zone de lecture après éjection :

```
context Card reader::eject card()
   post: reading zone→isEmpty()
```

Finalement, l'avalement (activité *swallow card*) consiste à placer la carte dans la zone de lecture[1] :

```
context Card reader::swallow card()
   post: reading zone→notEmpty()
```

Dans les contraintes OCL qui précèdent, les valeurs des navigations *reading zone* et *stock* évoluent bien entendu en fonction des opérations mises en œuvre dans le *State Machine Diagram* du lecteur de carte. Cela reste néanmoins de la spécification assez détaillée dont l'intérêt dès l'analyse peut sembler faible, voire déplacé, vu le nombre important d'exigences de caractère plus impératif à prendre en compte.

Par ailleurs, il est utile de préciser que de nombreuses autres versions du comportement du lecteur de carte ont été élaborées avant d'en arriver à celle de la figure 6.15. Par exemple, *store card* est vue comme une activité (*do/* dans l'état *Storing*) alors que *swallow card* et *eject card* sont vues comme des actions. Pourquoi ? La dichotomie interruptibilité/ininterruptibilité n'est ici en effet pas manifeste pour décider de voir *store card* par exemple, comme interruptible ou non. On aurait aussi pu imaginer une opération *read card*, *i.e.* lire la carte alors que l'on a les gardes *read card* et *not read card*, autrement dit carte lue et carte non lue, gardes qui gouvernent la réponse à l'*ATM*. Difficile de justifier ces choix dans UML sinon que la version définitive de la figure 6.15 présente l'aptitude de s'implanter facilement via la bibliothèque *PauWare.Statecharts* ce qui n'est pas un mince avantage.

Récupération d'argent

Les deux *State Machine Diagram* qui suivent n'ont rien de singulier sinon des transitions non étiquetées comme pour le *Cash dispenser*. Pas de remarque particulière donc, à l'exception du fait de leur ressemblance importante, ainsi que de la présence d'états, incontournables et inévitables, que sont *Idle*, *Working* et *Out of order* aussi partagés par *Card reader* et *Cash dispenser*. Nous verrons par la suite qu'ils seront l'inspiration d'une factorisation par héritage.

Par ailleurs, l'implémentation du comportement du système de récupération d'argent (*Deposit drawer*, figure 6.17) basée sur la bibliothèque *PauWare.Statecharts*, impose qu'à la réception de l'événement *to be deposited*, une émission d'événement à usage interne, disons *go on*, permette à l'automate du *Deposit drawer* de rejoindre l'état *Idle*. Dans l'état actuel de l'automate, la transition n'est pas étiquetée. Alors que nous avons explicitement corrigé l'automate du *Cash dispenser* (différence entre la figure 6.13 et la figure 6.14), nous ne le faisons pas ici.

1. On remarque que le stockage s'enchaîne directement (voir figure 6.15).

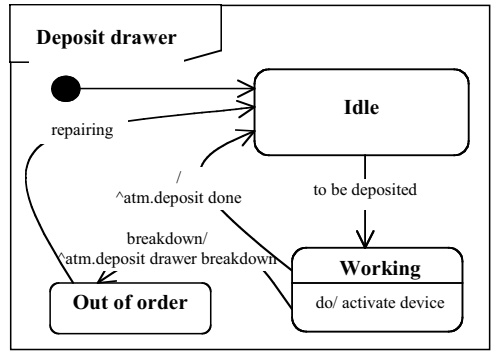


Figure 6.17 – Spécification du comportement du système de dépôt d'argent.

Impression de ticket

Dans le même esprit, le système d'impression de ticket (*Receipt printer*, figure 6.18) transite de *Working* à *Idle* sans événement déclencheur d'où le même ajout à l'implémentation par analogie avec *Deposit drawer*.

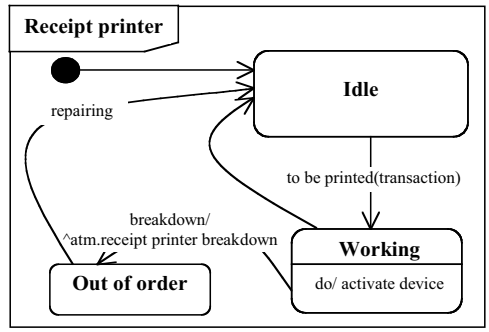


Figure 6.18 – Spécification du comportement du système d'impression de ticket.

Mise en commun des propriétés des périphériques d'ATM

La figure 6.19 propose l'introduction d'une classe abstraite *ATM device* pour factoriser les propriétés communes des périphériques d'*ATM*. Cette phase de capitalisation n'est pas toujours réaliste au moment du développement de l'application où les pressions temporelles de livraison sont importantes. En revanche, dans l'esprit du *design for reuse*, des gains substantiels sont possibles par la mise en évidence d'objets plus génériques.

Transaction bancaire

La transaction bancaire a elle aussi un comportement sommaire. On peut noter son processus d'archivage (*to be recorded*) qui la fait disparaître (point noir cerclé final).

Au-delà, il existe une cohérence forte entre les états vide (*Empty*) et non vide (*Not empty*) de *Transaction* et les relations qu'elle entretient avec *Operation* (figure

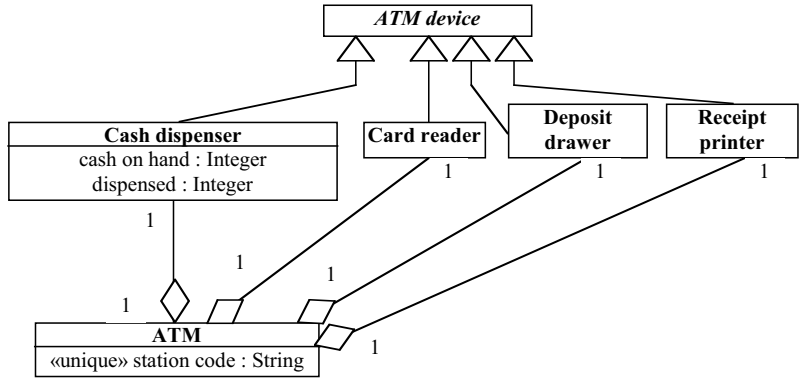


Figure 6.19 – Factorisation de propriétés comportementales dans un type *ATM device*.

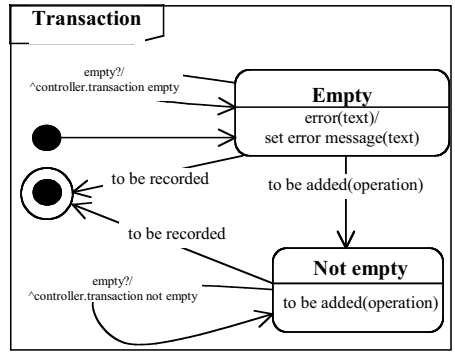


Figure 6.20 – Spécification du comportement de la transaction.

6.8). En l'occurrence une transaction vide est associée à zéro opération (la navigation *operation* est vide). Le contraire est aussi vrai, ce qui donne :

```
context Transaction inv:
   operation→isEmpty() implies oclInState(Empty)
   oclInState(Empty) implies operation→isEmpty()
   operation→notEmpty() implies oclInState(Not empty)
   oclInState(Not empty) implies operation→notEmpty()
```

On peut noter que d'un point de vue logique les deux dernières lignes de la contrainte sont inutiles car induites. En effet, *Empty* et *Not empty* ne sont que les deux seuls états possibles car le point noir et le point noir cerclé sont des états fictifs, *i.e.* des états matérialisant « l'inexistence ». Ces deux lignes sont donc ici écrites par souci d'évolutivité si jamais *Transaction* se voyait doter d'autres états.

Les autres contraintes portent sur les événements et les actions. À savoir pour les événements, le fait que la dernière opération terminée d'une transaction devient la première dans l'ordre des opérations déjà passées (contrainte *{ordered}* en figure 6.8) :

```
context Transaction::to be added(operation : Operation)
   post: operation→first() = operation
```

De plus, le prérequis à l'archivage est d'être la transaction en cours (rôle *current*) :

```
context Transaction::to be recorded()
   pre: controller.current = self
   post: controller.oclIsUndefined
```

Pour les actions maintenant, l'attribut *error message* est modifié par l'action *set error message* :

```
context Transaction::set error message(text : String)
   post: error message = text
```

Le début de transaction est enregistré via le positionnement de l'attribut *start* au moment courant. Pour cela, une opération générique appelée *creation* associée à la transition partant du point noir d'entrée en figure 6.20, matérialise la naissance d'une transaction :

```
   post: start = Time-Date.Now()context Transaction::creation()
```

On remarque ici l'utilisation du type *Time-Date* du chapitre 2.

Type d'objet *ATM*

Le comportement du type d'objet *ATM* est complexe. Par nécessité, nous avons scindé le *State Machine Diagram* en trois vues, partant du plus profond (détail) en allant au plus macroscopique (états les plus abstraits et donc possédant des états emboîtés). La première vue correspond à l'état *Transaction in progress* (figure 6.21) et modélise l'activité de traitement d'une transaction, notamment la possibilité de réaliser plusieurs opérations de différente nature au sein d'une transaction : retrait, dépôt, interrogation ou encore transfert.

Dans la figure 6.21, le seul état d'entrée (point noir) correspond à l'unique transition étiquetée par l'événement *response to -start of- OK(portfolio)* entrant sur l'état *Transaction in progress* en figure 6.22. En fait, la figure 6.21 donne la spécification détaillée de l'état *Transaction in progress* vu de manière abstraite (ou macroscopique) en figure 6.22. De façon symétrique, les états de sortie (point noir cerclé) en figure 6.21 mènent vers des états concrets en figure 6.22 : la transition étiquetée par l'événement *termination* par exemple rentre dans l'état *End* en figure 6.22. En figure 6.21, cette même transition est dirigée vers un état de sortie. De manière générale, la richesse du comportement de l'*ATM* contraint à ne pas pouvoir tout présenter sur un seul diagramme.

Les points remarquables de la figure 6.21 sont le choix du type d'opération à effectuer (*Operation kind choice*), le choix du montant (*Amount choice*) sauf si c'est une interrogation de compte, le choix du compte à opérer (*Account choice*) conduisant à deux choix en cas de transfert, la vérification du disponible sur le *Cash dispenser* en cas de retrait d'argent (*Amount verification*), et la vérification de l'opération elle-même (*Operation verification*) faite de manière externe par l'envoi de l'événement *to be processed* à *Consortium*. La gestion des cas d'erreur ou de rattrapage ainsi

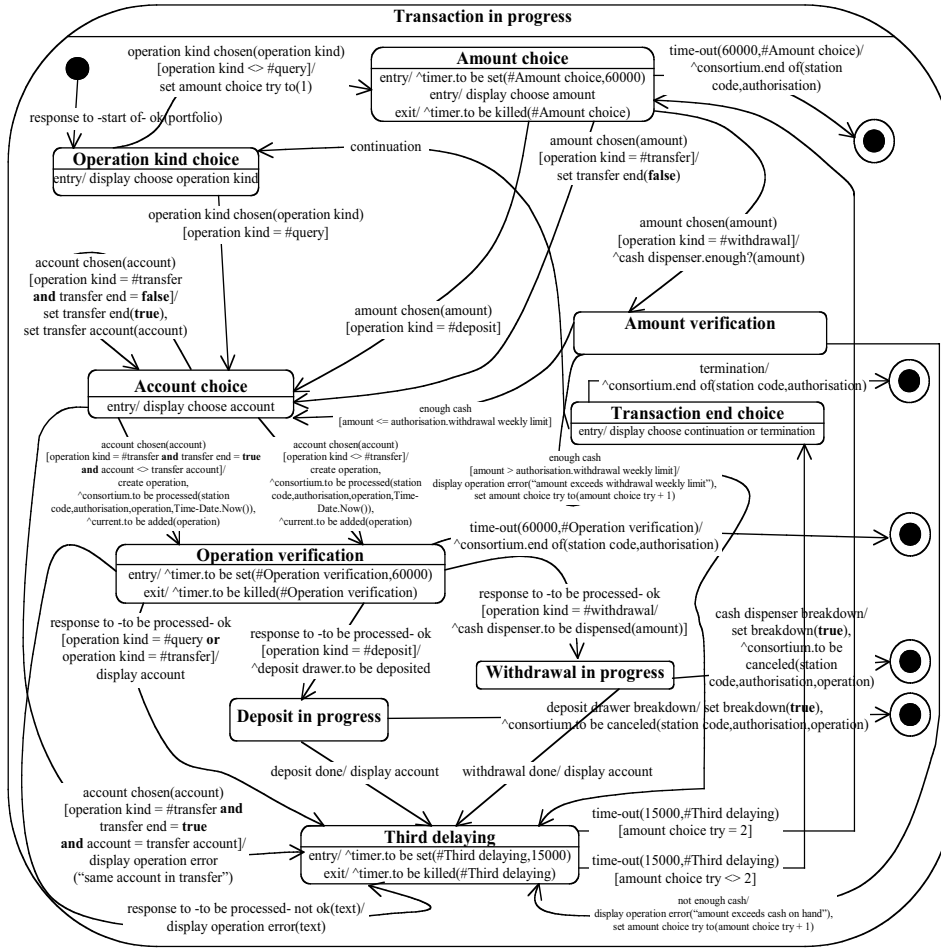


Figure 6.21 – Spécification du comportement du sous-état *ATM::Working::Transaction in progress*.

que le temps limité de certaines manipulations (envoi de l'événement *to be set* à *Timer*) engendrent un automate à états fini plutôt dense.

Les événements clés reçus par l'*ATM* restent néanmoins *response to -to be processed- ok* en cas d'acceptation de l'opération par le consortium de banques ou *response to -to be processed- not ok* en cas de refus expliqué par l'attribut d'événement *text*. En cas de succès, le portefeuille (*portfolio*) est véhiculé par l'événement *response to -to be processed- ok*.

Il y a lieu de préciser les contraintes qui s'appliquent, notamment celles concernant les attributs d'événement. Par exemple, l'événement *account chosen* est porteur de l'objet *account*. Ce compte bancaire n'est pas quelconque : il appartient au portefeuille client. La classe *Portfolio* en figure 6.25 matérialise cette notion de portefeuille client. Une instance de cette classe, sous le terme *portfolio*, est portée par

l'événement *response to -start of- OK* en figure 6.22. La contrainte est que le compte choisi est membre parmi tous les comptes composant le portefeuille d'un client, ce qui donne :

```
context ATM::account chosen(account : Account)
   pre: portfolio.account→includes(account)
```

D'autres termes sont utilisés décrivant le plus souvent des variables locales au *State Machine Diagram* comme le booléen *transfer end*. On peut prendre soin de les déclarer : décrire leur type en particulier ce qui est important. En Syntropy [2], une partie textuelle est à cet égard réservée en dessous de l'automate pour cette déclaration. Sinon, d'autres objets naissent dynamiquement et il y a lieu de l'expliquer. Par exemple, le terme *operation* (figure 6.21, sur une transition en entrée de l'état *Operation verification* et comme attribut de l'événement envoyé *to be processed*) désigne l'instance d'*Operation* qui se crée dans le processus *Transaction in progress* :

```
context ATM::create operation()
   post: operation.oclIsKindOf(Operation) and operation.oclIsNew
```

L'opérateur *oclIsNew* d'OCL est l'outil idoine pour décrire les créations d'objet dans les post-conditions. C'est en tout cas la seule manière correcte de décrire des créations d'objet.

De manière complémentaire, la figure 6.22 augmente la description du comportement de l'*ATM*. Cette figure décrit la phase d'ouverture et de fermeture de transaction au sein de laquelle a lieu une série d'opérations bancaires dans l'état *Transaction in progress* explicité en figure 6.21. Il faut comprendre que la figure 6.22 englobe la figure 6.21. Pour des raisons évidentes, nous n'avons pas fusionné ces deux figures pour ne voir qu'un seul *statechart* qui serait, par son volume, difficilement intelligible.

Dans la figure 6.22, apparaît le terme *portfolio* en tant qu'attribut de l'événement *response to -start of- ok* (sortie de l'état *Transaction verification*). Ce portefeuille client n'est pas quelconque : c'est celui du client propriétaire de la carte insérée ayant engendré la transaction dans laquelle on se trouve. L'événement *card read* émis par le *Card reader* et reçu par l'*ATM* (sortie de l'état *Start*) en figure 6.22 est porteur de l'attribut *authorisation* instance d'*Authorisation* (voir figure 6.25). Une instance de cette classe est intuitivement ce qui est codé sur la bande magnétique et/ou sur la puce de la carte bancaire. En utilisant les navigations disponibles en figure 6.25, on caractérise alors *portfolio* comme suit :

```
context ATM::response to -start of- ok(portofolio : Portfolio)
   pre: portfolio = authorisation.plastic card.logical card.client.portofolio
```

La lecture de la carte plastique qui a été insérée (événement *card read*) fait apparaître une instance d'*Authorisation* (attribut *authorisation* de *card read*) et surtout ouvre la transaction « courante ». Le rôle *current* vers *Transaction* dans la figure 6.8 évolue alors comme suit :

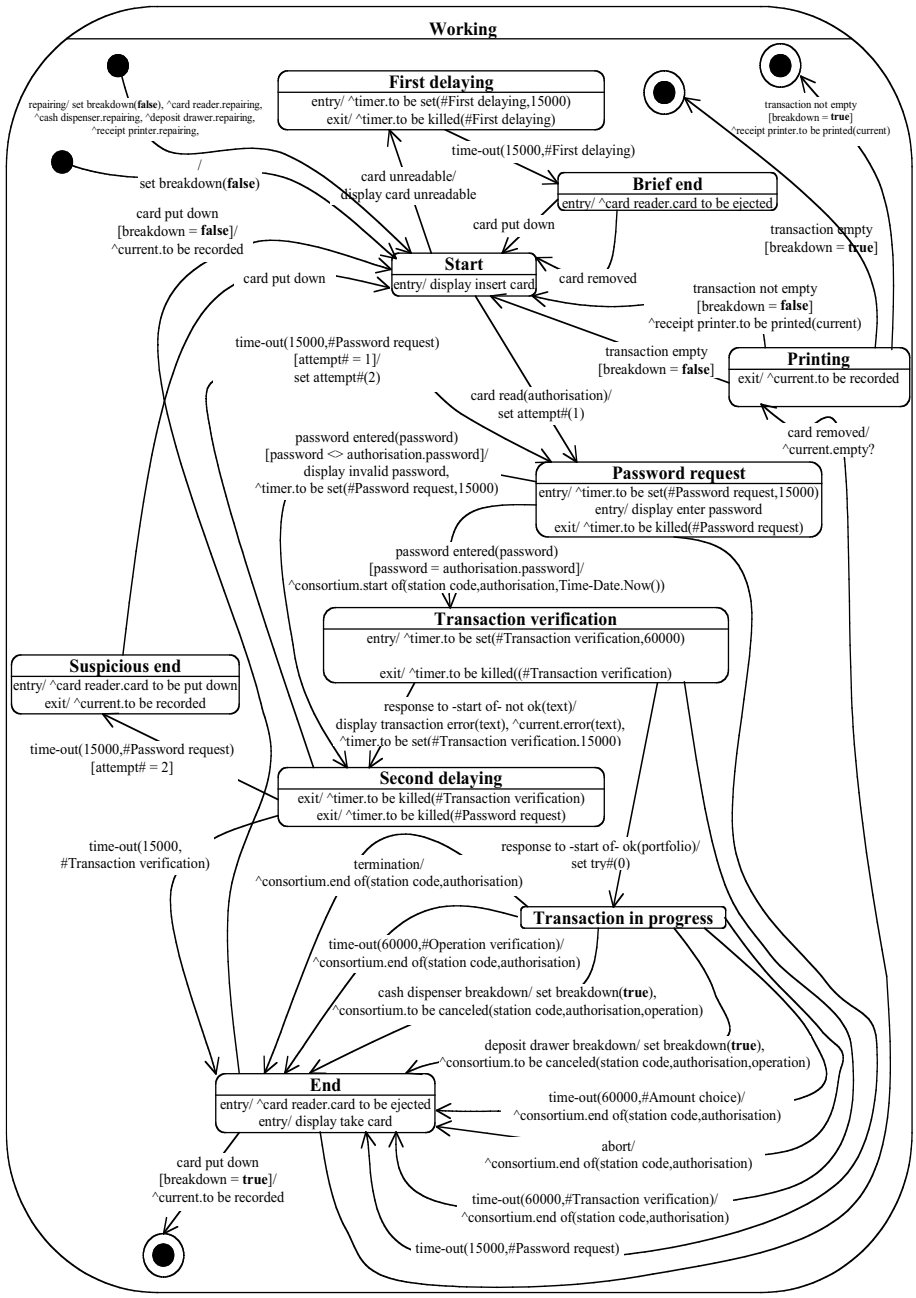


Figure 6.22 – Spécification du comportement du sous-état *ATM::Working*.

```
context ATM::card read(authorisation : Authorisation)
   post: current.oclIsNew
```

Finalement le *State Machine Diagram* de plus haut niveau (figure 6.23) est extrêmement concis du fait de l'occultation des détails de comportement relatifs à l'état *Working*. Il n'est là que pour modéliser des pannes de l'ATM ou de ses dispositifs (état *Out of order*) avant remise en route (événement *repairing*).

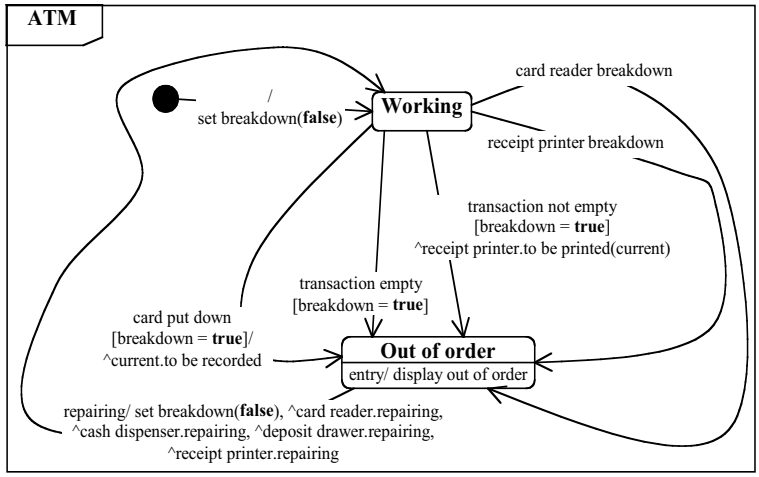


Figure 6.23 – *State Machine Diagram* de plus haut niveau du type d'objet *ATM*.

6.5.2 Package *_Consortium*

Ce package peut être considéré comme le côté serveur du système logiciel global. Plus exactement, le package *_Consortium* est un ensemble de services pour les ATM installés à divers endroits géographiques. En effet, si l'on s'en remet à ce que l'on voit en figure 6.1, le consortium apparaît comme un intergiciel permettant de s'abstraire des processus de gestion financière propres à chaque banque.

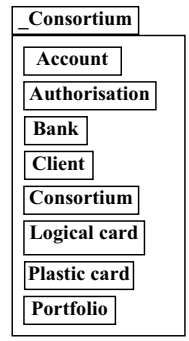


Figure 6.24 – Package *_Consortium*, liste des types d'objet.

Les types d'objet banque (*Bank*), client (*Client*) et compte bancaire (*Account*) forment une ossature commune au consortium. Dans la figure 6.25, le code banque (*bank code*) identifie chaque banque mais les clients et les comptes bancaires vus au niveau du consortium, sont différenciés via « leur » banque de rattachement (cardinalité *1* vers *Bank*). En UML il y a pour un objet client, un numéro client « local » (*client PIN*) propre à la banque du client. Un *qualifier* portant sur cet attribut *client PIN* s'ajoute pour signifier qu'étant donné un code banque et un numéro client, il y a ou non (cardinalité *0..1* côté *Client*) un client correspondant à cette identification. La contrainte *{subsets clients}* précise alors que le singleton ou l'ensemble vide obtenu par la navigation où se trouve le qualifier (*client PIN*) est inclus dans l'ensemble général des clients d'une banque (cardinalité * côté *Client*). Un raisonnement tout à fait similaire s'applique pour le type d'objet *Account* avec l'attribut *account#* (numéro de compte bancaire). Finalement, le type d'objet *Bank* se relie à un type d'objet *Consortium* pour formaliser le concept même de regroupement des organismes bancaires. Nous verrons en particulier que cette association est le support des communications entre les instances de *Bank* et l'instance de *Consortium*. Le statut bien particulier du type d'objet *Consortium* qui n'a qu'une seule instance possible dans le système, peut être mieux formalisé avec OCL :

```
context Consortium inv:
   Consortium.allInstances→size() = 1
```

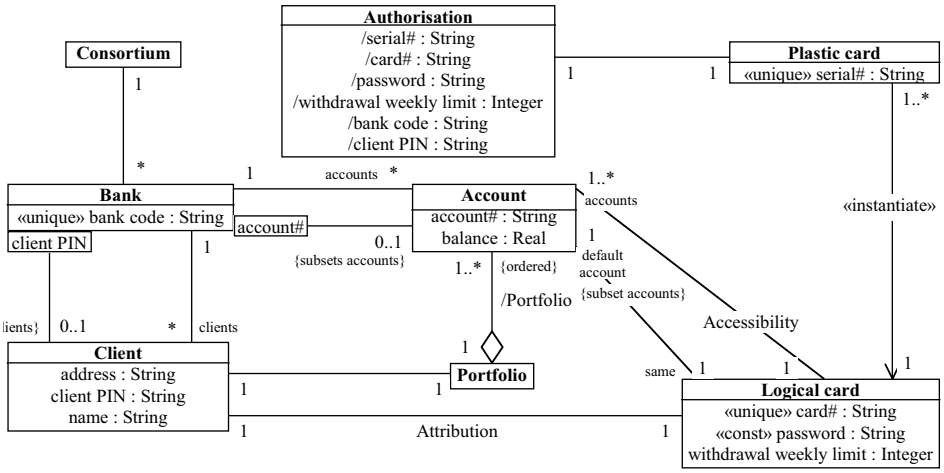


Figure 6.25 – Spécification détaillée du package *_Consortium*.

Package _Consortium, Class Diagram *détaillés*

Les types restant sont le portefeuille d'un client (*Portfolio*) vu comme un ensemble ordonné (contrainte *{ordered}*) de comptes bancaires. Par convention, le premier compte (qui existe toujours du fait de la cardinalité *1..** côté *Account*) est le compte par défaut sur lequel s'opèrent les opérations bancaires. En OCL, cela donne :

```
context Portfolio inv:
   account→first() = client.logical card.default account
   account = client.logical card.account
```

La seconde partie de la contrainte signifie que les comptes d'un portefeuille (terme *account*) sont les mêmes choses que les comptes accessibles par une carte logique (terme *logical card*). Cette carte logique est celle « du » client (cardinalités *1* de chaque côté de l'association *Attribution*) possédant ledit portefeuille. En fait, l'association entre *Portfolio* et *Account* est dérivée (*/Portfolio*) car déductible via la seconde partie de la contrainte.

Les types *Logical card* et *Plastic card* garantissent la distinction entre l'accès logique et l'accès physique à l'infrastructure des *ATM*. En d'autres termes, un client au sein du consortium a un numéro de carte logique qui lui est assigné (*card#*) ainsi que le mot de passe (*password*) et la limite hebdomadaire de retrait (*withdrawal weekly limit*), le tout formant l'entité *Logical card*. Les instances de *Plastic card* ont alors une réelle existence physique perceptible via leur numéro de série (*serial#*). Elles sont créées comme mentionné dans le cahier des charges à la demande du client. L'association entre *Logical card* et *Plastic card* a une cardinalité égale à *1* côté *Logical card* et le stéréotype *«instantiate»* en figure 6.25 pour bien montrer que *Logical card* est un modèle conceptuel pour *Plastic card*. À l'usage, un client ayant réclamé plusieurs cartes physiques a donc le même mot de passe pour ses cartes et la limite de retrait ne se cumule donc pas car elle est propre à la carte logique. Rappelons par ailleurs que le stéréotype *«instantiate»* (assimilé au concept de *metadata* d'OMT au chapitre 2) rend constant (instance non interchangeable) l'objet « modèle ». Ici, l'instance de *Logical card* liée à un objet *Plastic card* est figée en terme d'identité.

Tous les comptes d'un usager d'une banque ne sont pas forcément accessibles, d'où la nécessité de lister les seuls sujets à opérations bancaires sur les ATM : c'est le but recherché par l'association *Accessibility* entre *Logical card* et *Account* suppléée par une seconde association portant le rôle *default account* pour l'identification du compte par défaut. La contrainte *{subsets accounts}* précise alors naturellement que le singleton *default account* est inclus dans tous les comptes manipulables via une carte. Vu d'un compte bancaire, la carte logique obtenue par l'une ou l'autre des associations est obligatoirement la même :

```
context Account inv:
   same = logical card
```

Le dernier type d'objet à étudier est *Authorisation*. Il n'est pas vraiment intuitif dans le *Class Diagram* de la figure 6.25 mais est intensivement utilisé dans les *State Machine Diagram* qui suivent. Il représente en fait les informations qui doivent être portées sur la bande magnétique d'une carte plastique. En l'occurrence toutes les propriétés d'*Authorisation* sont établies (dérivées et donc préfixées d'un / en figure 6.25) en fonction des autres types d'objet comme suit :

```
context Authorisation inv:
   serial# = plastic card.serial#
   card# = plastic card.logical card.card#
   password = plastic card.logical card.password
```

```
withdrawal weekly limit = plastic card.logical card.withdrawal weekly limit
bank code = plastic card.logical card.client.bank.bank code
client PIN = plastic card.logical card.client.client PIN
```

Les contraintes restantes relatives à la figure 6.25 portent sur les attributs de *Logical card* :

```
context Logical card inv:
   password.size() = 4 -- password est de type String supportant l'opérateur size()
   withdrawal weekly limit > 0
```

La dernière contrainte est le fait que le code banque des comptes d'un client est le même que celui participant à l'identifiant du client :

```
context Client inv:
   bank.bank code = logical card.account.bank.bank code→asSet()
```

L'opérateur *asSet()* permet de transformer l'objet de type *Bag*[1] obtenu (collection de valeurs d'attributs *bank code* non nécessairement distinctes) en ensemble. L'égalité s'applique alors par le fait que ses deux membres sont des singletons comportant le même élément.

Finalement, le lecteur peut détecter le besoin d'une association entre *Client* et *Account* pour matérialiser la possession. Non nécessaire, cette dernière est alors dérivée : */Ownership* en figure 6.26.

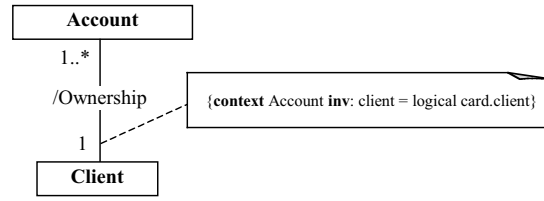


Figure 6.26 – Association dérivée additionnelle aidant à la compréhensibilité.

Package _Consortium, Behavior Diagram

Le type d'objet *Consortium* assure des tâches essentielles qui sont la prise en charge des requêtes émanant des *ATM*. De manière évidente, *Consortium* travaille en parallèle via sept processus (figure 6.27) utilisant de manière concurrente quatre ressources qui sont :

- *GonyVA* (*Group of not yet Verified Authorisations*) est une liste d'instances d'*Authorisation* à vérifier. Un premier processus (figure 6.27, état *Start of transaction*) traite toutes les occurrences de requête *start of* provenant des *ATM*. Ce processus procède à l'insertion dans *GonyVA* de l'autorisation portée par une occurrence de l'événement *start of*. À cause d'usages concomitants de

1. Type OCL ou Smalltalk bien connu désignant une collection quelconque (non ordonnée, présence éventuelle de doublons par opposition à *Set*…)

cette ressource, des tentatives de verrouillage se produisent (*to be locked*) associées à des déverrouillages (*to be unlocked*) dès lors que l'insertion est passée. Ces mécanismes sont ceux du type *Locked list<T>* du chapitre 3 dont la mise en œuvre ici particulière est présentée dans la section 6.5.3. De façon complémentaire et cohérente, un second processus en tâche de fond (figure 6.27, état *Authorisation checking*) vide continuellement *GonyVA* en envoyant chaque autorisation à la banque concernée. On voit ici que les autorisations vont être acceptées ou refusées par les banques sur la base de politiques complètement externes au cahier des charges initial. En cas d'acceptation, c'est ce même processus explicité en figure 6.28 à l'aide d'un *Activity Diagram* qui va manipuler une autre ressource critique : *GoaVA* ;

- *GoaVA* (*Group of already Verified Authorisations*) est une liste d'instances d'*Authorisation* vérifiées. Le processus chargé d'interpréter les demandes de fin de transaction émanant des *ATM* (figure 6.27, état *End of transaction*) vide jusqu'à épuisement *GoaVA*. Comme *GonyVA*, *GoaVA* est sujette à des accès concurrents. De façon opportuniste, les comportements de ces deux types d'objet sont factorisés dans des modèles standard présentés dans la section 6.5.3 ;
- *GonyPO* (*Group of not yet Processed Operations*) est remplie par le processus caractérisé par l'état *Operation processing request* et vidée par le processus incarné par l'état *Operation processing*. Son fonctionnement est similaire à *GonyVA* et bénéficie donc du même genre de standardisation que *GonyVA*.
- *GoCO* (*Group of Canceled Operations*) est remplie par le processus caractérisé par l'état *Operation canceling request* et vidée par le processus incarné par l'état *Operation canceling*.

Le *State Machine Diagram* en figure 6.27 montre par la séparation avec des traits pointillés l'existence des sept processus. Seuls les trois du haut du schéma se déclinent en modèles plus détaillés apparaissant dans la figure 6.28 (*Authorisation checking*), la figure 6.29 (*Operation processing*) et la figure 6.30 (*Operation canceling*). Du fait de leur grande ressemblance, nous allons expliquer le fonctionnement du processus de contrôle des autorisations en figure 6.28, cela en corsant quelque peu les difficultés puisque nous utilisons un *Activity Diagram* au lieu d'un *State Machine Diagram*. Rappelons ce qui a été dit au chapitre 3 : un *Activity Diagram* n'est jamais qu'une sorte de *State Machine Diagram* (UML 1.x) ou un type de diagramme assez concurrent (UML 2.x). Nous en profitons d'ailleurs ici pour faire une évaluation critique de ce formalisme des *Activity Diagram* sur un cas concret cette fois.

Le processus de contrôle des autorisations (figure 6.27, état *Authorisation checking*) est réveillé par un événement *newly not empty* et mis en veille par un événement inverse *newly empty*. Ces deux événements sont émis par un objet de type *Locked list<T>* (voir chapitre 3) à la condition de s'être enregistré auprès de lui comme « écouteur », cela sur le modèle des *JavaBeans*. Le processus se déclare en l'occurrence écouteur (événement envoyé *register*), à sa création, auprès d'objets

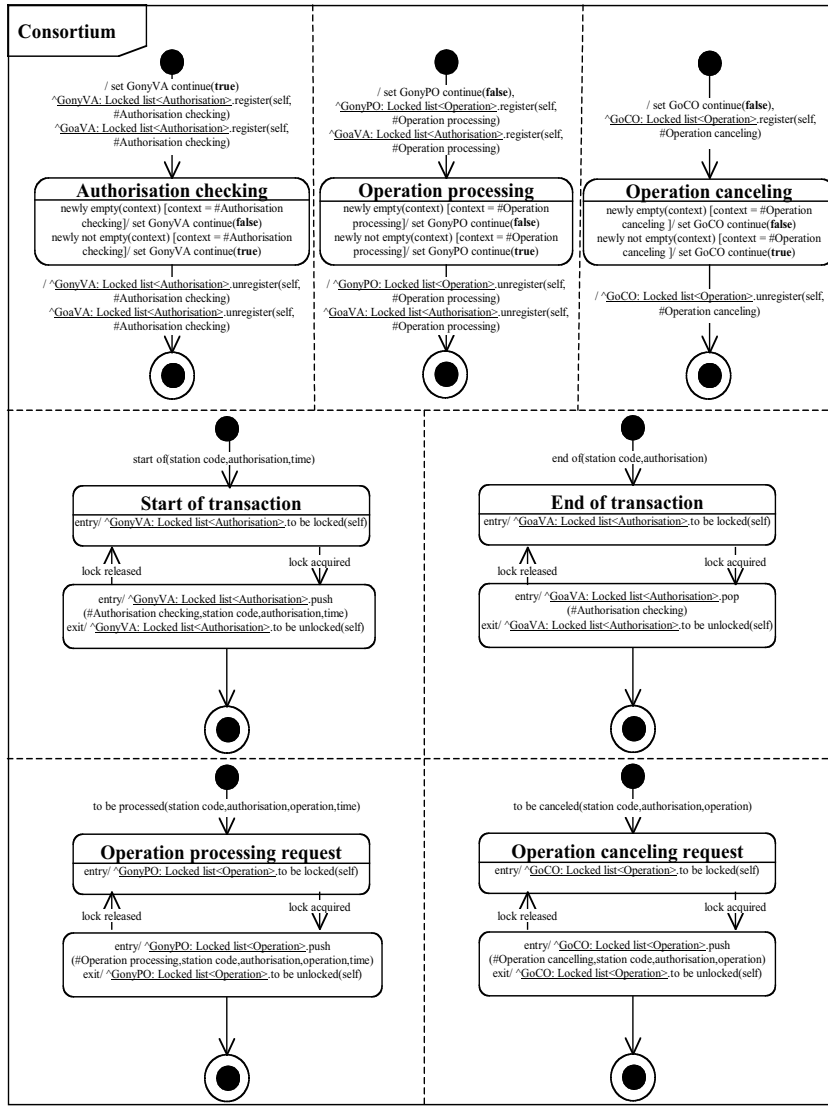


Figure 6.27 – *State Machine Diagram* de plus haut niveau du type d'objet *Consortium*.

explicitement nommés (notation soulignée en UML) instances de *Locked list<T>* : *GonyVA: Locked list<Authorisation>* et *GoaVA: Locked list<Authorisation>*. À sa mort, le processus se retire en tant qu'écouteur (événement envoyé *unregister*).

Un booléen local au *State Machine Diagram* appelé *GonyVA continue* est positionné à « vrai », respectivement à « faux », lors de l'apparition de *newly not empty*, respectivement *newly empty*. C'est ce booléen utilisé comme une garde au début de l'automate en figure 6.28 qui gouverne le traitement proprement dit des autorisations, à savoir l'envoi à la banque concernée pour contrôle interne (politique d'acceptation ou de refus de transaction propre à la banque), l'attente de la récep-

tion de la réponse de la banque (*authorisation checked*) et finalement l'interprétation du résultat via l'évaluation de la valeur des attributs *portfolio* et *text* de l'événement *authorisation checked*. Le processus en figure 6.28 assure le routage du résultat définitif à l'ATM concerné via la production des événements *response to -start of- ok* (succès) ou *response to -start of- not ok* (échec).

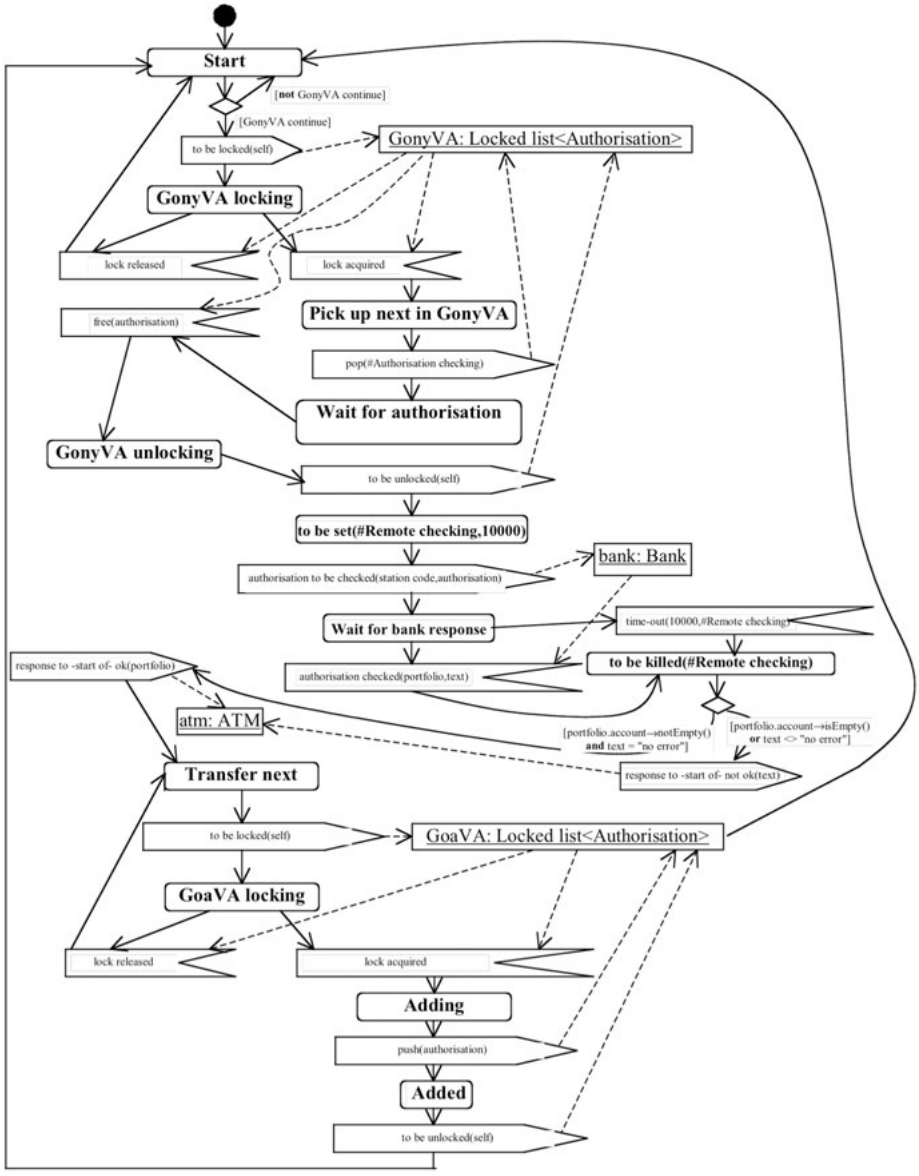


Figure 6.28 – *Activity Diagram* du processus de contrôle des autorisations.

Le schéma en figure 6.28 présente beaucoup de défauts, par exemple : quelle est l'instance exacte de *Bank* qui est censée recevoir l'événement *authorisation to be checked* ? Sans approfondir, notons qu'il existe des possibilités d'interprétation à géométrie variable de modèles tels que ceux de la figure 6.28. En ce sens, les *Activity Diagram*, de notre point de vue, sont plus « faibles » que les *State Machine Diagram*.

La figure 6.29 et la figure 6.30 incluent des modèles tout à fait semblables à celui de la figure 6.28. Nous ne les commentons pas et nous nous limitons ci-après à lister l'ensemble des contraintes s'appliquant au comportement tout entier de *Consortium*. Tout d'abord, voyons les post-conditions des actions de positionnement des variables servant de gardes : *GonyVA continue*, *GonyPO continue* et *GoCO continue* :

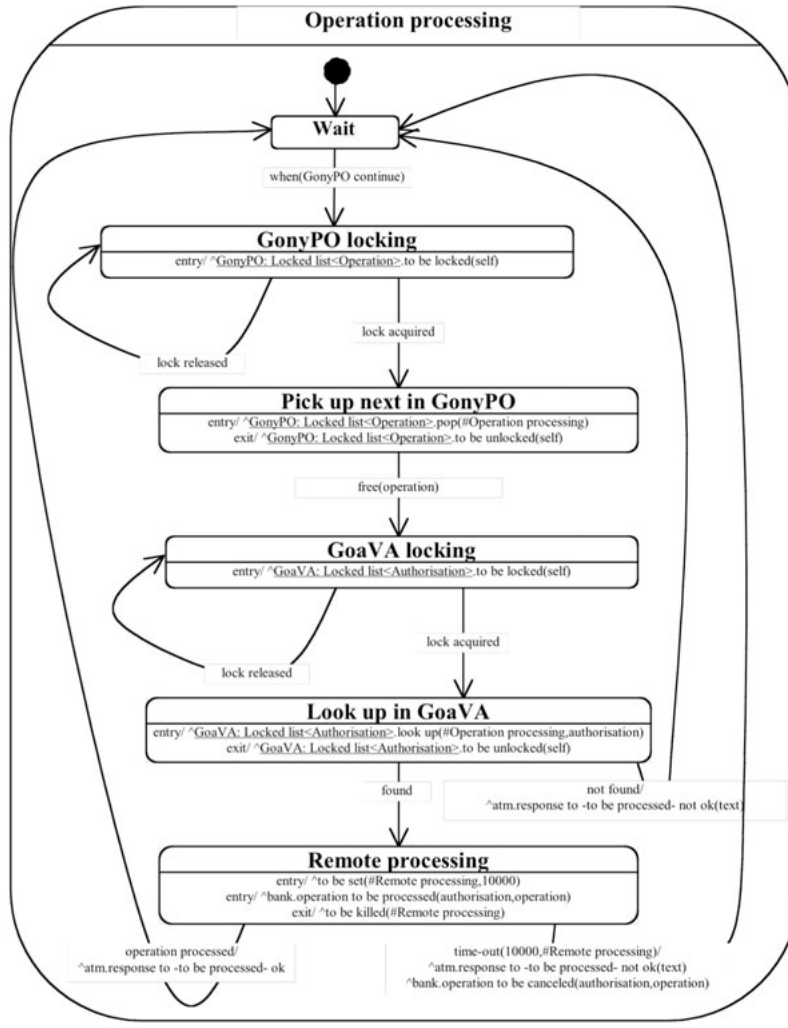


Figure 6.29 – *State Machine Diagram* du processus de traitement des opérations bancaires.

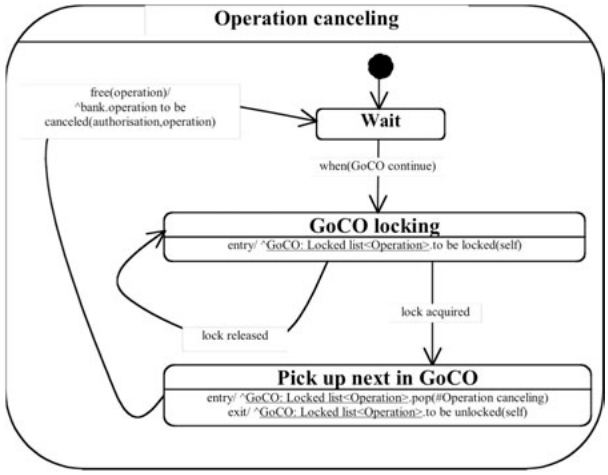


Figure 6.30 – *State Machine Diagram* du processus d'annulation des opérations bancaires.

```
context Consortium::set GonyVA continue(b : Boolean)
   post: GonyVA continue = b
context Consortium::set GonyPO continue(b : Boolean)
   post: GonyPO continue = b
context Consortium::set GoCO continue(b : Boolean)
   post: GoCO continue = b
```

On remarque que la ressource *GoaVA* est, contrairement à ses trois consœurs, vidée à la demande, c'est-à-dire chaque fois qu'une requête *end of* est reçue d'un *ATM* et non par une tâche de fond.

Sinon, d'autres contraintes plus subtiles peuvent accroître la précision des modèles. En l'occurrence, les objets contextuels *bank: Bank* et *atm: ATM* mis en œuvre dans la figure 6.28 posent problème quant à leur identité. Il est possible d'établir ces identités comme suit :

```
   oclInState(Authorisation checking) implies bank.bank code =
     ➥ authorisation.bank codecontext Consortium inv:
   oclInState(Authorisation checking) implies atm.station code = station code
```

Le terme *bank* dans *bank: Bank* désigne donc « la » banque avec qui communiquer, banque parfaitement connue via son code banque, attribut de l'autorisation en cours de contrôle (terme *authorisation*, attribut de l'événement *start of*). De plus, le terme *atm* dans *atm: ATM* est l'*ATM* à qui répondre, en l'occurrence connu par l'attribut *station code* aussi porté par l'événement *start of*. Dans la même logique, on peut écrire :

```
context Consortium inv:
   oclInState(Operation processing) implies bank.bank code =
     ➥ authorisation.bank code
   oclInState(Operation processing) implies atm.station code = station code
```

```
context Consortium inv:
  oclInState(Operation canceling) implies bank.bank code = authorisation.bank code
```

6.5.3 Type d'objet paramétré *Locked list<T>*

L'objet de cette section est de discuter la manière de capitaliser les modèles, c'est-à-dire de les segmenter en vue de les rendre modulaires et génériques (ou plus « ouverts »), au sens où certains d'entre eux doivent à terme ne plus avoir aucune corrélation avec le domaine applicatif. Leur intégration dans la spécification globale se fait alors par des relations et interactions stéréotypées avec les modèles métier.

Dans le chapitre 5, nous avons étudié le package *_Timer*. Toutes les spécifications faisant appel à du cadencement réutilisent ce package. Pour preuve, on l'utilise à la fois dans l'application domotique du chapitre 5 et pour le cas bancaire. L'usage stéréotypé du package est, rappelons-le, le plus souvent et le simplement d'hériter de *Timer client*.

Dans le même esprit, on met ici en œuvre le type *Locked list<T>* du chapitre 3 selon trois formes distinctes et en compétition (figures 6.31, 6.32 et 6.33). Dans la figure 6.31, quatre types d'objet sont créés en héritant de *Locked list<T>* pour obtenir *GonyVA*, *GoaVA*, *GonyPO* et *GoCO*. *T* est remplacé par le « bon type » (*i.e. Authorisation* ou *Operation*). Le reproche est que chaque type n'a qu'une seule instance. La question à se poser est donc : est-ce que chacun des types étend de manière significative les services hérités de *Locked list<T>* ? La réponse est : pas vraiment en fait (voir les automates).

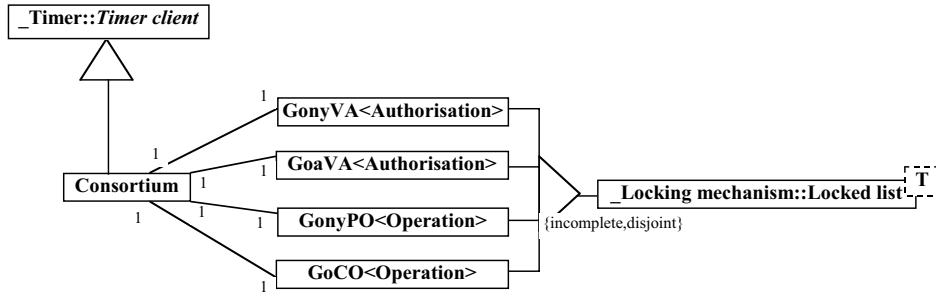


Figure 6.31 – *Class Diagram* complémentaire pour le package *_Consortium* (version 1).

Dans la figure 6.32, avec une notation différente supportée par UML 2.x, on poursuit le même but que dans la figure 6.31 avec un résultat quelque peu différent.

La figure 6.33 est la solution la plus efficace et celle que nous avons retenue. Via un simple modèle d'instances, nous montrons comment une instance anonyme de *Consortium* doit se connecter à quatre instances de *Locked list<T>* avec *T* toujours, remplacé par un type précis.

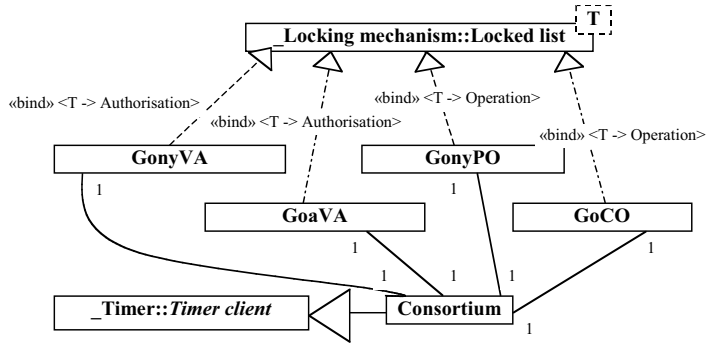


Figure 6.32 – *Class Diagram* complémentaire pour le package *_Consortium* (version 2).

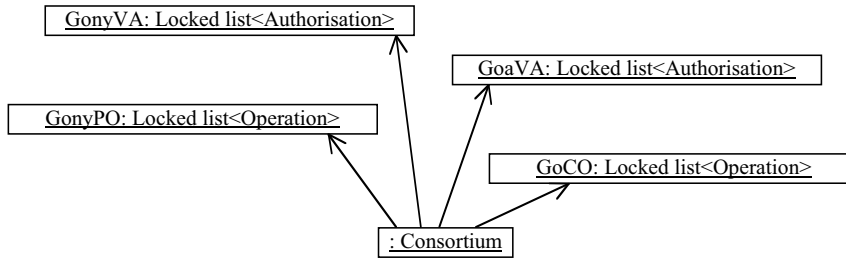


Figure 6.33 – *Class Diagram* complémentaire pour le package *_Consortium* (version 3).

6.6 ÉVOLUTION DU SYSTÈME, MAINTENABILITÉ

Nous y voilà, cette spécification est enfin terminée ! Le coût de sa production est élevé. Il faut donc maintenant prouver que cet investissement est payant du point de vue de la qualité. L'idée est de poser un problème de maintenance.

L'évolution s'inscrit sous deux angles, celui de la maintenance « courante » d'un système ou adaptation à l'évolution des besoins, et celui relatif à l'action de déboguer ou réparation. Nous distinguons ces deux aspects et proposons en conséquence des modifications à la spécification initiale. Nous ne traitons que la prise en charge d'un seul bogue (section « Archivage des transactions sur l'ATM »). L'autre bogue (section « Choix du type d'opération bancaire et du compte ») ainsi que l'amélioration naturelle du système (section 6.6.1) ne sont pas examinés dans ce chapitre. À titre didactique, le lecteur pourra lui-même s'atteler à ces tâches de correction et de gestion de l'évolution des modèles UML jusqu'à présent produits. Au-delà, une fois les modèles mis à jour, le lecteur peut se poser la question de leur répercussion et de l'intervention rapides dans le code. Sans le démontrer, il est évident que les modèles favorisent grandement le cycle de maintenance globale.

6.6.1 Adaptation à l'évolution des besoins

La mise en œuvre et l'exploitation du système ont montré que, d'une part, l'essentiel des opérations bancaires étaient des retraits (79 %) et que, d'autre part, pour 100 retraits demandés, 88 étaient réalisés alors que 12 ne l'étaient pas, dont 9 parce que la somme des retraits effectués par le client est supérieure à la limite hebdomadaire de retrait de la carte. Les banques du consortium ont à l'unanimité choisi de ne plus effectuer ce contrôle et d'en laisser la charge à l'ordinateur central. Les raisons ont été que :

- la charge actuelle de l'ordinateur central permet de lui affecter un travail supplémentaire ;
- un gain appréciable peut être obtenu puisque la communication (opération bancaire à traiter et résultat de traitement d'opération bancaire) entre l'ordinateur central et l'ordinateur de banque sera supprimée dès lors que la somme des retraits effectués par le client sera supérieure à la limite hebdomadaire de retrait de la carte ;
- la charge actuelle de travail de l'ordinateur de banque baissera.

6.6.2 Réparation

Comme tout programme, une spécification est sujette à des bogues (*cf.* les sept péchés capitaux de l'analyse du chapitre 4). Les deux sous-sections qui suivent évoquent chacune une erreur présente dans les modèles UML.

Choix du type d'opération bancaire et du compte

En l'absence de choix par l'utilisateur du type d'opération bancaire (état *Operation kind choice*, figure 6.21) ou du compte sur lequel opérer (état *Account choice*, figure 6.21), le processus se bloque sans issue de sortie. Il n'y a pas de limite temporelle à ces choix. La correction proposée est : au-delà d'une limite à définir, l'ATM assimile automatiquement le type d'opération bancaire à l'interrogation et le compte au compte courant du portefeuille.

Archivage des transactions sur l'ATM

Toute transaction est archivée au moment où elle se termine. L'épuration de la zone de stockage secondaire de l'ATM n'a malencontreusement pas été prévue. Initialement, les archives avaient pour but de tracer l'activité de l'ATM, et en particulier de suivre les tentatives de fraude (nombre, fréquence, etc.). Cependant, à l'usage, les archives ne sont pas exploitées et il se crée un encombrement de la zone de stockage secondaire de l'ATM (l'ATM tombe en panne de manière impromptue). Deux modes d'épuration doivent être mis en œuvre. Le premier, automatique, épure les archives toutes les quarante-huit heures. L'ATM doit bien évidemment être libre, c'est-à-dire ne pas être en mode de fonctionnement avec l'utilisateur et la carte. Le second mode, quant à lui, épure les archives à la reconnaissance d'une carte particulière (numéro de série particulier). La solution apparaît en figure 6.34. Sur la droite

du modèle, figure un attribut supplémentaire dans *ATM*. Sur la gauche, figure un nouvel état avec des transitions idoines dans l'automate global de l'ATM.

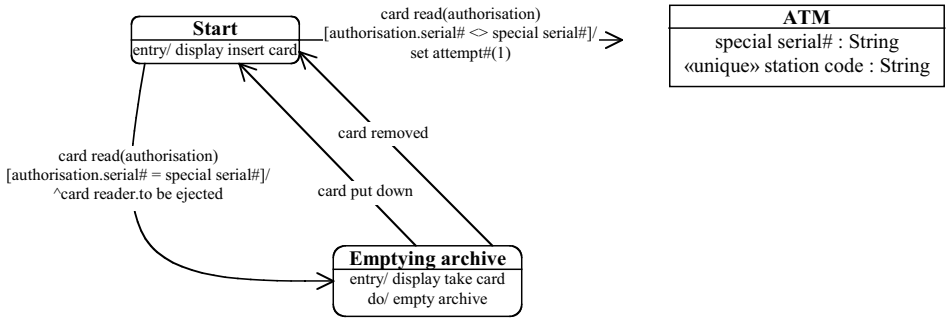


Figure 6.34 – Exemple de réparation des spécifications.

L'activité nouvelle *empty archive* déroulée dans le nouvel état *Emptying archive* est contrainte par la post-condition suivante :

```
context ATM::empty archive()
   post: transaction→isEmpty()
```

6.7 CONCEPTION

Une vision idéaliste du processus de développement logiciel objet est que le modèle de conception est une version augmentée du modèle d'analyse. « Augmentée » signifie qu'il n'y a que des ajouts et idéalement, ce qui n'est malheureusement pas souvent le cas, pas de modifications. Nous avons discuté ce point dans le chapitre 1, mais même si c'est un objectif difficile à atteindre, notre opinion est qu'il faut le rechercher à tout prix car c'est le moyen le plus simple et efficace de créer une bonne traçabilité entre analyse et conception. Au pire, il y a lieu d'éviter que le modèle de conception « casse » le modèle d'analyse pour arriver à une organisation logicielle (architecture) des composants sans rapport intelligible avec la spécification.

Une cause, mineure à notre sens, peut venir des types d'objet parfaitement identifiés en analyse mais n'ayant pas lieu d'être transformés en classes dans la version opérationnelle de la spécification. Ce phénomène n'est pas nouveau : une entité d'un modèle « conceptuel » ne devient pas nécessairement une entité dans un modèle « logique » en implantation relationnelle (voir chapitre 4). La transformation d'un modèle d'analyse en modèle de conception doit être considérée dans le même esprit et quel que soit le support de réalisation : SGBDR, SGBDOO, Java, C++… Un exemple concret est le type d'objet *Time-Date* qui inonde presque tous les modèles de cet ouvrage et dont il est quasiment certain de trouver un équivalent opérationnel dans une quelconque bibliothèque (types prédéfinis *time_t* ou *tm* en C, classe *CTime* en C++/MFC, classes *java.sql.Date*, *java.sql.Time* ou encore *java.sql.-*

Timestamp dans le monde Java, etc.) : voir le chapitre 5 à titre d'illustration (le type Java *GregorianCalendar* est mis en œuvre).

Un autre exemple est le type *Authorisation* dont toutes les propriétés sont dérivées : il peut ou non être matérialisé en tant que classe dans un langage de programmation objet, voire en tant que composant logiciel dans une infrastructure appropriée. De plus, la répartition de l'application en mode client/serveur fait ici que certains types d'objet existent sur le poste client sans réalité sur le poste serveur, et vice-versa. Finalement, tous les modèles qui suivent, parce qu'ils sont justement « de conception », comportent des entités opérationnelles au sens où elles vont exister sous forme de code à un moment ou à un autre.

Au-delà, la réutilisation de services préimplémentés dans un système d'exploitation ou tout autre outil doit aboutir à « en faire le minimum » : c'est la vraie réutilisation, celle rêvée ! Le cas du type d'objet *Consortium* dont le comportement est spécifié à l'aide d'*Activity Diagram* et de *State Machine Diagram* plutôt compliqués (figure 6.27, figure 6.28, figure 6.29 et figure 6.30) est révélateur : à l'aide du SGBD Oracle par exemple, tous les mécanismes de verrouillage, déverrouillage ou encore synchronisation exigés par la manipulation des types *GonyVA, GoaVA, GonyPO* et *GoCO* qui deviennent des tables, sont directement activables via du code SQL embarqué dans le code Java. En bilan, la classe Java *Consortium* côté client (voir parties de code ci-dessous) se résume à peu et ne nécessite pas la bibliothèque *PauWare.Statecharts* comme le nécessite *ATM* par exemple.

6.7.1 Implantation base de données, partie serveur

La base de données du système bancaire, correspondant grossièrement au package *_Consortium* et située côté serveur, est présentée en figure 6.35 à l'aide du SGBDR Access[1] (l'implantation Oracle figure, elle, en fin de ce chapitre).

Nous ne revenons pas sur la méthodologie d'obtention de schémas relationnels à partir de *Class Diagram*, abondamment illustrée dans le chapitre 4. En revanche, il est à noter que les dépendances fonctionnelles entre les tables *GonyVA* et *GoaVA* d'une part, avec les tables *ATM* et *Plastic_card* d'autre part, ont pour vocation de renforcer la sécurité du système. En fait, du point de vue de la sécurité, les transactions en cours validées (*i.e.* présentes dans *GoaVA*) ou non encore acceptées par la banque concernée (*i.e.* présentes dans *GonyVA*) sont identifiées par l'*ATM* sur lequel elles ont été ouvertes et par l'autorisation figurant sur la carte plastique insérée. En traduction relationnelle, les champs *station_code* et *serial_number* sont deux clés étrangères dans les tables *GonyVA* et *GoaVA* référençant respectivement la table *ATM* et la table *Plastic_card*. Les contraintes d'intégrité référentielle dépeintes en figure 6.35 assurent donc que des transactions ne peuvent pas être ouvertes sur des *ATM* ou avec des cartes plastiques, tous deux imaginaires. Dans la même logique, les opérations bancaires demandées (*GonyPO*) ou annulées (*GoCO*) font réfé-

1. Attention à l'entité *GoaVA_1* en figure 6.35 qui n'est pas une table mais une notation propre à Access, notation engendrée par la présence de deux clés étrangères sur le même champ.

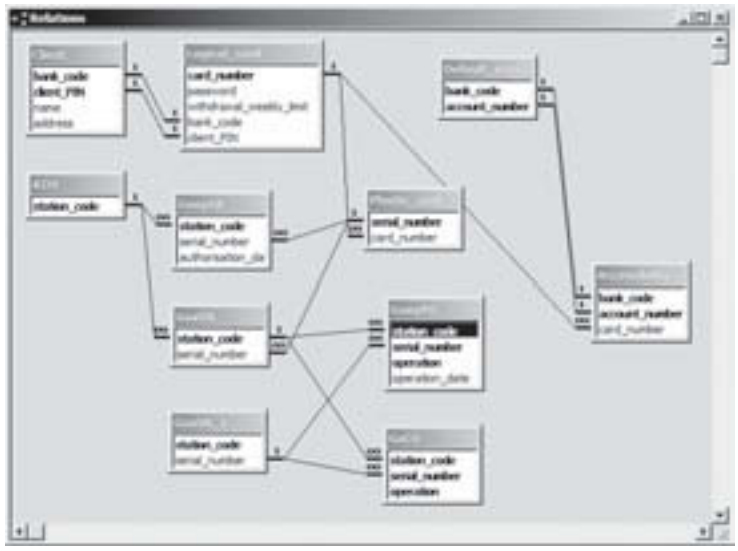


Figure 6.35 – Implantation relationnelle du package *_Consortium*.

rence à la transaction à laquelle elles appartiennent de sorte que des opérations illicites ne soient pas enregistrables dans la base de données.

Les autres tables sont du point de vue des création, mise à jour et suppression de données, gérées hors de l'application proprement dite que nous avons spécifiée. En pratique, le traitement d'une transaction se borne donc à la lecture des seules tables *Client*, *Logical_card*, *Plastic_card*, *Accessibility* et *Default_account*.

6.7.2 Partie client

Le logiciel tournant sur l'ATM va faire apparaître deux types de connectivité. La première est celle avec la partie serveur. Son design est fait à partir de JDBC. La seconde est celle avec l'environnement (incluant l'interface utilisateur) et plus spécifiquement ici avec des périphériques physiques comme le lecteur de carte. D'un point de vue logiciel, nous proposons ici une méthode pragmatique et simple pour intégrer des pilotes de ces périphériques de manière souple et surtout, sans déstabilisation de l'architecture logicielle (figure 6.36) que nous avons dérivée des modèles d'analyse.

L'architecture en figure 6.36 donne les classes Java, la façon dont elles se référencent (la direction des associations est déterminée par les flèches), la visibilité qui est le plus souvent *private* (- en UML) et finalement des classes techniques comme *java.sql.Connection* (*java::sql::Connection* en notation UML dans la figure) qui crée le lien avec JDBC en l'occurrence.

La comparaison de la figure 6.36 avec la figure 6.25 (modèle d'analyse) et la figure 6.35 (modèle de conception côté serveur) montre :

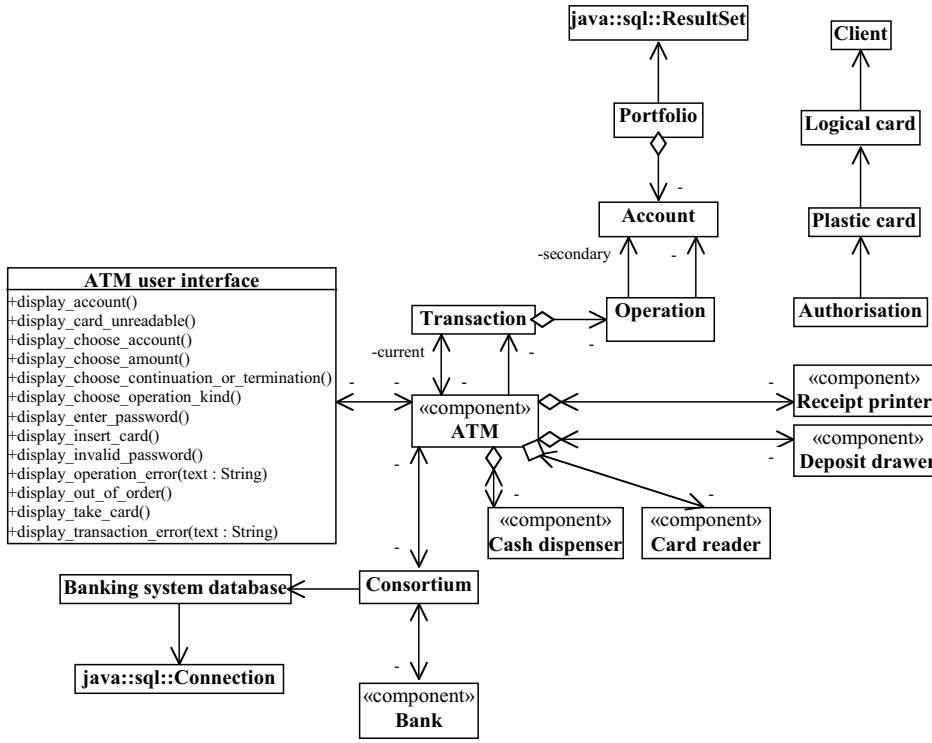


Figure 6.36 – Architecture logicielle côté client (modèle incomplet).

- l'absence de liaison entre *Logical card* et *Account* dans la figure 6.36 qui signifie que les informations magnétisées sur la carte plastique et manipulées dans l'application côté client, selon les navigations prévues entre *Authorisation*, *Plastic card*, *Logical card* et *Client* (en haut, à droite de la figure 6.36), ne permettent pas de connaître les comptes bancaires accessibles par une carte plastique. C'est tout à fait sain car cette accessibilité, évolutive s'il en est (création de nouveaux comptes…), est connue côté serveur (figure 6.35). À chaque transaction donc, le portefeuille d'un client est calculé dynamiquement (*snapshot* de la base de données). C'est pour cette raison que le type *Portfolio* met en œuvre le type JDBC *ResultSet* tout en haut du modèle de la figure 6.36 ;
- l'orientation différente des relations entre la partie serveur et la partie client qui est naturelle : elle reflète les usages des objets (navigations) qui peuvent être différents côté serveur et côté client. Par ailleurs, les directions retenues sont majoritairement imposées par les flux d'envoi des événements dans les *State Machine Diagram* ;
- l'introduction de types d'objet propres à l'application cliente comme *ATM user interface*. Notons que le modèle de la figure 6.36 est pour l'instant incomplet, nous le complétons progressivement dans la discussion qui suit.

Lien avec la base de données

La figure 6.36 montre une association bidirectionnelle entre *ATM* et *Consortium*. L'ouverture de transaction par l'émission de l'événement *start of* (figure 6.22) va s'effectuer dans la méthode *password_entered* d'*ATM*. L'utilisation comme au chapitre 5 de la bibliothèque *PauWare.Statecharts* va grandement faciliter toute la gestion de l'automate, des gardes… comme cela apparaît dans le code qui suit :

```
public void password_entered(String password) throws Statechart_exception {
   boolean guard = password.equals(_authorisation.password());
   Object[] args = new Object[3];
   args[0] = _station_code;
   args[1] = _authorisation;
   args[2] = new
java.sql.Time(java.util.GregorianCalendar.getInstance().getTimeInMillis());
   // l'appel de start_of s'opère par réflexion dans PauWare.Statecharts
   // d'où la préparation préalable du tableau args avant configuration
   // de la transition :
   _ATM.fires(_Password_request,_Transaction_verification,guard,_consortium,"
     ➥ start_of",args,Statechart_monitor.Reentrance);
   _ATM.fires(_Password_request,_Second_delaying,! guard,this,
     ➥ "display_password_invalid",null);
   args = new Object[2];
   args[0] = _Password_request;
   args[1] = new Long(15000);
   _ATM.fires(_Password_request,_Second_delaying,! guard,this,"to_be_set",args);
   _ATM.run_to_completion();
}
```

La référence *_consortium*[1] matérialise donc le lien d'*ATM* vers *Consortium* de la figure 6.36. L'option *Statechart_monitor.Reentrance* lors de l'envoi de *start of* est contingent à la réponse de *_consortium* (envoi des événements *response to -start of- OK* ou au contraire *response to -start of- not OK*). Cette option permet de ré-entrer dans l'instance d'*ATM* émettrice sans bouleverser l'automate, *i.e.* l'interprétation de *response to -start of- OK* ou de *response to -start of- not OK*) a nécessairement lieu *après* l'événement en cours de traitement, ici *password entered*. Pour information, l'option *Statechart_monitor.Reentrance* est inutile lorsque le receveur ne répond pas, cela est en particulier moins coûteux car la réentrance met en œuvre le *multithreading*.

La classe Java *Consortium* implantée côté client va donc assurer le lien avec la partie serveur. La structure partielle de cette classe est décrite ci-après ainsi que la méthode publique *start_of* en particulier, qui repose sur JDBC.

```
import java.sql.*;
…
protected Connection _database;
protected PreparedStatement _start_of;
…
public Consortium(ATM atm) throws SQLException {
_atm = atm;
```

1. Cet objet réside à la fois côté serveur et côté client mais avec des formes différentes. Côté client, il est une sorte de frontal avec toute la partie serveur.

```
_database = Banking_system_database.Database();
_start_of = _database.prepareStatement("INSERT INTO GonyVA VALUES(?,?,?)");
_to_be_processed = _database.prepareStatement("INSERT INTO
     ➥ GonyPO VALUES(?,?,?,?)");
_to_be_canceled = _database.prepareStatement("INSERT INTO GoCO VALUES(?,?,?)");
_end_of = _database.prepareStatement("DELETE FROM GoaVA WHERE station_code =
     ➥ ? AND serial_number = ?");
}
…
public void start_of(String station_code,Authorisation authorisation,Time now)
throws Statechart_exception {
   try {
      if(_database.isClosed()) response_to_start_of_not_ok(new
     ➥ Response_to_start_of_not_ok_event(this,"Fatal error (start_of):
     ➥ JDBC connection abnormally closed"));
      if(Banking_system_database.Oracle()) _database.createStatement().
     ➥ execute("LOCK TABLE GonyVA IN EXCLUSIVE MODE");
      _start_of.setString(1,station_code);
      _start_of.setString(2,authorisation.unique());
      _start_of.setTime(3,now);
      _start_of.execute();
      if(_start_of.getWarnings() == null) {
      if(_database.getTransactionIsolation() != Connection.TRANSACTION_NONE)
       _database.commit();
      if(_bank.get(authorisation.bank_code()) == null)
     ➥ _bank.put(authorisation.bank_code(),new
     ➥ Bank(authorisation.bank_code(),this));
      ((Bank)_bank.get(authorisation.bank_code())).authorisation_to_be_checked(
     ➥ station_code,authorisation);
      }
      else {
         if(_database.getTransactionIsolation() != Connection.
     ➥ TRANSACTION_NONE) _database.rollback();
         response_to_start_of_not_ok(new Response_to_start_of_not_ok_event(
     ➥ this,"Error (start_of): JDBC failure"));
      }
   }
   catch(SQLException sqle) {
      System.err.println(sqle.getClass().toString() + ": " + sqle.getMessage());
      response_to_start_of_not_ok(new Response_to_start_of_not_ok_event(
     ➥ this,"Fatal error (start_of): " + sqle.getMessage()));
   }
}
```

On remarque, dans le code qui précède, la liaison avec un objet _bank de type collection de Bank (cardinalité * côté Bank dans la figure 6.25) pour établir la décision d'ouverture ou non de transaction après routage à « la » banque appropriée (code banque de l'objet authorisation). En cas de problème technique lié à JDBC (interruption de connexion, requêtes échouant…), la réponse d'ouverture de transaction est automatiquement négative avec le motif de l'échec de type String. Le type Bank est vu comme un composant logiciel (stéréotype «component») en figure 6.36. Ses instances peuvent être des EJB ou tout autre type de composant.

Connectivité avec l'environnement, attribution du stéréotype «component»

Dans la figure 6.36, certains objets comme *ATM* se sont vus attribuer le stéréotype *«component»* alors que d'autres non (*e.g. Transaction*) ce qui fait dire en UML que ces derniers sont des classes (stéréotype *«class»* par défaut). Cette attribution *«component»* signifie unité de déploiement indépendante (définition du chapitre 1) au sens où il doit être possible (mais non obligatoire) de localiser/relier dynamiquement les instances des composants logiciels via par exemple, un mécanisme comme JNDI dans les EJB (voir chapitre 4 pour cette technique). Ce qui nous intéresse plutôt ici, c'est la méthode de fabrication des composants puis la définition de leur connectivité avec l'environnement, ce qui établit bien entendu leurs interfaces.

Soit ainsi un automate d'un type d'objet *T* avec $E_T = \{e_1, e_2 \ldots e_n\}$[1] l'ensemble des événements qu'il reçoit. Il existe une partition de E_T dont chaque partie devient une interface. Cela donne le modèle de la figure 6.37 où deux vues alternatives sont proposées. Dans la figure 6.37, en bas, à gauche, la présence d'un compartiment débutant par le stéréotype *«realizations»* fait que l'on parle de représentation « boîte blanche » : on donne l'entité logicielle qui implémente le composant logiciel. La partie droite de la figure 6.37 est plus classique et utilise l'ancienne notation UML (trait avec boule blanche à l'extrémité, interface fournie) doublée de la nouvelle d'UML 2.x (trait avec demi-cercle à l'extrémité, interface requise).

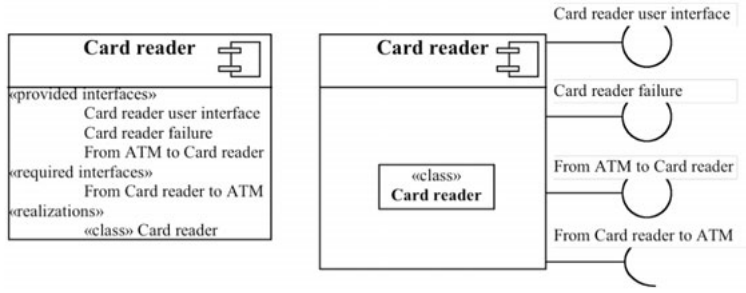


Figure 6.37 – Deux représentations en compétition du composant logiciel *Card reader*.

Il y a trois interfaces fournies qui sont les événements reçus par *Card reader* à l'exception de *time-out* pris en charge par le fait que *Card reader* hérite de *Timer client* (figure 6.7). La seule interface requise est la liaison avec *ATM* (*From Card reader to ATM*) pour les acquittements/notifications enchaînant les requêtes faites par *ATM*. Une vision expansée du *Component Diagram* de la figure 6.37 est en figure 6.38. Tous les services mis en œuvre apparaissent explicitement avec leur signature et en totale cohérence avec l'automate de la figure 6.15.

Les actions, activités, gardes et autres fonctions spéciales propres aux considérations d'implémentation sont établies dans la classe Java qui sert à implémenter le

1. On effectue une différentiation uniquement syntaxique entre les événements (*i.e.* leur nom) ce qui peut être insuffisant : deux types d'événements, bien que de même nom, peuvent différer par le type de leur(s) attribut(s) et/ou par leur provenance.

composant logiciel *Card reader* (stéréotype *«implement»* en *figure* 6.38 ou inclusion topologique en figure 6.37 correspondant au stéréotype *«resides»* d'UML 1.x.). Attention donc dans la figure 6.38, à faire la distinction entre la classe d'implémentation (stéréotype *«class»*) et le composant logiciel proprement dit. On remarque que les actions/activités sont étrangement déclarées publiques (+ en UML). Cela est dû à la réflexion Java qui impose ce type de visibilité. Sinon, par simple agrément, nous montrons comment les états peuvent être associés à des fonctions d'accès, privées ici en l'occurrence (- en UML) : voir aussi la classe *In*, vue au chapitre 5, de la bibliothèque *PauWare.Statecharts*.

Pour terminer, remarquons à quel point l'implémentation peut être détaillée. Dans la figure 6.38, la façon dont la classe Java *_PauWare::Statechart_monitor* est réutilisée est explicitée par l'association orientée partant de la classe *Card reader* et avec la visibilité *protected* (# en UML).

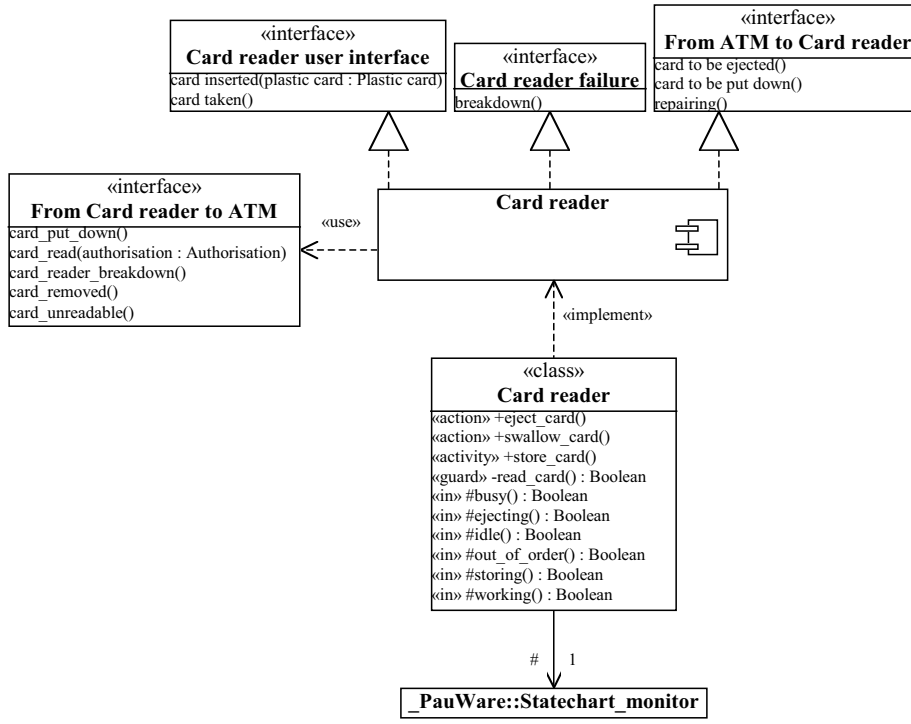


Figure 6.38 – Représentation détaillée du modèle de la figure 6.37.

La partition des différentes interfaces est empirique et s'appuie sur la provenance des événements : trois flux d'entrée pour *Card reader*, les événements générés par l'utilisateur comme « carte retirée du lecteur » (*card taken*), « panne » (*breakdown*) et le flux contrôle émanant de l'ATM comme l'ordre d'éjection de carte par exemple (*card to be ejected*).

En adoptant la même procédure pour les autres composants de la figure 6.36, on peut décrire des assemblages comme dans la figure 6.39.

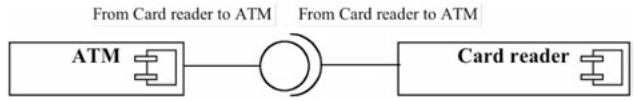


Figure 6.39 – Assemblage entre *ATM* et *Card reader*.

Bien qu'élégant et clair, ce genre de modèle est plus un complément de documentation que de la véritable spécification. Pour s'en convaincre, il suffit de jeter un coup d'œil à l'automate d'*ATM* revu et corrigé en figure 6.40. Les dispositifs vus jusqu'à présent comme externes sont reliés par une relation de composition (losange noir) au lieu de l'agrégation initiale (losange blanc). De plus, ils sont vus comme des sous-états parallèles dans l'esprit de la discussion du chapitre 3 sur l'exécutabilité. Pourquoi pas ? En conception, ces dispositifs ne peuvent donc plus être vus comme des unités de déploiement indépendantes et donc des composants logiciels, d'où des différences importantes dans les *Component Diagram* avec l'une ou l'autre des solutions. Il est donc fondamental de comprendre que les choix faits dans les *State Machine Diagram* régissent de façon stricte la manière de concevoir.

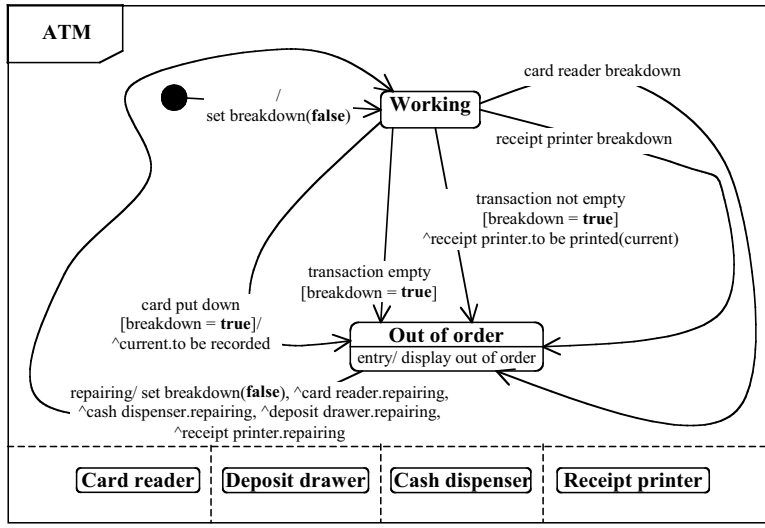


Figure 6.40 – Refonte de la spécification d'*ATM*.

L'idée phare de la figure 6.40 est que le type d'objet *Card reader* par exemple, parfaitement identifié en analyse, va être fondu dans le composant logiciel *ATM* et ne devient donc pas lui-même un composant logiciel. En conception, il garde son caractère de type d'objet ou plus précisément celui de *Classifier* implémentant le

composant logiciel *ATM* mais *n'interagissant pas* avec lui comme montré en figure 6.39.

Détails d'implémentation

Les types d'objet ayant un identifiant (stéréotype *«unique»*) doivent préférablement implémenter une interface Java appelée *Unique* prédéfinie pour formaliser l'égalité de deux instances et avoir un accès à l'identifiant (méthode *unique()* ci-dessous). Voici l'exemple du type d'objet *Logical card* :

```
public class Logical_card implements Unique {
   /** UML attributes */
   protected final String _card_number;
   public final String password;
   public final int withdrawal_weekly_limit;
   /** UML associations */
   protected final Client _client;
   public boolean equals(Object object) {
   if(object instanceof Logical_card) return
     (_card_number.equals(((Logical_card)object)._card_number));
   return false;
   }
   public String unique() {
   return _card_number;
   }
   …
}
```

Pour aller plus loin dans les détails d'implémentation, UML 2.x propose les *Composite Structure Diagram* avec les notions de *Port* et de *Connector* (deux types : *Assembly connector* et *Delegation connector*). Le petit carré blanc sur le contour de l'instance de composant logiciel *ATM* en figure 6.41 matérialise un *Port*. Le *Component Diagram* de la figure 6.41 est basé « instance ». Attention en effet, chaque élément du modèle (tous en l'occurrence instances de *Classifier*) est préfixé d'un *:*. En UML 2.x, ce formalisme est celui des *Composite Structure Diagram* qui s'entrelace donc avec celui des *Component Diagram* et a vocation d'unifier, en général, toutes les représentations fondées sur la contenance impliquant donc des objets conteneurs (*:ATM* dans l'exemple) et des objets contenus (*:Card reader* dans l'exemple). En figure 6.41, la flèche partant du *provided port* à la base de la *provided interface* nommée *Card reader user interface* et allant vers l'instance de *Classifier* appelée *:Card reader* correspondant au type d'objet *Card reader* des modèles d'analyse, est un *Delegation connector*.

Ce que nous cherchons à faire dans la figure 6.41 est au pire contraire aux modèles de la figure 6.37 et de la figure 6.38 et au mieux incongru : ce n'est pas une instance d'*ATM* qui doit traiter les messages *card inserted* et *card taken*, seuls constituants de l'interface *Card reader user interface*. En fait, l'application Java que nous avons implémentée simule l'occurrence de ces deux messages depuis l'interface utilisateur de l'ATM (type *ATM user interface* en figure 6.36). Nous réceptionnons donc ces messages dans *ATM* qui doit les déléguer (stéréotype *«delegate»* en figure

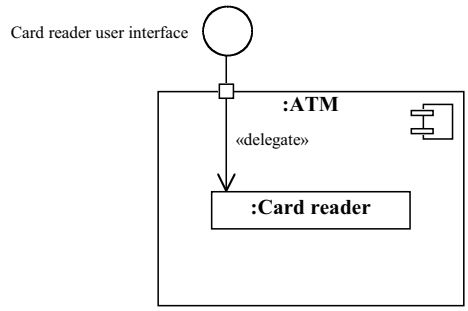


Figure 6.41 – Instance de composant logiciel *ATM* (représentation partielle de type *Composite Structure Diagram*).

6.41) au composant logiciel (ou classe)[1] référencé dans *ATM*. Insistons bien sur le fait que cette adaptation ne résulte que de la réalisation d'un simulateur en lieu et place d'un distributeur automatique bancaire « réel ».

Il subsiste bon nombre d'ambiguïtés dans la documentation UML 2.x quant à la notation des *Composite Structure Diagram* en général. À ce titre, le schéma de la figure 6.42 est-il correct ? Nous ne tranchons pas sur la question du fait des schémas à interprétation multiple de la documentation (figure 89) [4, p. 142], qui n'aident pas dans cette réflexion.

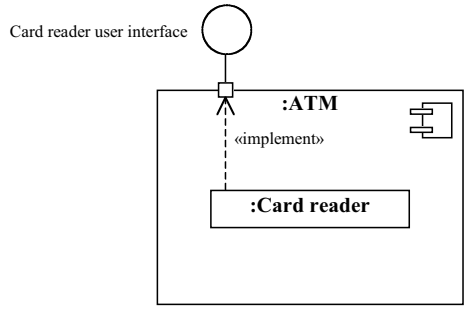


Figure 6.42 – Variante de la figure 6.41.

6.8 CONCLUSION

La valeur démonstrative de cette étude de cas quant à la facilité/pénibilité d'usage d'UML, est qu'elle mélange, en compliquant, aspects statiques (prédominants dans l'étude de cas du chapitre 4) et aspects dynamiques (prédominants dans l'étude de cas du chapitre 5). En outre, son caractère client/serveur et son côté éternellement didactique en font une étude de cas étalon.

1. Peu importe ici car tout est dénaturé.

D'un point de vue technique, la mise en œuvre des *Activity Diagram*, en compétition en fait des *State Machine Diagram*, contribue à la réflexion sur l'intérêt d'un multiformalisme. En effet, créer la nécessité d'une compétence plus grande pour tout comprendre n'est pas judicieux. La complexité intrinsèque de l'application est déjà suffisamment importante pour que l'on n'ait pas parallèlement à souffrir de celle de « l'outil UML ». Encore une fois et c'est notre vision générale, mieux vaut maîtriser peu de constructions de modélisation mais dont on a stabilisé et personnalisé la sémantique que de s'éparpiller sur des modèles à interprétation multiple, par les autres acteurs du développement en particulier.

D'un point de vue industriel, avec la probabilité forte d'avoir à faire à des applications de plus grande échelle, peut-on conclure qu'UML est inabordable, ou alors, avec une volonté farouche, un effort maximal et une implication sans faille, ce standard de modélisation peut-il raisonnablement être profitable à ses propres projets informatiques ? Si la question est difficile, elle est tout simplement parfois vaine car UML est imposé : par le client, par une norme, par un chef de projet qui veut via cet outil contrôler son projet. Par ailleurs, désabusé voire même rebuté par la complexité de mise en œuvre d'UML, le rationalisme et/ou la sagesse ne sont-ils pas de se dire que sans UML, ce serait pire ? À vérifier sur le terrain.

6.9 CODE ORACLE DE L'ÉTUDE DE CAS

```
create schema authorization barbier

create table ATM(station_code char(10),
   constraint ATM_key primary key(station_code))

create table Client(bank_code char(10),
   client_PIN char(10),
   name char(100),
   address char(100),
   constraint Client_key primary key(bank_code,client_PIN))

create table Logical_card(card_number char(10),
   password char(4),
   withdrawal_weekly_limit number(5),
   bank_code char(10),
   client_PIN char(10),
   constraint Logical_card_key primary key(card_number),
   constraint Client_foreign_key foreign key(bank_code,client_PIN) references
      ➥ Client(bank_code,client_PIN) on delete cascade)

create table Plastic_card(serial_number char(10),
   card_number char(10),
   constraint Plastic_card_key primary key(serial_number),
   constraint Logical_card_foreign_key foreign key(card_number) references
      ➥ Logical_card(card_number) on delete cascade)

create table Accessibility(bank_code char(10),
```

```
   account_number char(10),
   card_number char(10),
   constraint Accessibility_key primary key(bank_code,account_number),
   constraint Logical_card_foreign_key2 foreign key(card_number) references
      ➥ Logical_card(card_number) on delete cascade)

create table Default_account(bank_code char(10),
   account_number char(10),
   constraint Default_account_key primary key(bank_code,account_number),
   constraint Accessibility_foreign_key foreign key(bank_code,account_number)
      ➥ references Accessibility(bank_code,account_number) on delete cascade)

create table GonyVA(station_code char(10),
   serial_number char(10),
   authorisation_date date,
   constraint GonyVA_key primary key(station_code),
   constraint GonyVA_unique unique(serial_number),
   constraint ATM_foreign_key foreign key(station_code) references
      ➥ ATM(station_code),
   constraint Plastic_card_foreign_key foreign key(serial_number) references
      ➥ Plastic_card(serial_number))

create table GoaVA(station_code char(10),
   serial_number char(10),
   constraint GoaVA_key primary key(station_code),
   constraint GoaVA_unique unique(serial_number),
   constraint ATM_foreign_key2 foreign key(station_code) references
      ➥ ATM(station_code),
   constraint Plastic_card_foreign_key2 foreign key(serial_number) references
      ➥ Plastic_card(serial_number))

create table GonyPO(station_code char(10),
   serial_number char(10),
   operation char(10),
   operation_date date,
   constraint GonyPO_key primary key(station_code,serial_number,operation),
   constraint GoaVA_foreign_key foreign key(station_code) references
      ➥ GoaVA(station_code),
   constraint GoaVA_foreign_key2 foreign key(serial_number) references
      ➥ GoaVA(serial_number))

create table GoCO(station_code char(10),
   serial_number char(10),
   operation char(10),
   constraint GoCO_key primary key(station_code,serial_number,operation),
   constraint GoaVA_foreign_key3 foreign key(station_code) references
      ➥ GoaVA(station_code),
   constraint GoaVA_foreign_key4 foreign key(serial_number) references
      ➥ GoaVA(serial_number));
```

6.10 BIBLIOGRAPHIE

1. André, P., Barbier, F., and Royer, J.-C. : « Une expérimentation de développement formel à objets », *Technique et Science Informatiques*, 14(8), (1995) 973-1005
2. Cook, S., and Daniels, J. : *Designing Object Systems – Object-Oriented Modelling with Syntropy*, Prentice Hall (1994)
3. Harel, D. : "From Play-In Scenarios to Code: An Achievable Dream", *IEEE Computer*, 34(1), (2001) 53-60
4. Object Management Group : *UML 2.0 Superstructure Specification* (2003)
5. Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., and Lorensen, W. : *Object-Oriented Modeling and Design*, Prentice Hall (1991)

6.11 WEBOGRAPHIE

Sources de l'étude de cas : www.PauWare.com

Index

D

E

F

G

H

I

M

N

O

P

Q

R

S

T

049526 - (I) - (1,5) - OSB 100° - AUT - ABS

Achevé d'imprimer sur les presses de
SNEL Grafics sa
rue Saint-Vincent 12 – B-4020 Liège
Tél +32(0)4 344 65 60 - Fax +32(0)4 341 48 41
novembre 2005 — 35938

Dépôt légal : novembre 2005

Imprimé en Belgique

INFOPRO

TYPE D'OUVRAGE

L'ESSENTIEL	SE FORMER	RETOURS D'EXPÉRIENCE

MANAGEMENT DES SYSTÈMES D'INFORMATION

APPLICATIONS MÉTIERS

ÉTUDES, DÉVELOPPEMENT, INTÉGRATION

EXPLOITATION ET ADMINISTRATION

RÉSEAUX & TÉLÉCOMS

Franck Barbier

UML 2 ET MDE

Ingénierie des modèles avec études de cas

Ce livre s'adresse aux ingénieurs logiciel, développeurs, architectes et chefs de projet ainsi qu'aux étudiants d'écoles d'ingénieurs et masters informatiques.

Il traite du nouveau paradigme informatique MDE (*Model-Driven Engineering*) ou « ingénierie des modèles » qui est intimement lié au standard international UML 2 (*Unified Modeling Language*).

- La première partie revient sur la technologie des objets en général, et notamment sur le lien entre la modélisation orientée objet et la programmation orientée objet. Elle est illustrée de nombreux exemples de code et d'une étude de cas concise en C++.
- La deuxième partie est une présentation approfondie d'UML 2, et notamment de toutes ses différences et ses avancées en regard d'UML 1.x.
- La dernière partie comporte trois études de cas implantées en totalité (les modèles UML sont fournis de manière exhaustive), leur code est téléchargeable.

FRANCK BARBIER
est professeur des universités, conseiller scientifique de Reich Technologies (devenue Projexion Netsoft), l'une des 17 sociétés qui ont créé UML à l'OMG en 1997. Il est aussi coauteur des documents du consortium DSTC soumis à l'OMG pour la création d'UML 2.0.

9 782100 495269

ISBN 2 10 049526 7

www.dunod.com

DUNOD