Modélisation objet

avec UML

Pierre-Alain Muller

Préface

UML est le résultat d'un long processus initialisé depuis maintenant deux ans par trois des méthodologistes les plus réputés : Grady Booch, Ivar Jacobson et Jim Rumbaugh. Conscients que leurs travaux respectifs ne représentaient que les diverses facettes d'un même problème, ils ont eu l'intelligence de joindre leurs efforts plutôt que de se lancer dans d'inutiles querelles de chapelles. Deux ans de travail au sein de Rational Software Corporation, soutenus dans leurs efforts par de nombreux autres spécialistes du domaine, leur ont permis de définir cette nouvelle approche de la modélisation des logiciels à base d'objets. UML est à la fois la synthèse et le successeur naturel de leurs différents travaux. De par sa standardisation par l'OMG, qui devrait aboutir dans le courant de l'année 1997, et de par son adoption par les plus grands acteurs du monde de l'informatique : IBM, Microsoft, Oracle, Hewlett Packard, pour ne citer que les plus grands, UML est appelé à devenir à très court terme le standard pour la modélisation des applications informatiques de demain.

Pierre Alain Muller, pour avoir passé cinq ans à Rational Software en contact direct avec les gourous du domaine, était la personne la plus à même de rédiger ce premier ouvrage en français sur UML. Ses compétences indiscutables dans le domaine, alliées à son sens profond de la pédagogie, font de cet ouvrage un guide indispensable à tous ceux qui veulent comprendre et appliquer les principes d'UML dans le développement de leurs applications.

Etienne Morel

Directeur Général

Rational Software Europe

Redde Caesari quae sunt Caesaris

Les principaux auteurs de la notation UML sont Grady Booch, Ivar Jacobson et Jim Rumbaugh. Cet ouvrage fait constamment référence à leurs travaux et je leur rends bien volontiers ce qui leur appartient.

Au-delà de la notation UML, cet ouvrage décrit un processus de pilotage des projets objet qui est largement inspiré du processus de développement mis au point par les consultants de Rational Software Corporation. Qu'ils soient ici tous remerciés, en particulier Philippe Kruchten.

Je remercie également les personnes qui ont accepté de relire le manuscrit et qui m'ont apporté leurs précieux commentaires ; je pense surtout à Jean-Luc Adda, Jean Bézivin, Jérôme Desquilbet, Nathalie Gaertner, Yves Holvoët, Nasser Kettani, Mireille Muller, Philippe Perrin, Michel Reyrolle, Marie-Christine Roch et Philippe Studer.

Je n'oublie pas toutes les personnes qui ont influencé la rédaction de cet ouvrage, tout particulièrement Etienne Morel et Bernard Thirion, mais aussi Jean-Jacques Bockstaller, Tom Joad, Philippe Laroque, Gérard Metzger, Bernard Monnoye et les quelques centaines de personnes qui ont assisté à mes formations sur l'objet. Je ne remercie pas Dominique Griesinger.

Enfin, je remercie Anne, Jonathan, Roxane et Lara pour leur patience et je leur dédie cet ouvrage.

pam, avril 1997

Table des matières

A qui s’adresse ce livre

Ce livre traite de modélisation objet. Il décrit la mise en œuvre de la notation UML (*Unified Modeling Language*), développée en réponse à l’appel à propositions lancé par l’OMG (*Object Management Group*), dans le but de définir la notation standard pour la modélisation des applications construites à l’aide d’objets.

La notation UML représente l’état de l’art des langages de modélisation objet. Elle se place comme le successeur naturel des notations des méthodes de Booch, OMT (*Object Modeling Technique*) et OOSE (*Object Oriented Software Engineering*) et, de ce fait, UML s’est très rapidement imposée, à la fois auprès des utilisateurs et sur le terrain de la normalisation.

Ce livre s'adresse à tous ceux qui désirent comprendre et mettre en œuvre l'approche objet, et plus particulièrement aux professionnels de l'informatique qui doivent aborder la transition vers UML. La lecture de l'ouvrage ne demande pas de connaissances préalables en technologie objet, mais suppose des connaissances informatiques de base. Le livre peut servir d'accompagnement à un cours de second cycle ou d'année de spécialisation d'école d'ingénieur.

Le contenu de ce livre, le cheminement proposé et le niveau d’abstraction retenu dans la présentation de la notation UML, sont le fruit d’une pratique des méthodes objet, dans des projets réels. La rédaction insiste tout particulièrement sur la modélisation objet, c’est-à-dire sur l’analyse et la définition des besoins de l’utilisateur ; ceci non pas, parce que la conception ou la programmation seraient des tâches moins nobles, mais tout simplement, parce que les informaticiens ont beaucoup plus de mal à trouver ce qu’il faut faire, qu’à trouver comment le faire.

L’ouvrage est divisé en 5 chapitres qui peuvent être lus de manière quasi indépendante :

- le premier chapitre met en avant le besoin de méthodes et décrit la genèse de la notation unifiée UML,
- le deuxième chapitre présente les concepts de base de la technologie objet, afin de rendre la lecture de l'ouvrage accessible aux néophytes,
- le troisième chapitre décrit les concepts d'UML, en utilisant la notation comme support pour la description de la sémantique des éléments de modélisation,
- le quatrième chapitre introduit les bases de l'encadrement des projets objet, avec une description du processus de développement sous-tendu par UML (piloté par les *use cases*, centré sur l'architecture, itératif et incrémental),
- le cinquième chapitre présente une étude de cas : la modélisation objet avec UML d'un système de contrôle des accès à un bâtiment.

Le livre se termine par des annexes sur la transition vers UML et sur des règles de mise en œuvre, organisées de la manière suivante :

- un guide de transition de Booch et OMT vers UML,
- la génération de code C++,
- la génération de code Java,
- la génération de code IDL,
- la génération de code Visual Basic,
- la génération de code SQL.

Plusieurs lectures de l'ouvrage sont possibles, selon les connaissances et les centres d'intérêts du lecteur :

- Le lecteur novice est invité à lire l'ensemble du livre, en suivant l'ordre des chapitres. Ce conseil s'applique également aux programmeurs qui ont acquis une connaissance d'un langage de programmation objet comme C++, sans avoir suivi de formation spécifique à l'analyse et à la conception objet.
- Le lecteur qui possède déjà une bonne connaissance d'une méthode de modélisation, telle que Booch ou OMT, pourra entamer la lecture à partir du troisième chapitre. Il lui est par contre recommandé de parcourir également le deuxième chapitre, car l'expérience montre que certains concepts comme celui de *généralisation* sont souvent mal compris.
- Les architectes du logiciel, après avoir étudié la notation UML dans le troisième chapitre, se concentreront tout particulièrement sur le quatrième chapitre qui traite d'architecture objet et présente le modèle des 4 + 1 vues.
- Les chefs de projets trouveront également dans le quatrième chapitre les informations nécessaires pour la mise en place d'un processus de développement piloté par les cas d'utilisation (*use cases*), centré sur

l'architecture, itératif et incrémental. Il leur sera aussi profitable de parcourir les premiers chapitres, afin de pouvoir lire et comprendre les modèles.

Il est probablement utile de dire aussi que ce livre n'est :

- ni un traité de programmation objet ; il ne décrit aucun langage de programmation en détail,
- ni une apologie d'UML ; mais plutôt un ouvrage de modélisation objet, qui faut appel à la notation UML.

1

La genèse d'UML

L'informatique s'est glissée imperceptiblement dans notre vie quotidienne. Des machines à laver aux lecteurs de disques compacts, en passant par les distributeurs de billets et les téléphones, quasiment toutes nos activités quotidiennes utilisent du logiciel et, plus le temps passe, plus ce logiciel devient complexe et coûteux.

La demande de logiciels sophistiqués alourdit considérablement les contraintes imposées aux équipes de développement. Le futur des informaticiens s'annonce comme un monde de complexité croissante, à la fois du fait de la nature des applications, des environnements distribués et hétérogènes, de la taille des logiciels, de la composition des équipes de développement, et des attentes ergonomiques des utilisateurs.

Pour surmonter ces difficultés, les informaticiens vont devoir apprendre à faire, à expliquer, et à comprendre. C'est pour ces raisons qu'ils ont et auront toujours plus besoin de méthodes. Le temps de l'informatique intuitive et des programmes qui *tombent en marche* s'achève. Place au doux rêve de l'informatique adulte, raisonnée, efficace !

Les méthodes d'analyse et de conception

A quoi sert une méthode ?

Une méthode définit une démarche reproductible pour obtenir des résultats fiables. Tous les domaines de la connaissance utilisent des méthodes plus ou moins sophistiquées et plus ou moins formalisées. Les cuisiniers parlent de

recettes de cuisine, les pilotes déroulent des *check-lists* avant le décollage, les architectes dessinent des plans et les musiciens suivent des règles de composition.

De même, une méthode d'élaboration de logiciels décrit comment modéliser et construire des systèmes logiciels de manière fiable et reproductible.

De manière générale, les méthodes permettent de construire des modèles à partir d'éléments de modélisation qui constituent des concepts fondamentaux pour la représentation de systèmes ou de phénomènes. Les notes reportées sur les partitions sont des éléments de modélisation pour la musique. L'approche objet propose l'équivalent des notes – ce sont les objets – pour la représentation des logiciels.

Les méthodes définissent également une représentation – souvent graphique – qui permet d'une part de manipuler aisément les modèles, et d'autre part de communiquer et d'échanger l'information entre les différents intervenants. Une bonne représentation recherche l'équilibre entre la densité d'information et la lisibilité.

En plus des éléments de modélisation et de leurs représentations graphiques, une méthode définit des règles de mise en œuvre qui décrivent l'articulation des différents points de vue, l'enchaînement des actions, l'ordonnancement des tâches et la répartition des responsabilités. Ces règles définissent un processus qui assure l'harmonie au sein d'un ensemble d'éléments coopératifs et qui explique comment il convient de se servir de la méthode.

Avec le temps, les utilisateurs d'une méthode développent un savoir-faire lié à sa mise en œuvre. Ce savoir-faire, également appelé expérience, n'est pas toujours formulé clairement, ni aisément transmissible.

Des méthodes fonctionnelles aux méthodes objet

Les méthodes structurées et fonctionnelles se sont imposées les premières, bien que les méthodes objet aient leurs racines solidement ancrées dans les années 60. Ce n'est guère surprenant, car les méthodes fonctionnelles s'inspirent directement de l'architecture des ordinateurs, c'est-à-dire d'un domaine éprouvé et bien connu des informaticiens. La séparation entre les données et le code, telle qu'elle existe physiquement dans le matériel, a été transposée vers les méthodes ; c'est ainsi que les informaticiens ont pris l'habitude de raisonner en termes de fonctions du système. Cette démarche est naturelle lorsqu'elle est replacée dans son contexte historique ; aujourd'hui, de part son manque d'abstraction, elle est devenue quasiment anachronique. Il n'y a en effet aucune raison d'inscrire des réponses matérielles dans des solutions logicielles. Le logiciel s'exécute sur du matériel qui doit être à son service et non sur du matériel qui lui impose des contraintes d'architecture.

Plus récemment, vers le début des années 80, les méthodes objet ont commencé à émerger. L'Histoire – avec un grand H – est un éternel recommencement. La petite histoire des méthodes se répète elle aussi. Le cheminement des méthodes fonctionnelles et des méthodes objet est similaire. Au début existait la programmation, avec dans un cas le sous-programme et dans l'autre, l'objet comme élément structurant de base. Quelques années plus tard, les informaticiens poussent le concept structurant vers la conception et inventent la conception structurée dans un cas et la conception objet dans l'autre. Plus tard encore, la progression vers l'analyse est opérée, toujours en exploitant le même paradigme, soit fonctionnel, soit objet. Chaque approche peut donc proposer une démarche complète, sur l'ensemble du cycle de vie du logiciel.

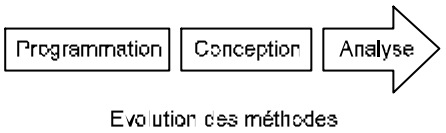


Figure 1 : L'évolution des méthodes, objet ou non, s'est toujours faite de la programmation vers l'analyse.

Dans les faits, la situation est légèrement plus complexe. En effet, souvent les méthodes ne couvrent pas l'ensemble du cycle de vie. Cela se traduit alors par la juxtaposition de méthodes : une méthode A pour l'analyse, suivie d'une méthode C pour la conception. Cette approche parcellaire, tant qu'elle est confinée dans un des paradigmes – l'approche fonctionnelle ou l'approche objet – reste raisonnable. En revanche, le mélange de paradigmes est nettement moins raisonnable, bien que compréhensible. Vers le milieu des années 80, les bienfaits de la programmation objet commencent à être largement reconnus, et la conception objet semble une approche raisonnable pour qui veut mettre en œuvre un langage de programmation objet comme Smalltalk. Du côté de l'analyse par contre, la notion d'analyse objet n'est que vapeur et supputation. Les entreprises ont développé à cette époque une solide connaissance des méthodes d'analyse fonctionnelle et de modélisation sémantique des données ; les informaticiens s'emploient donc tout naturellement à juxtaposer une phase de conception objet à une phase d'analyse fonctionnelle. Cette manière de procéder présente de nombreux inconvénients liés au changement de paradigme. Le passage de l'analyse fonctionnelle à la conception objet nécessite une traduction des éléments de modélisation fonctionnelle vers les éléments de modélisation objet, ce qui est loin d'être commode et naturel. En effet, il n'y a pas de bijection entre les deux ensembles, de sorte qu'il faut casser les éléments de modélisation d'une des approches pour construire des fragment d'éléments de modélisation de l'autre approche. Le résultat net de ce changement d'état d'esprit en cours de développement est de limiter considérablement la navigation entre l'énoncé des besoins en amont de l'analyse et la satisfaction de ces besoins en aval de la

conception. D'autre part, une conception objet obtenue après traduction manque très souvent d'abstraction et se limite à l'encapsulation des objets de bas niveaux, disponibles dans les environnements de réalisation et d'exécution. Tout ceci implique beaucoup d'efforts pour des résultats somme toute bien peu satisfaisants.

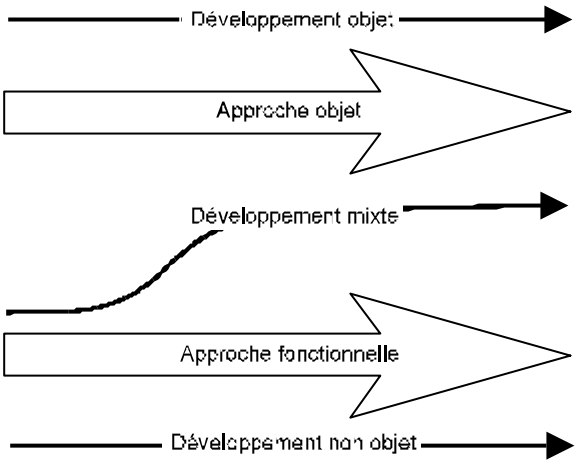


Figure 2 : La combinaison d'une approche fonctionnelle pour l'analyse et d'une approche objet pour la conception et la réalisation n'a plus lieu d'être aujourd'hui car les méthodes objet actuelles couvrent l'ensemble du cycle de vie du logiciel.

Durant la dernière décennie, des applications objet – depuis l'analyse des besoins, jusqu'à la réalisation – ont été développées dans tous les secteurs d'applications. L'expérience acquise sur ces projets permet de savoir comment enchaîner les différentes activités selon une approche totalement objet. A ce jour, l'évolution des pratiques n'est pas totale et des adeptes de la mixité existent toujours, prisonniers du poids de leurs habitudes. Les organisations qui sont sur le point de faire la transition vers l'objet ne doivent pas reproduire ce travers. Il est bien plus simple de mettre en œuvre une approche totalement objet, cela d'autant plus que les logiciels développés de cette façon sont plus simples, plus robustes et mieux adaptés aux attentes de leurs utilisateurs.

La prolifération des méthodes objet

La première moitié des années 90 a vu fleurir une cinquantaine de méthodes objet. Cette prolifération est le signe de la grande vitalité de l'objet, mais aussi le fruit d'une multitude d'interprétations de ce qui est objet, de ce qui l'est moins et de ce

qui ne l'est pas du tout[1]. Le travers de cette effervescence méthodologique est d'encourager la confusion, de sorte que les utilisateurs se placent dans une situation d'attentisme qui limite les progrès des méthodes. La meilleure validation reste la mise en œuvre ; les méthodes ne sont pas figées, elles évoluent en réponse aux commentaires des utilisateurs.

Fort heureusement, l'examen des méthodes dominantes permet de dégager un consensus autour d'idées communes. Les grands traits des objets, repris par de nombreuses méthodes, s'articulent autour des notions de classe, d'association (décrite par James Rumbaugh), de partition en sous-systèmes (Grady Booch) et autour de l'expression des besoins à partir de l'étude de l'interaction entre l'utilisateur et le système (les *use cases* d'Ivar Jacobson).

Enfin, les méthodes bien implantées – comme celles de Booch et OMT (*Object Modeling Technique*)– se sont renforcées par l'expérience, en adoptant les éléments méthodologiques les plus appréciés par les utilisateurs.

Rapprochement de Booch et OMT

Les deuxièmes moutures des méthodes de Booch et OMT, appelées respectivement Booch'93 et OMT-2, se sont rapprochées tant et si bien qu'elles sont devenues plus ressemblantes que différentes. Les variations subsistantes sont minimes et concentrées principalement dans la terminologie et dans la notation.

Booch'93 s'inspire d'OMT et adopte les associations, les diagrammes de Harel[2], les traces d'événements, etc.

OMT-2 s'inspire de Booch et introduit les flots de message, les modèles hiérarchiques et les sous-systèmes, les composants modèles, et surtout, retire du modèle fonctionnel les diagrammes de flot de données, hérités d'un passé fonctionnel et peu intégrés avec la forme générale d'OMT.

A ce stade, les deux méthodes offrent une couverture complète du cycle de vie, avec toutefois une différence notable dans l'éclairage. Booch-93 insiste plus sur la construction alors qu'OMT-2 se concentre sur l'analyse et l'abstraction. Néanmoins, il n'existe entre les deux méthodes aucune incompatibilité majeure.

Les concepts objet ont souvent une histoire compliquée et imbriquée. Les éléments présentés dans le tableau suivant se sont dégagés de l'expérience de

---

[1] Amghar Y. & al. 25-27 octobre 1995, *Domaines et utilisation de la technologie à objets*. Congrès AFCET'95, Toulouse, pp. 43-72.

[2] Harel, D. 1987. *Statecharts : A Visual Formalism for Complex Systems*. Science of Computer Programming vol. 8.

mise en œuvre des différentes méthodes et ont marqué l'effort d'unification des méthodes de Booch et OMT.

Origine	Elément
Booch	Catégories et sous-systèmes
Embley	Classes singletons et objets composites
Fusion	Description des opérations, numérotation des messages
Gamma et al.	Frameworks, patterns et notes
Harel	Automates (*Statecharts*)
Jacobson	Cas d'utilisation (*use cases*)
Meyer	Pré- et post-conditions
Odell	Classification dynamique, éclairage sur les événements
OMT	Associations
Shlaer-Mellor	Cycle de vie des objets
Wirfs-Brock	Responsabilités (*CRC*)

Figure 3 : Principaux éléments empruntés par Booch'93 et OMT-2 aux différentes méthodes objet.

L'unification des méthodes

Vers un langage unifié pour la modélisation

L'unification des méthodes de modélisation objet est rendue possible parce que l'expérience a permis de faire le tri entre les différents concepts proposés par les méthodes existantes.

Partant de la constatation que les différences entre les méthodes s'amenuisent et que la guerre des méthodes ne fait plus progresser la technologie objet, Jim Rumbaugh et Grady Booch décident fin 94 d'unifier leurs travaux au sein d'une méthode unique : la méthode unifiée (*The Unified Method*). Une année plus tard, ils sont rejoints par Ivar Jacobson, le créateur des cas d'utilisation (*use cases*), une technique très efficace pour la détermination des besoins.

Booch, Jacobson et Rumbaugh se fixent quatre objectifs :

- représenter des systèmes entiers (au-delà du seul logiciel) par des concepts objets,

- établir un couplage explicite entre les concepts et les artefacts exécutables,
- prendre en compte les facteurs d'échelle inhérents aux systèmes complexes et critiques,
- créer un langage de modélisation utilisable à la fois par les humains et les machines.

Les auteurs de la méthode unifiée atteignent très rapidement un consensus sur les concepts fondamentaux de l'objet. En revanche, la convergence sur les éléments de notation est plus difficile à obtenir et la représentation graphique retenue pour les différents éléments de modélisation connaîtra plusieurs modifications.

La première version de la description de la méthode unifiée a été présentée en octobre 1995, dans un document intitulé *Unified Method V0.8*. Ce document a été largement diffusé et les auteurs ont recueilli plus d'un millier de commentaires détaillés de la part de la communauté des utilisateurs. Ces commentaires ont été pris en compte dans la version 0.9 parue en juin 1996, mais c'est surtout la version 0.91, disponible depuis octobre 1996, qui permet de noter l'évolution de la méthode unifiée.

La principale modification consiste en la réorientation de la portée de l'effort d'unification, d'abord vers la définition d'un langage universel pour la modélisation objet, et éventuellement ensuite vers la standardisation du processus de développement objet. La méthode unifiée (*Unified Method*) se transforme en UML (*The Unified Modeling Language for Object-Oriented Development*).

En 1996, il apparaît clairement qu'UML est perçue comme un élément de base dans la stratégie de plusieurs grandes entreprises. C'est ainsi que se créé un consortium de partenaires pour travailler à la définition de la version 1.0 d'UML ; il regroupe notamment : DEC, HP, i-Logix, Intellicorp, IBM, ICON Computing, MCI Systemhouse, Microsoft, Oracle, Rational Software, TI et Unisys.

De cette collaboration naît la description d'UML version 1.0 remise à l'OMG le 17 janvier 1997, en vue de sa standardisation durant l'année 1997.

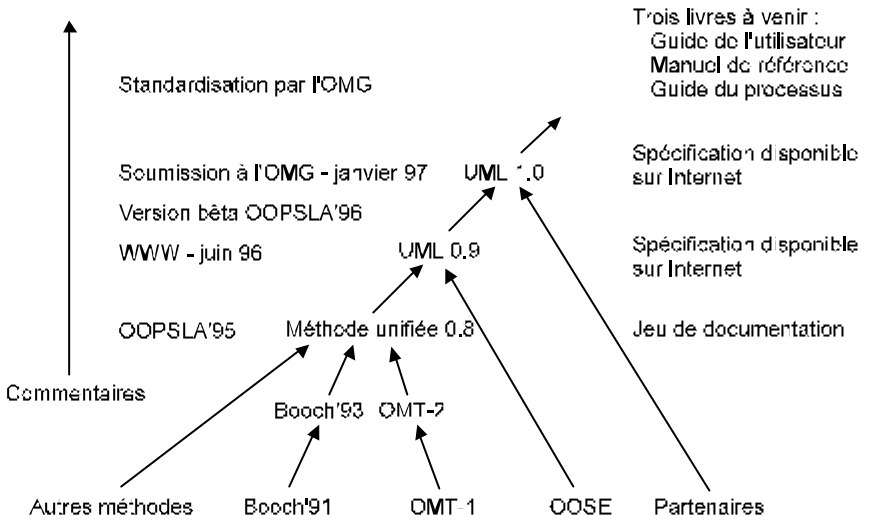


Figure 4 : Principales étapes de la définition d'UML.

Les créateurs d'UML insistent tout particulièrement sur le fait que la notation UML est un langage de modélisation objet et non pas une méthode objet. Les commentaires recueillis en réponse à la publication de la version 0.8 ont clairement montré que les utilisateurs attendaient une formalisation des artefacts du développement, plutôt que du processus d'élaboration de ces artefacts. En conséquence, la notation UML a été conçue pour servir de langage de modélisation objet, indépendamment de la méthode mise en œuvre. La notation UML peut ainsi se substituer – sans perte d'information – aux notations des méthodes de Booch, OMT ou encore OOSE (*Object Oriented Software Engineering* également appelée *Objectory*).

UML n'est pas une notation propriétaire : elle est accessible à tous et les fabricants d'outils ainsi que les entreprises de formation peuvent librement en faire usage. La volonté d'ouverture, la richesse de la notation et la définition sémantique précise des éléments de modélisation font d'UML une notation générale et simple, sans être simpliste.

En français, UML pourrait se traduire par *langage unifié pour la modélisation objet*, mais il est plus que probable qu'UML se traduise plutôt par *notation unifiée*, voire *notation UML*, sur le même schéma que *méthode OMT*.

Modèle et métamodèle

L'effort initial porte sur l'identification et la définition de la sémantique des concepts fondamentaux qui forment les briques de base de la modélisation objet. Ces concepts constituent les artefacts du développement et ils doivent pouvoir

être échangés entre les différents intervenants des projets. Pour réaliser ces échanges, il faut d'abord s'accorder sur l'importance relative de chaque concept, étudier les conséquences de ces choix et choisir une représentation graphique dont la syntaxe soit à la fois simple, intuitive et expressive.

Pour faciliter ce travail de définition et pour formaliser UML, tous les différents concepts ont été eux-mêmes modélisés avec UML. Cette définition récursive, appelée métamodélisation, présente le double avantage de permettre de classer les concepts par niveau d'abstraction, de complexité et de domaine d'application, et aussi de faire la preuve de la puissance d'expression de la notation, capable entre autres de se représenter elle-même.

Un métamodèle décrit de manière formelle les éléments de modélisation et la syntaxe et la sémantique de la notation qui permet de les manipuler. Le gain d'abstraction induit par la construction d'un métamodèle facilite l'identification d'éventuelles incohérences et encourage la généricité. Le métamodèle d'UML sert de description de référence pour la construction d'outils et pour le partage de modèles entre outils différents.

Un modèle est une description abstraite d'un système ou d'un processus, une représentation simplifiée qui permet de comprendre et de simuler. Le terme modélisation est souvent employé comme synonyme d'analyse, c'est-à-dire de décomposition en éléments simples, plus faciles à comprendre. En informatique, la modélisation consiste tout d'abord à décrire un problème, puis à décrire la solution de ce problème ; ces activités s'appellent respectivement l'analyse et la conception.

La forme du modèle dépend du métamodèle. La modélisation fonctionnelle décompose les tâches en fonctions plus simples à réaliser. La modélisation objet décompose les systèmes en objets collaborants. Chaque métamodèle définit des éléments de modélisation et des règles pour la composition de ces éléments de modélisation.

Le contenu du modèle dépend du problème. Un langage de modélisation comme UML est suffisamment général pour être employé dans tous les domaines informatiques et même au-delà, par exemple pour l'ingénierie des affaires.

Un modèle est l'unité de base du développement ; il est fortement cohérent avec lui-même et faiblement couplé avec les autres modèles par des liens de navigation. En règle générale, un modèle est relié à une phase précise du développement et est construit à partir d'éléments de modélisation avec leurs différentes vues associées.

Un modèle n'est pas directement visible par les utilisateurs. Il capture la sémantique sous-jacente d'un problème et contient des données exploitées par les outils pour l'échange d'information, la génération de code, la navigation, etc.

UML définit plusieurs modèles pour la représentation des systèmes :

- le modèle des classes qui capture la structure statique,
- le modèle des états qui exprime le comportement dynamique des objets,
- le modèle des cas d'utilisation qui décrit les besoins de l'utilisateur,
- le modèle d'interaction qui représente les scénarios et les flots de messages,
- le modèle de réalisation qui montre les unités de travail,
- Le modèle de déploiement qui précise la répartition des processus.

Les modèles sont regardés et manipulés par les utilisateurs au moyen de vues graphiques, véritables projections au travers des éléments de modélisation contenus par un ou plusieurs modèles. De nombreuses vues peuvent être construites à partir des modèles de base ; elles peuvent montrer tout ou partie des modèles. A chaque vue correspondent un ou plusieurs diagrammes. UML définit neuf types de diagrammes différents :

- les diagrammes de classes,
- les diagrammes de séquence,
- les diagrammes de collaboration,
- les diagrammes d'objets,
- les diagrammes d'états-transitions,
- les diagrammes d'activités,
- les diagrammes de cas d'utilisation,
- les diagrammes de composants,
- les diagrammes de déploiement.

Des notations différentes peuvent être des vues du même modèle. Les notations de Booch, OMT et OOSE utilisent des syntaxes graphiques différentes, mais représentent les mêmes concepts objet. Ces notations graphiques différentes ne sont en fait que des vues différentes des mêmes éléments de modélisation, de sorte qu'il est tout à fait possible d'utiliser différentes notations sans perdre le contenu sémantique. UML n'est qu'une autre représentation graphique d'un modèle sémantique commun.

2

L’approche objet

Ce chapitre présente les concepts de base de l’approche objet. Il s’adresse tout particulièrement aux informaticiens qui s’orientent vers l’objet. La maîtrise des notions présentées dans ce chapitre est essentielle pour une bonne lecture de la suite de l’ouvrage. Les exemples présentés introduisent graduellement quelques bases de la notation UML.

Pourquoi l’approche objet ?

Quelle est la raison qui rend l’approche objet tellement attractive ? A cette question, les adeptes de l’objet répondent invariablement que les avantages de l’approche objet sont la stabilité de la modélisation par rapport aux entités du monde réel, la construction itérative facilitée par le couplage faible entre composants et la possibilité de réutiliser des éléments d’un développement à un autre. Certains insistent également sur la simplicité du modèle qui ne fait appel qu’à cinq concepts fondateurs (les objets, les messages, les classes, l’héritage et le polymorphisme) pour exprimer de manière uniforme l’analyse, la conception et la réalisation d’une application informatique. Tous ces points sont parfaitement exacts, mais ils ne sont que la conséquence de la fantastique capacité d’intégration – d’unification – de l’approche objet.

D’une manière générale, toute méthode de construction de logiciels doit prendre en compte l'organisation, la mise en relation et l'articulation de structures pour en faire émerger un comportement macroscopique complexe : le système à réaliser. L'étude du système doit donc nécessairement prendre en considération l'agencement collectif des parties, et conduire à une vue plus globale des éléments qui le composent. Elle doit progresser à différents niveaux d'abstraction

et, par effet de zoom, s'intéresser aussi bien aux détails qu'à l'ordonnancement de l'ensemble.

La construction d'un logiciel est par conséquent une suite d'itérations du genre division-réunion ; il faut décomposer – diviser – pour comprendre et il faut composer – réunir – pour construire. Ceci conduit à une situation paradoxale : il faut diviser pour réunir !

Face à ce paradoxe, le processus de décomposition a été dirigé traditionnellement par un critère fonctionnel. Les fonctions du système sont identifiées, puis décomposées en sous-fonctions, et cela récursivement jusqu'à l'obtention d'éléments simples, directement représentables dans les langages de programmation (par les fonctions et les procédures).

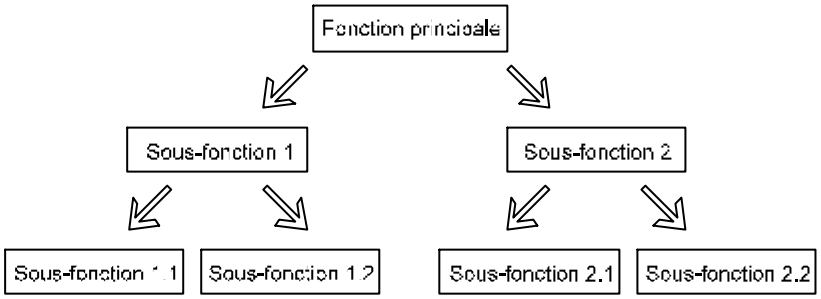


Figure 5 : Décomposition fonctionnelle et hiérarchique.

L'architecture du logiciel ainsi réalisé est le reflet des fonctions du système. Cette démarche, dont les mécanismes intégrateurs sont la fonction et la hiérarchie, apporte des résultats satisfaisants lorsque les fonctions sont bien identifiées et lorsqu'elles sont stables dans le temps. Toutefois, étant donné que la fonction induit la structure, les évolutions fonctionnelles peuvent impliquer des modifications structurelles lourdes, du fait du couplage statique entre architecture et fonctions.

L'approche objet repose à la fois sur le rationalisme d'une démarche cartésienne et sur une démarche systémique qui considère un système comme une totalité organisée, dont les éléments solidaires ne peuvent être définis que les uns par rapport aux autres. Elle propose une méthode de décomposition, non pas basée uniquement sur ce que le système fait, mais plutôt sur l'intégration de ce que le système est et fait.

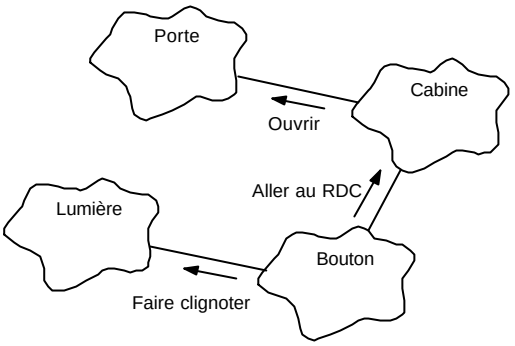


Figure 6 : Décomposition objet intégrant la structure et le comportement (notation de Booch).

Initialement basée sur l'utilisation d'objets en tant qu'abstractions du monde réel, l'approche objet a pour but une modélisation des propriétés statiques et dynamiques de l'environnement dans lequel sont définis les besoins, appelé le domaine du problème. Elle formalise notre perception du monde et des phénomènes qui s'y déroulent, et met en correspondance l'espace du problème et l'espace de la solution (l'ensemble des artefacts de réalisation), en préservant la structure et le comportement du système analysé. Les fonctions se représentent alors par des formes de collaboration entre les objets qui composent le système. Le couplage devient dynamique, et les évolutions fonctionnelles ne remettent plus en cause la structure statique du logiciel.

Lorsque le problème est totalement compris, le système informatique est réalisé en composant les représentations informatiques des éléments simples, identifiés par l'une des techniques de décomposition précédentes. La construction d'un système apparaît donc comme une démarche d'intégration, une organisation harmonieuse de composants plus élémentaires, dont la raison d'être est notre manière de gérer la complexité par la décomposition.

Dans une organisation unifiée, la distinction des composants va de pair avec leur intégration. La force de l'approche objet provient alors de sa double capacité à décomposer – différencier – et à recomposer – réunir – du fait de la richesse de ses mécanismes d'intégration, qui concernent à la fois les aspects statiques et les aspects dynamiques des logiciels. L'intégration apparaît comme la qualité fondamentale des objets ; elle assure la cohérence entre les composants d'un système informatique, au sein d'un modèle uniforme applicable à toutes les phases du cycle de vie du logiciel.

L'approche objet tire sa force de sa capacité à regrouper ce qui a été séparé, à construire le complexe à partir de l'élémentaire, et surtout à intégrer statiquement et dynamiquement les constituants d'un système.

Les objets

L'objet est une unité atomique formée de l'union d'un état et d'un comportement. Il fournit une relation d'encapsulation qui assure à la fois une cohésion interne très forte et un faible couplage avec l'extérieur. L'objet révèle son vrai rôle et sa vraie responsabilité lorsque, par l'intermédiaire de l'envoi de messages, il s'insère dans un scénario de communication.

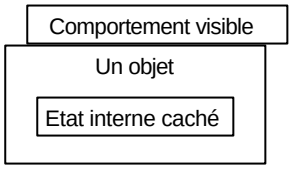


Figure 7 : Chaque objet contient un état interne qui lui est propre et un comportement accessible aux autres objets.

Le monde dans lequel nous vivons est constitué d'objets matériels de toutes sortes. La taille de ces objets est très variable : certains sont petits, comme les grains de sable, et d'autres très gros, comme les étoiles. Notre perception intuitive de ce qui constitue un objet est fondée sur le concept de masse, c'est-à-dire sur une grandeur qui caractérise la quantité de matière d'un corps.

Par extension, il est tout à fait possible de définir d'autres objets, sans masse, comme les comptes en banques, les polices d'assurance ou encore les équations mathématiques. Ces objets correspondent alors à des concepts plutôt qu'à des entités physiques.

Toujours par extension, les objets peuvent également appartenir à des mondes virtuels, par exemple, en association avec Internet, pour créer des communautés de personnes qui ne sont pas situées au même point géographique.

Les objets informatiques définissent une représentation abstraite des entités d'un monde réel ou virtuel, dans le but de les piloter ou de les simuler. Cette représentation abstraite peut être vue comme une sorte de miroir informatique, qui renvoie une image simplifiée d'un objet qui existe dans le monde perçu par l'utilisateur. Les objets informatiques, que nous appellerons désormais objets, encapsulent une partie de la connaissance du monde dans lequel ils évoluent.

Les utilisateurs des technologies objet ont pris l'habitude de considérer les objets comme des êtres animés d'une vie propre, de sorte qu'ils les présentent souvent

dans une vision anthropomorphique[3]. Ainsi, les objets sont souvent désignés en anglais par *she* ou *he* plutôt que par *it*.

Comme les êtres vivants, les objets du monde réel qui nous entourent naissent, vivent et meurent. La modélisation objet permet de représenter le cycle de vie des objets au travers de leurs interactions.

En UML, un objet se représente sous la forme d'un rectangle ; le nom de l'objet est souligné. Le diagramme suivant représente trois objets.

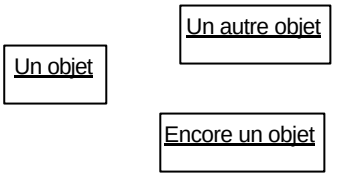


Figure 8 : En UML les objets se représentent au moyen d'icônes rectangulaires.

Le diagramme suivant représente plusieurs clients d'une banque et les comptes associés à chacun de ces clients. Les traits qui relient les objets symbolisent les liens qui existent entre un client particulier et un compte particulier. Le graphique présente également un rectangle dont le coin haut droit est replié : il s'agit de la représentation d'une note, c'est-à-dire d'une information de clarification optionnelle exprimée dans un format libre, afin de faciliter la compréhension du diagramme. Les traits discontinus permettent de relier n'importe quel ensemble d'éléments de modélisation à une note descriptive.

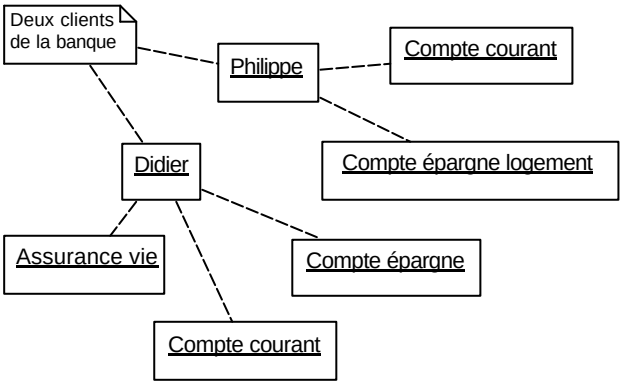


Figure 9 : Représentation graphique d'objets ayant un lien entre eux.

---

[3] Wirfs-Brock R., Wilkerson B., Wiener L. 1990, *Designing Object-Oriented Software*. Prentice-Hall, Englewood Cliffs, NJ.

Il est souvent difficile de trouver un nom pour désigner chaque objet ; c'est pourquoi la notation permet d'indiquer un nom générique plutôt que leur nom individuel. Cet artifice permet de parler des objets en termes généraux, tout en évitant la multiplication de noms du type **`ga`**, **`bu`**, **`zo`** ou **`meu`**, qui véhiculent un contenu sémantique très approximatif.

Le diagramme suivant montre des élèves et des professeurs. Les deux points précisent qu'il s'agit d'objets anonymes, de genre **`Elève`** et **`Professeur`**.

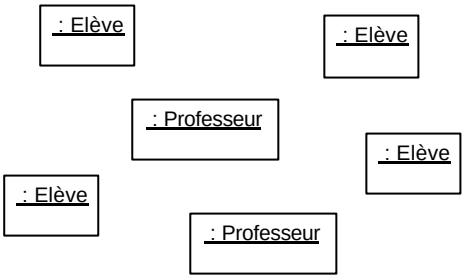


Figure 10 : Représentation d'objets anonymes.

Caractéristiques fondamentales d'un objet

La présentation des caractéristiques fondamentales d'un objet permet de répondre de manière plus formelle à la sempiternelle question : « mais qu'est-ce qui définit un objet ? ». Tout objet présente les trois caractéristiques suivantes : un état, un comportement et une identité.

```
Objet = Etat + Comportement + Identité
```

Un objet doit apporter une valeur ajoutée par rapport à la simple juxtaposition d'informations ou de code exécutable. Un objet sans état ou sans comportement peut exister marginalement, mais dans tous les cas, un objet possède une identité.

L'état

L'état regroupe les valeurs instantanées de tous les attributs d'un objet sachant qu'un attribut est une information qui qualifie l'objet qui le contient. Chaque attribut peut prendre une valeur dans un domaine de définition donné. L'état d'un objet, à un instant donné, correspond à une sélection de valeurs, parmi toutes les valeurs possibles des différents attributs.

Le diagramme suivant montre une voiture qui contient les valeurs de trois attributs différents : la couleur, la masse et la puissance fiscale.

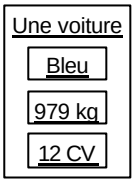


Figure 11 : L'état regroupe les valeurs des différents attributs qui caractérisent un objet. Dans le cas d'une voiture, la couleur ou la masse font partie de l'état.

L'état évolue au cours du temps ; ainsi, lorsqu'une voiture roule, la quantité de carburant diminue, les pneus s'usent et la température dans l'habitacle varie. Certaines composantes de l'état peuvent être constantes : c'est le cas par exemple de la marque de la voiture, ou encore du pays de sa construction. Toutefois, en règle générale, l'état d'un objet est variable et peut être vu comme la conséquence de ses comportements passés.

Dans l'exemple suivant, une voiture donnée parcourt une centaine de kilomètres ; pendant qu'elle se déplace, la valeur qui symbolise la quantité de carburant diminue avec la distance parcourue.

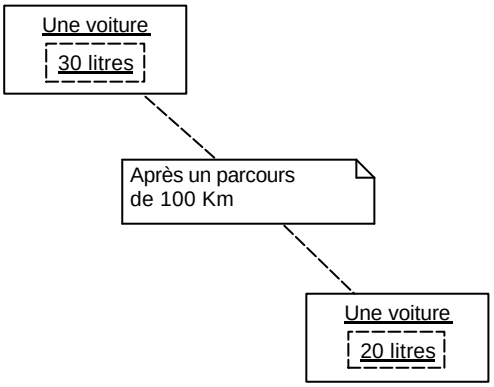


Figure 12 : L'état d'un objet à un moment donné est la conséquence des comportements passés.

Le comportement

Le comportement regroupe toutes les compétences d'un objet et décrit les actions et les réactions de cet objet. Chaque atome de comportement est appelé opération. Les opérations d'un objet sont déclenchées suite à une stimulation externe, représentée sous la forme d'un message envoyé par un autre objet.

Dans le diagramme suivant, selon la valeur du message, l'**`Opération 1`** ou l'**`Opération 2`** est déclenchée.

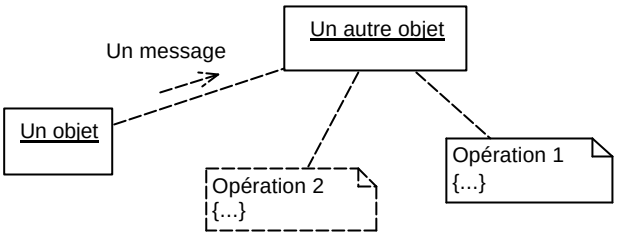


Figure 13 : En réponse à un message, l'objet destinataire déclenche un comportement. Le message sert de sélecteur pour déterminer l'opération à exécuter.

Les interactions entre objets se représentent au moyen de diagrammes dans lesquels les objets qui interagissent sont reliés entre eux par des lignes continues appelées liens. La présence d'un lien signifie qu'un objet connaît ou voit un autre objet. Les messages naviguent le long des liens, a priori dans les deux directions. Dans l'exemple suivant, l'objet **A** demande à l'objet **B** de manger, et l'objet **B** demande à l'objet **C** de dormir. Ceci sous-entend que l'objet **B** possède la faculté de manger et que l'objet **C** est capable de dormir.

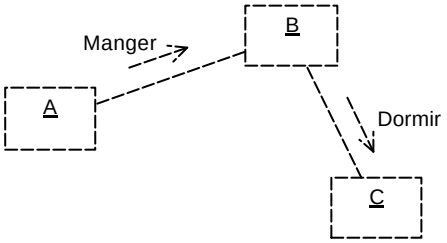


Figure 14 : Représentation d'une interaction entre des objets.

L'état et le comportement sont liés ; en effet, le comportement à un instant donné dépend de l'état courant, et l'état peut être modifié par le comportement. Il n'est possible de faire atterrir un avion que s'il est en train de voler, c'est-à-dire que le comportement **`Atterrir`** n'est valide que si l'information **`En Vol`** est valide. Après l'atterrissage, l'information **`En Vol`** devient invalide, et l'opération **`Atterrir`** n'a plus de sens. L'exemple suivant montre les liens entre l'état et le comportement.

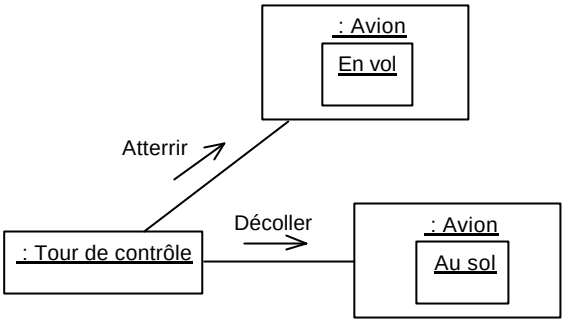


Figure 15 : Le comportement et les attributs sont liés.

L'identité

En plus de son état, un objet possède une identité qui caractérise son existence propre. L'identité permet de distinguer tout objet de façon non ambiguë, et cela indépendamment de son état[4]. Cela permet, entre autres, de distinguer deux objets dont toutes les valeurs d'attributs sont identiques.

L'identité est un concept, elle ne se représente pas de manière spécifique en modélisation. Chaque objet possède une identité de manière implicite.

En phase de réalisation, l'identité est souvent construite à partir d'un identifiant issu naturellement du domaine du problème. Nos voitures possèdent toutes un numéro d'immatriculation, nos téléphones un numéro d'appel, et nous-mêmes sommes identifiés par notre numéro de sécurité sociale. Ce genre d'identifiant, appelé également clé naturelle, peut être rajouté dans l'état des objets afin de les distinguer. Il ne s'agit toutefois que d'un artifice de réalisation, car le concept d'identité reste indépendant du concept d'état.

Contraintes de réalisation

Les notions d'état, de comportement et d'identité décrites dans le chapitre précédent, s'appliquent aux objets de manière très générale, quel que soit l'environnement de réalisation. Toutefois, les objets peuvent également posséder des caractéristiques plus informatiques, liées aux contingences de réalisation comme la distribution des programmes, les bases de données ou les développements multilangages.

---

[4] Khoshafian, S. N., Copeland G. P. 1986, *Object Identity*. OOPSLA'86 Conference Proceedings.

La persistance des objets

La persistance désigne la capacité d'un objet à transcender le temps ou l'espace. Un objet persistant sauvegarde son état dans un système de stockage permanent, de sorte qu'il est possible d'arrêter le processus qui l'a créé sans perdre l'information représentée par l'objet (passivation de l'objet). Par la suite, l'objet peut être reconstruit (activation de l'objet) par un autre processus et se comportera exactement comme dans le processus initial. Les objets non persistants sont dits transitoires ou éphémères. Par défaut, les objets ne sont pas considérés comme persistants.

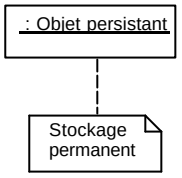


Figure 16 : L'état des objets persistants est sauvegardé dans un système de stockage permanent.

Dans leur ensemble, les langages de programmation objet ne proposent pas de support direct pour assurer la persistance des objets. Cela est regrettable et oblige à recourir à des artifices extérieurs pour assurer la persistance des objets. Les constructeurs de bases de données fournissent des solutions pour la sauvegarde des objets, soit totalement objet, soit hybrides.

La transmission des objets

La problématique de la persistance des objets est très proche de celle de la migration des objets d'un processus vers un autre. En effet, il s'agit alors d'envoyer un objet par un moyen de communication quelconque d'un espace d'adressage vers un autre espace d'adressage. Cette opération s'apparente étroitement à la démarche de passivation et d'activation décrite précédemment. Les objets ne voyagent pas réellement : l'objet original est analysé lors de l'émission, la description de l'objet est transmise au travers du support de communication, un clone de l'objet est reconstruit lors de la réception et l'objet initial est supprimé. Les amateurs de *Star Trek* auront reconnu le principe de la télé-transportation[5].

[5] Metzger G. 1996, *Beam me up, Spock. There is no life on this planet*. Communication privée.

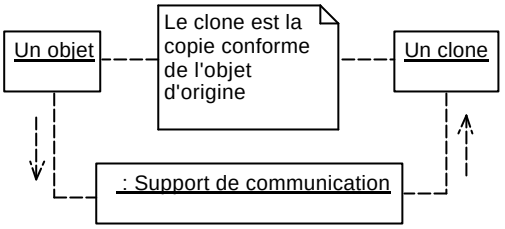


Figure 17 : Les objets peuvent être transmis le long d'un support de communication.

Les objets miroirs

Les objets miroirs constituent une alternative à la transmission des objets. Un objet miroir se comporte comme un autre objet avec lequel il est synchronisé. Le client manipule le miroir comme s'il manipulait l'objet distant, ce qui permet de dissimuler toute la complexité de la communication au fond des objets miroirs.

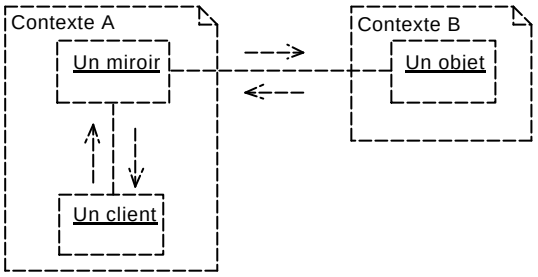


Figure 18 : Le miroir renvoie dans le contexte ***A****, l'image d'un objet défini dans le contexte* ***B****.*

Communication entre objets

Les systèmes informatiques à objets peuvent être vus comme des sociétés d'objets qui travaillent en synergie afin de réaliser les fonctions de l'application. Lors de l'exécution d'un programme, les objets contribuent solidairement au bon déroulement de l'application informatique. Le comportement global d'une application repose donc sur la communication entre les objets qui la composent.

De ce fait, l'étude des formes de communication entre objets du domaine est de première importance dans la modélisation objet et, d'ailleurs, la grande différence entre l'approche fonctionnelle et l'approche objet réside précisément dans cette articulation qui réduit le couplage entre la structure et la fonction.

Catégories de comportement

Les objets interagissent pour réaliser les fonctions de l'application. Selon la nature des interactions, c'est-à-dire selon la direction des messages échangés, il

est possible de décrire de manière générale le comportement des objets. Il est fréquent de retenir trois catégories de comportement : les acteurs, les serveurs et les agents.

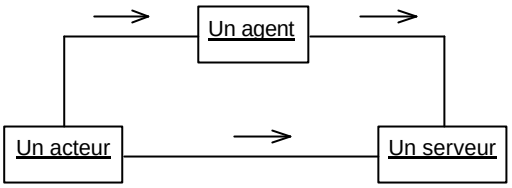


Figure 19 : Les objets peuvent être regroupés dans trois catégories selon la nature de leur comportement.

Les acteurs[6] sont toujours des objets à l'origine d'une interaction. Ce sont généralement des objets actifs, c'est-à-dire qu'ils possèdent un fil d'exécution (*thread*) et que ce sont eux qui passent la main aux autres objets.

Les serveurs, au contraire, ne sont jamais à l'origine d'une interaction, mais sont toujours les destinataires des messages. Ce sont souvent des objets passifs qui attendent qu'un autre objet ait besoin de leurs services. Dans ce cas, le flot de contrôle est passé au serveur par l'objet qui envoie le message et est récupéré après exécution du service.

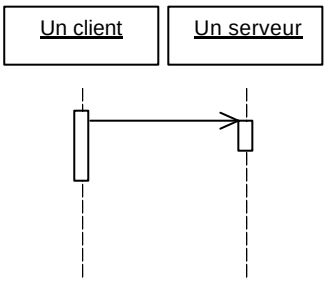


Figure 20 : L'objet client passe la main à l'objet serveur, puis attend la fin du service avant de reprendre son exécution.

Les agents cumulent les caractéristiques des acteurs et des serveurs. Ces objets ont un comportement très proche de celui des humains ; ils peuvent interagir avec les autres objets à tout moment, de leur fait ou suite à une sollicitation externe.

---

[6] Le terme acteur est également employé pour désigner une catégorie d'utilisateurs dans le modèle des cas d'utilisation (*use case*) qui est présenté plus loin dans l'ouvrage.

Les agents sont à la base du mécanisme de délégation qui permet à un objet de se comporter comme un paravent devant un autre objet. Les agents découplent les objets clients des objets fournisseurs, en introduisant une indirection dans le mécanisme de propagation des messages. De cette manière, un client peut communiquer avec un serveur qu'il ne connaît pas directement et, de plus, le serveur peut changer entre deux passages de messages.

Dans l'exemple suivant, le client communique indirectement avec le premier serveur sans le connaître et sans savoir qu'il existe deux autres serveurs. Le routage des messages du client vers le serveur est assuré dynamiquement par l'agent intermédiaire.

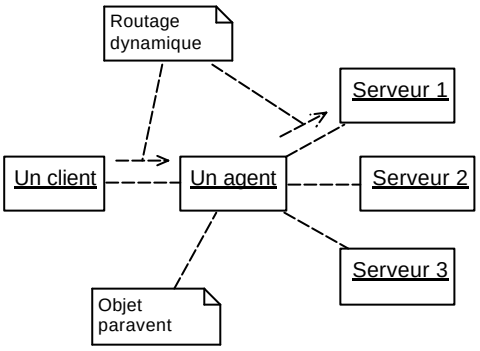


Figure 21 : Illustration du mécanisme de délégation.

Le concept de message

L'unité de communication entre objets s'appelle le message. Le message est le support d'une relation de communication qui relie, de façon dynamique, les objets qui ont été séparés par le processus de décomposition. Il permet l'interaction de manière flexible, en étant à la fois agent de couplage et agent de découplage. C'est lui qui assure la délégation des tâches et garantit le respect des contraintes. Le message est un intégrateur dynamique qui permet de reconstituer une fonction de l'application par la mise en collaboration d'un groupe d'objets. Il acquiert toute sa force d'intégration lorsqu'il est associé au polymorphisme et à la liaison dynamique, définis plus loin dans ce chapitre. Les messages, comme le montre la figure suivante, sont représentés par des flèches placées le long des liens qui unissent les objets.

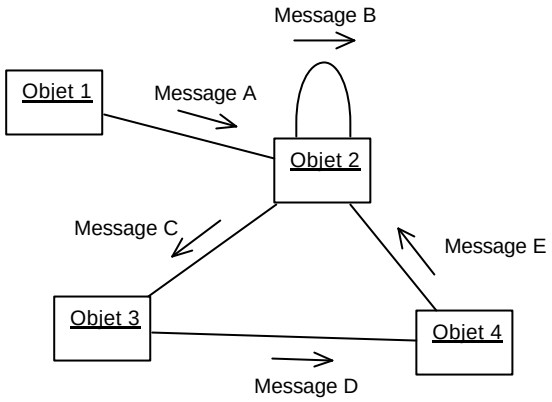


Figure 22 : Les objets communiquent en échangeant des messages.

Un message regroupe les flots de contrôle et les flots de données au sein d'une entité unique. La notion de message est un concept abstrait qui peut être mis en œuvre selon de nombreuses variantes, comme l'appel de procédure, l'événement discret, l'interruption, le datagramme UDP, la recherche dynamique, etc.

Le diagramme suivant décrit complètement un message. La flèche simple indique le flot de contrôle et les flèches décorées de petits cercles montrent les flots de données.

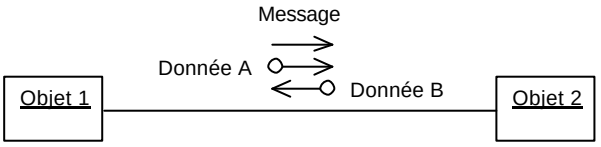


Figure 23 : Les différentes flèches montrent le flot de contrôle et les flots de données.

Catégories de messages

Il existe cinq catégories principales de messages :

- les constructeurs qui créent des objets,
- les destructeurs qui détruisent des objets,
- les sélecteurs qui renvoient tout ou partie de l'état d'un objet,
- les modificateurs qui changent tout ou partie de l'état d'un objet,
- les itérateurs qui visitent l'état d'un objet ou le contenu d'une structure de données qui contient plusieurs objets.

L'exemple suivant montre une classe C++ dont les fonctions membres ont été regroupées selon la classification proposée plus haut :

```
class Lapin
{
public:
    // Constructeurs
    Lapin();
    Lapin(const Lapin &right);
    // Destructeur
    ~Lapin();
    // Assignation
    const Lapin & operator=(const Lapin &right);
    // Egalité
    int operator==(const Lapin &right) const;
    int operator!=(const Lapin &right) const;
    // Autres Operations
    void Manger();
    void Dormir();
    // Sélecteurs
    const String Nom() const;
    const Int Age() const;
    // Modifieurs
    void ChangerNom(const String value);
    void ChangerAge(const Int value);
private:
  String Nom;
  Int Age;
};
```

Figure 24 : Exemple de classe C++ dont les fonctions membres ont été regroupées selon les grandes catégories de messages.

Formes de synchronisation des messages

Les formes de synchronisation des messages décrivent la nature des mécanismes de communication qui permettent le passage de messages d'un objet vers un autre objet.

La notion de synchronisation prend tout son intérêt lorsque plusieurs objets sont actifs simultanément et qu'il faut, par exemple, protéger l'accès à des objets partagés. Le diagramme suivant montre un objet partagé qui assure l'interface vers un dispositif d'entrée-sortie.

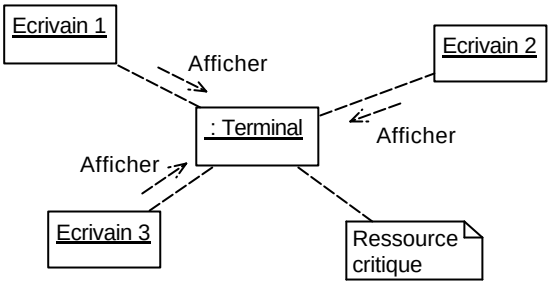


Figure 25 : Exemple d'un objet ***`Terminal`*** *accédé par plusieurs objets* ***`Ecrivain`***.

Dans une application purement séquentielle, les écrivains parlent au terminal chacun à leur tour et l'affichage sur l'écran ne pose aucun problème. En revanche, dès lors que plusieurs objets **`Ecrivain`** peuvent être actifs simultanément, l'objet **`Terminal`** devient une ressource critique dont il convient de protéger les accès par synchronisation des envois de messages.

La notion de synchronisation précise la nature de la communication, et les règles qui régissent le passage des messages. Il existe cinq grandes catégories d'envoi de message :

- *Envoi de message simple.* Cette catégorie convient pour les systèmes à un seul flot d'exécution, dans lesquels un seul objet à la fois est actif. Le passage du contrôle s'effectue lors de l'envoi d'un message de l'objet actif vers un objet passif. Un envoi de message simple se représente par une flèche simple.

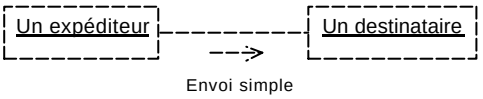


Figure 26 : Représentation d'un envoi de message simple.

- *Envoi de message synchrone.* Un message synchrone ne déclenche une opération que lorsque le destinataire accepte le message. Une fois le message envoyé, l'expéditeur est bloqué jusqu'à ce que le destinataire accepte le message. Un envoi de message synchrone se représente par une flèche barrée d'une croix.

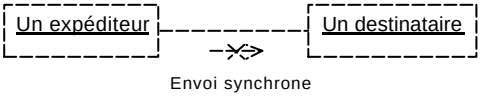


Figure 27 : Représentation d'un envoi de message synchrone.

- *Envoi de message dérobant.* Un message dérobant déclenche une opération seulement si le destinataire s'est préalablement mis en attente du message. Ce type de synchronisation correspond à une tolérance d'attente inverse de celle de l'envoi de message synchrone. Dans le cas d'un message synchrone, l'expéditeur accepte d'attendre ; dans le cas d'un message dérobant, le destinataire accepte d'attendre. Un envoi de message dérobant se représente par une flèche qui se retourne vers l'expéditeur.

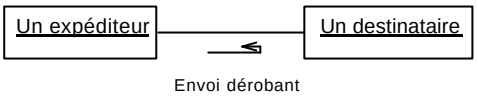


Figure 28 : Représentation d'un envoi de message dérobant.

- *Envoi de message minuté.* Un message minuté bloque l'expéditeur pendant un temps donné, en attendant la prise en compte par le destinataire. L'expéditeur est libéré si la prise en compte n'a pas eu lieu au bout du temps spécifié dans la description de l'envoi du message minuté. L'envoi de message dérobant correspond au cas limite d'un envoi minuté avec un délai d'attente nul. Un envoi de message minuté se représente par une flèche décorée d'une montre symbolisée par un petit cercle.

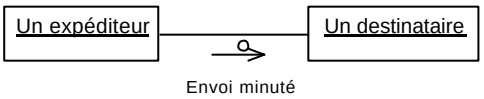


Figure 29 : Représentation d'un envoi de message minuté.

- *Envoi de message asynchrone.* Un message asynchrone n'interrompt pas l'exécution de l'expéditeur. L'expéditeur envoie le message sans savoir quand, ni même si, le message sera traité par le destinataire. Du point de vue du destinataire, un envoi asynchrone doit pouvoir être pris en compte à tout moment. Un envoi de message asynchrone se représente par une demi-flèche.

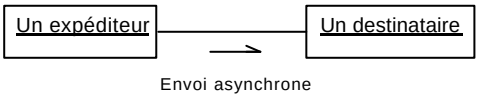


Figure 30 : Représentation d'un envoi de message asynchrone.

La précision de la forme de synchronisation des messages est souvent opérée en conception, pour réaliser par exemple une exclusion mutuelle autour d'une ressource critique. Le diagramme suivant montre que les différents écrivains

communiquent de façon synchrone avec le terminal. Le terminal sérialise les affichages et les différents écrivains doivent attendre chacun leur tour.

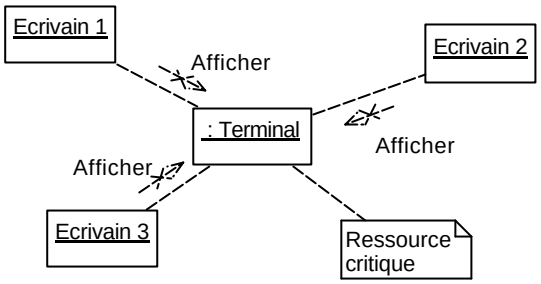


Figure 31 : Exemple de communication par messages synchrones.

La représentation de la synchronisation des messages est également utile en analyse, comme le montre le diagramme suivant qui illustre une communication par téléphone. **`Antoine`** appelle **`Marc`** au téléphone ; la communication ne peut avoir lieu que si **`Marc`** décroche. **`Antoine`** n'attend pas indéfiniment et peut raccrocher après trois sonneries.

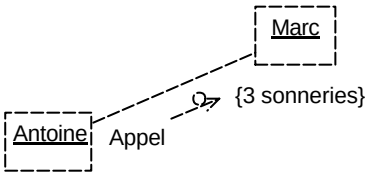


Figure 32 : Exemple d'une communication minutée.

La communication par voie épistolaire suit un schéma asynchrone. **`Gérard`** envoie une lettre à **`Bernard`** ; il ne sait pas quand la lettre arrivera, ni même si la lettre arrivera, et il n'attend pas que **`Bernard`** la reçoive.

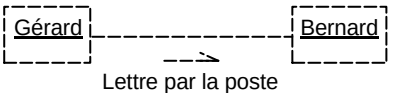


Figure 33 : Exemple d'une communication asynchrone.

Représentation des interactions entre les objets

Les objets interagissent pour réaliser collectivement les services offerts par les applications. Les diagrammes d'interaction représentent les objets les uns par rapport aux autres et montrent comment ils communiquent au sein d'une

interaction. Chaque interaction possède un nom et un contexte de validité qu'il convient de préciser de manière textuelle.

Il existe deux sortes de diagrammes d'interaction : les diagrammes de collaboration et les diagrammes de séquence.

Les diagrammes de collaboration

Les diagrammes de collaboration correspondent aux diagrammes utilisés dans les exemples précédents. Ces diagrammes montrent quelques objets dans une situation donnée. Les objets sont représentés sous forme de rectangles, des liens relient les objets qui se connaissent (c'est-à-dire qui peuvent interagir) et les messages échangés par les objets sont représentés le long de ces liens. L'ordre d'envoi des différents messages est matérialisé par un numéro placé en tête du message.

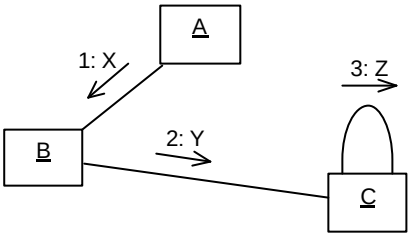


Figure 34 : Exemple de diagramme de collaboration.

Le diagramme ci-dessus se lit de la manière suivante :

le scénario débute par un objet ***A*** *qui envoie un message* ***X*** *à un objet* ***B****, puis l'objet* ***B*** *envoie un message* ***Y*** *à un objet* ***C****, et enfin* ***C*** *s'envoie un message* ***Z****.*

Le message **Z** est un artifice de notation pour représenter une activité ayant lieu dans l'objet **C**.

Les diagrammes de collaboration sont particulièrement indiqués pour la phase exploratoire qui correspond à la recherche des objets. L'agencement des objets dans le diagramme peut évoquer la disposition spatiale des objets dans le monde réel, tout en montrant une forme d'interaction. Les diagrammes d'objets décrivent à la fois la structure statique par des liens entre objets, et le comportement, à travers des envois de messages le long de ces liens. Ce type de diagramme possède cependant les limites habituelles des représentations graphiques – la visualisation claire d'un nombre limité d'informations de sorte que seule une petite collaboration est représentable. Le diagramme suivant illustre les limites du diagramme de collaboration : le grand nombre de messages obscurcit le diagramme.

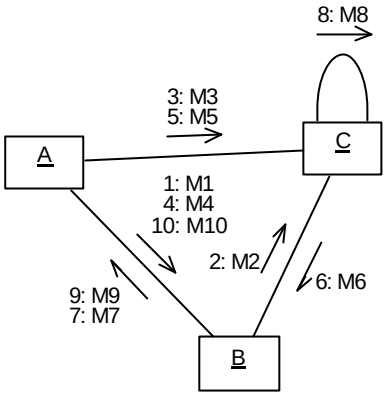


Figure 35 : Illustration de la limite d'expressivité d'un diagramme de collaboration.

Les diagrammes de séquence

Les diagrammes de séquence montrent à peu près les mêmes informations que les diagrammes précédents, mais l'accent est mis sur la communication, au détriment de la structure spatiale. Chaque objet est représenté par une barre verticale. Le temps s'écoule de haut en bas, de sorte que la numérotation des messages est optionnelle.

Le passage d'un diagramme à l'autre est possible automatiquement, dès lors que seules les informations de présence d'objets et de communication sont retenues. Le diagramme de collaboration précédent correspond au diagramme de séquence suivant :

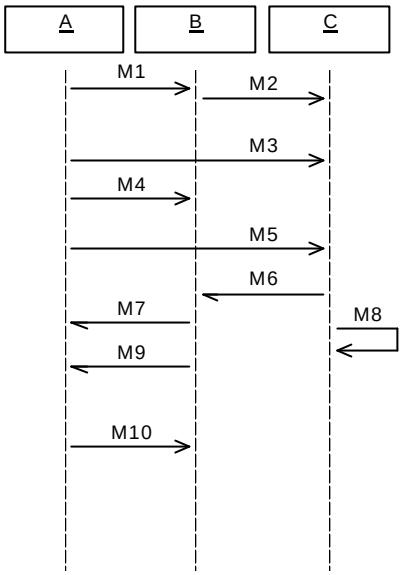


Figure 36 : Diagramme de séquence dérivé du diagramme de collaboration précédent.

Le diagramme précédent montre uniquement la chronologie des messages. Les barres verticales peuvent être décorées de bandes rectangulaires, afin de montrer la distribution du flot de contrôle parmi les différents objets.

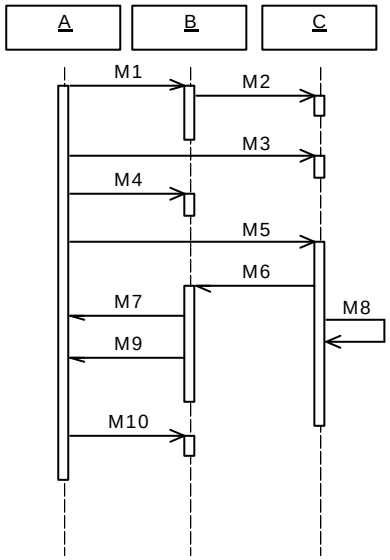


Figure 37 : Représentation de la distribution du flot de contrôle parmi les objets.

Les deux types de diagrammes d'interaction sont intéressants pour la modélisation objet. Le diagramme de séquence est particulièrement bien adapté pour la représentation d'interactions complexes, du fait de sa forme quasi tabulaire. Le diagramme de collaboration se prête mieux à la découverte des abstractions, car il permet de montrer les objets du domaine dans une disposition physique proche de la réalité. Dans la pratique, il est fréquent de commencer par représenter les principaux objets du domaine dans des diagrammes de collaboration, puis, une fois les objets bien identifiés, de migrer vers les diagrammes de séquence pour la représentation des interactions dans toute leur complexité.

Les classes

Le monde réel est constitué de très nombreux objets en interaction. Ces objets constituent des amalgames souvent trop complexes pour être compris dans leur intégralité du premier coup. Pour réduire cette complexité – ou du moins pour la maîtriser – et comprendre ainsi le monde qui l'entoure, l'être humain a appris à regrouper les éléments qui se ressemblent et à distinguer des structures de plus haut niveau d'abstraction, débarrassées de détails inutiles.

La démarche d'abstraction

L'abstraction est une faculté des êtres humains qui consiste à concentrer la réflexion sur un élément d'une représentation ou d'une notion, en portant spécialement l'attention sur lui et en négligeant tous les autres. La démarche d'abstraction procède de l'identification des caractéristiques communes à un ensemble d'éléments, puis de la description condensée – analogue à la description d'un ensemble en compréhension – de ces caractéristiques dans ce qu'il est convenu d'appeler une classe. La démarche d'abstraction est arbitraire : elle se définit par rapport à un point de vue. Ainsi, un objet du monde réel peut être vu au travers d'abstractions différentes, ce qui implique qu'il est important de déterminer quels sont les critères pertinents dans le domaine d'application considéré.

La classe décrit le domaine de définition d'un ensemble d'objets. Chaque objet appartient à une classe. Les généralités sont contenues dans la classe et les particularités sont contenues dans les objets. Les objets informatiques sont construits à partir de la classe, par un processus appelé instanciation. De ce fait, tout objet est une instance de classe.

Les langages objet permettent de décrire et de manipuler des classes et leurs instances. Cela signifie que l'utilisateur peut construire en machine une représentation informatique des abstractions qu'il a l'habitude de manipuler

mentalement, sans traduction vers des concepts de plus bas niveau (comme les fonctions ou les procédures des langages de programmation non objet).

Les langages objet réduisent la distance entre notre façon de raisonner (par abstraction) et le langage compris par les ordinateurs, de sorte qu'il est globalement plus facile de réaliser une application avec un langage objet qu'avec un langage traditionnel, même si l'approche objet demande une remise en cause des habitudes acquises.

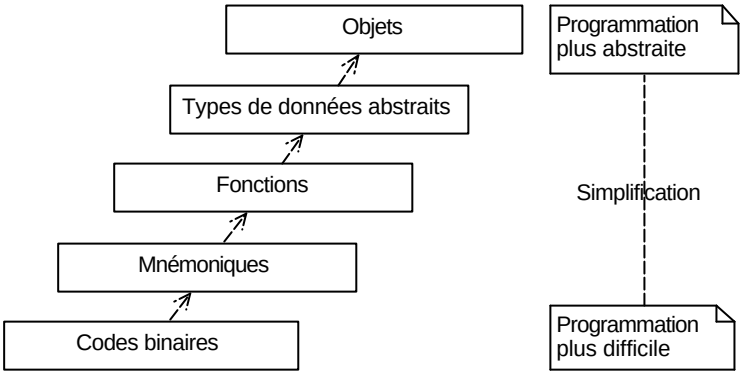


Figure 38 : Les langages de programmation objet permettent une programmation plus abstraite.

Représentation graphique des classes

Chaque classe est représentée sous la forme d'un rectangle divisé en trois compartiments. Le premier compartiment contient le nom de la classe, le second les attributs et le dernier les opérations. Par défaut, les attributs sont cachés et les opérations sont visibles. Les compartiments peuvent être supprimés pour alléger les diagrammes.

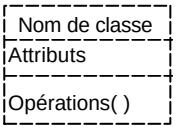


Figure 39 : Représentation graphique des classes.

Les quelques exemples suivants illustrent l'emploi des classes pour décrire de manière générale quelques objets de notre entourage.

La classe **Motocyclette** contient les attributs **Couleur**, **Cylindrée** et **Vitesse maximale**. La classe regroupe également les opérations applicables aux instances de la classe, comme ici les opérations **Démarrer()**, **Accélérer()** et **Freiner()**.

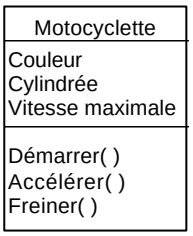


Figure 40 : Exemple d'une classe ***Motocyclette****.*

L'ensemble des nombres complexes regroupe des nombres qui ont une partie réelle et une partie imaginaire. La connaissance de la représentation interne du nombre complexe – cartésienne ou polaire – n'est pas nécessaire pour utiliser les opérations décrites dans l'exemple suivant. La classe **Nombre complexe** cache les détails de sa réalisation.

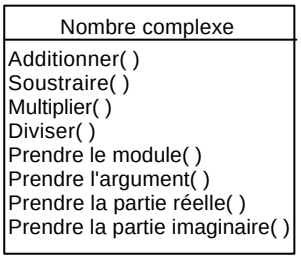


Figure 41 : La classe ***Nombre complexe*** *dissimule les détails de sa représentation interne.*

Un téléviseur est un équipement électronique d'une complexité non négligeable, qui peut néanmoins être utilisé aisément même par de très petits enfants. Le téléviseur offre une abstraction simple, au travers de quelques opérations élémentaires comme **Changer de programme** ou **Régler le volume sonore**.

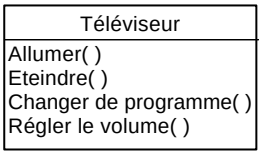


Figure 42 : Exemple d'une classe ***`Téléviseur`****.*

Une transaction bancaire est une abstraction d'une opération immatérielle, qui réifie une interaction entre un client et une banque. Le détail de réalisation des transactions courantes, comme le retrait ou le dépôt, ne sont pas connus du client qui se contente d'indiquer le compte sur lequel il désire opérer et le montant concerné. Le compte est une autre abstraction du domaine bancaire.

L'abstraction dissimule la complexité de la gestion des comptes, de sorte que les transactions peuvent être réalisées simplement par le client tout seul, depuis un guichet automatique ou depuis un Minitel.

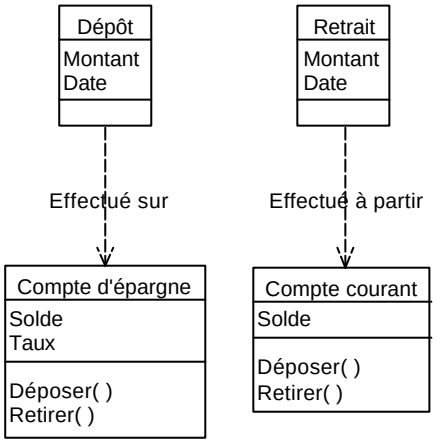


Figure 43 : Représentation d'une partie du domaine bancaire.

L'électronique des circuits intégrés exploite la notion d'abstraction avec beaucoup de succès. La complexité électronique et les détails internes sont totalement invisibles pour l'utilisateur de ces composants. Dans le cas des portes logiques, le circuit intégré réifie une abstraction fonctionnelle.

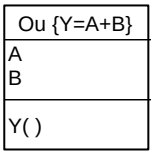


Figure 44 : Exemple de réification d'une abstraction fonctionnelle.

Tous les types de données abstraits manipulés par les informaticiens sont, comme leur nom l'indique, des abstractions décrites en termes d'opérations applicables sur des valeurs. Ce genre d'abstraction appartient typiquement à la conception, et n'apparaît jamais en analyse où le terme collection est suffisant pour désigner les regroupements d'objets.

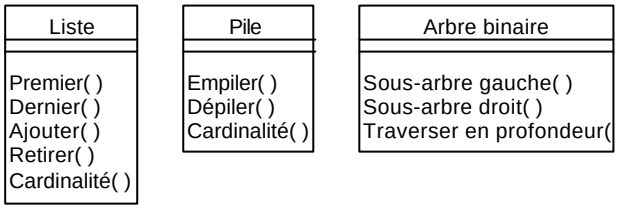


Figure 45 : Types de données abstraits présentés sous la forme de classes.

Description des classes

La description des classes est séparée en deux parties :

- la spécification d'une classe qui décrit le domaine de définition et les propriétés des instances de cette classe, et qui correspond à la notion de type telle qu'elle est définie dans les langages de programmation classiques,
- la réalisation qui décrit comment la spécification est réalisée et qui contient le corps des opérations et les données nécessaires à leur fonctionnement.

Une classe passe un contrat avec d'autres classes ; elle s'engage à fournir les services publiés dans sa spécification et les autres classes s'engagent à ne pas faire usage de connaissances autres que celles décrites dans cette spécification.

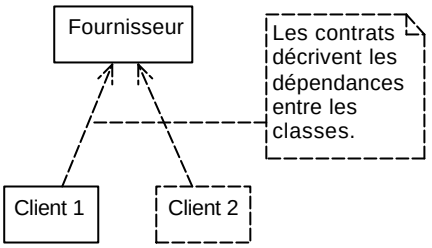


Figure 46 : Les classes passent un contrat basé sur la spécification du fournisseur.

Les langages modulaires permettent la compilation séparée de la spécification et de la réalisation, de sorte qu'il est possible de valider tout d'abord la cohérence des spécifications (appelées aussi interfaces) et de s'attaquer plus tard à la réalisation.

Selon les langages de programmation, les concepts de type, de description et de modules sont plus ou moins intégrés dans le concept de classe :

- En Ada 95 par exemple, une classe est construite explicitement, en plaçant un type (privé) accompagné de ses opérations dans un module (paquetage). Cette approche permet notamment de placer plusieurs types dans un module et donc de réaliser l'équivalent des classes amies de C++.
- En C++ au contraire, la classe est réalisée directement par une construction syntaxique qui englobe les notions de type, de description et de module. La classe peut être dégradée afin d'obtenir un module seul, par l'ajout du mot-clé **`static`** devant toutes les opérations.
- En Java, comme en C++, la classe est l'intégration des notions de type, de description et de module, mais il existe en plus une notion de module plus large (le paquetage) qui peut contenir plusieurs classes.

La séparation entre la spécification et la réalisation des classes participe à l'élévation du niveau d'abstraction. Les traits remarquables sont décrits dans les spécifications alors que les détails sont confinés dans les réalisations. L'occultation des détails de réalisation est appelée encapsulation.

L'encapsulation présente un double avantage. D'une part, les données encapsulées dans les objets sont protégées des accès intempestifs – ce qui permet de garantir leur intégrité – et d'autre part, les utilisateurs d'une abstraction ne dépendent pas de la réalisation de l'abstraction mais seulement de sa spécification, ce qui réduit le couplage dans les modèles.

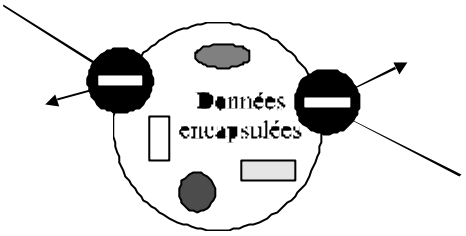


Figure 47 : Les données encapsulées dans un objet ne sont pas accessibles depuis l'extérieur.

Par défaut, les valeurs d'attributs d'un objet sont encapsulées dans l'objet et ne peuvent pas être manipulées directement par les autres objets. Toutes les interactions entre les objets sont effectuées en déclenchant les diverses opérations déclarées dans la spécification de la classe et accessibles depuis les autres objets.

Les règles de visibilité viennent compléter ou préciser la notion d'encapsulation. Ainsi, il est possible d'assouplir le degré d'encapsulation, mais aussi de protection, au profit de certaines classes utilisatrices bien particulières, désignées dans la spécification de la classe fournisseur. L'intérêt de briser l'encapsulation est par exemple de réduire le temps d'accès aux attributs en supprimant la nécessité de recourir à des opérations de type sélecteur.

Les trois niveaux distincts d'encapsulation couramment retenus correspondent à ceux proposés par le langage de programmation C++ :

- Le niveau le plus fort est appelé niveau privé ; la partie privée de la classe est alors totalement opaque et seuls les amis (au sens C++) peuvent accéder aux attributs placés dans la partie privée.
- Il est possible de relâcher légèrement le niveau d'encapsulation, en plaçant certains attributs dans la partie protégée de la classe. Ces attributs sont alors visibles à la fois pour les amis et les classes dérivées de la classe fournisseur. Pour toutes les autres classes, les attributs restent invisibles.
- Le niveau le plus faible est obtenu en plaçant les attributs dans la partie publique de la classe. Ceci revient à se passer de la notion d'encapsulation et à rendre visibles les attributs pour toutes les classes.

Le niveau de visibilité peut être précisé dans les représentations graphiques des classes au moyen des caractères **+**, **#** et **-**, qui correspondent respectivement aux niveaux public, protégé et privé.

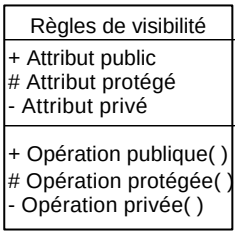


Figure 48 : Précision des niveaux de visibilité dans les représentations graphiques des classes.

L'exemple des nombres complexes décrit précédemment fournit une bonne illustration des vertus de l'encapsulation. Parce que la spécification des nombres complexes – qui regroupe entre autres les opérations d'addition, de soustraction, de multiplication et de division – n'est pas du tout affectée par le changement de représentation interne (de la notation polaire à la notation cartésienne), les objets qui utilisent des nombres complexes – et qui dépendent uniquement de la spécification – ne sont pas affectés par cette modification. Le diagramme suivant montre les deux représentations des nombres complexes. La partie publique de l'abstraction est identique dans les deux cas, mais la partie privée est différente.

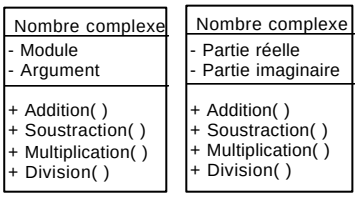


Figure 49 : La spécification n'est pas affectée par un changement de représentation interne.

L'encapsulation réduit le couplage au sein du modèle, favorise la modularité et facilite la maintenance des logiciels. L'encapsulation agit comme l'enceinte de confinement d'une centrale nucléaire : les défauts restent enfermés dans la classe concernée, ils ne se propagent pas dans tout le modèle.

Les critères d'encapsulation reposent sur la forte cohérence interne à l'intérieur d'une classe et sur le faible couplage entre les classes. Il ne suffit pas pour obtenir une bonne abstraction, de rassembler des données et de fournir des opérations pour la lecture et l'écriture de ces données. Une classe doit offrir une valeur ajoutée par rapport à la simple juxtaposition d'information. C'est le cas de la classe **`Nombre complexe`** qui fournit des opérations arithmétiques.

Les relations entre les classes

Les liens particuliers qui relient les objets peuvent être vus de manière abstraite dans le monde des classes : à chaque famille de liens entre objets correspond une relation entre les classes de ces mêmes objets. De même que les objets sont instances des classes, les liens entre objets sont instances des relations entre classes.

L'association

L'association exprime une connexion sémantique bidirectionnelle entre classes. Une association est une abstraction des liens qui existent entre les objets instances des classes associées. Le diagramme suivant représente des objets liés entre eux et les classes associées correspondantes. Les associations se représentent de la même manière que les liens. La distinction entre un lien et une association est opérée en fonction du contexte du diagramme.

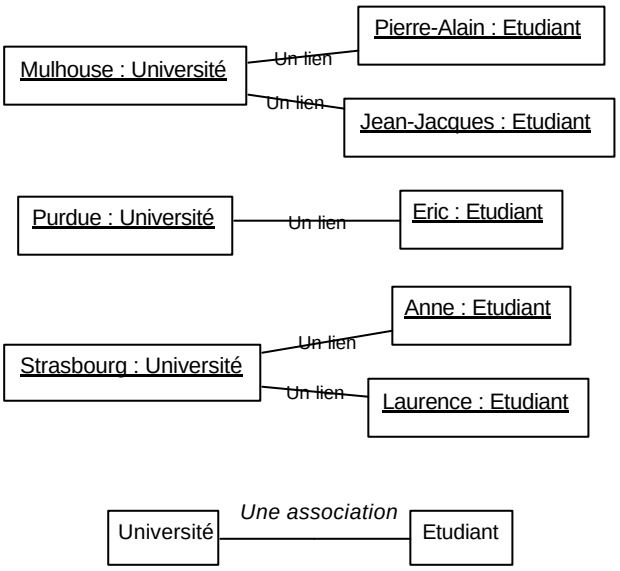


Figure 50 : Les liens entre les universités et les étudiants sont tous instances de l'association entre la classe ***`Université`*** *et la classe* ***`Etudiant`***.

Il faut noter que l'association est un concept de même niveau d'abstraction que les classes. L'association n'est pas contenue par les classes ou subordonnée aux classes ; elle est le reflet d'une connexion qui existe dans le domaine d'application.

Pour améliorer la lisibilité des diagrammes, l'association peut être décorée par une forme verbale active ou passive. Dans les exemples suivants, le sens de lecture est précisé par les signes **<** et **>**.

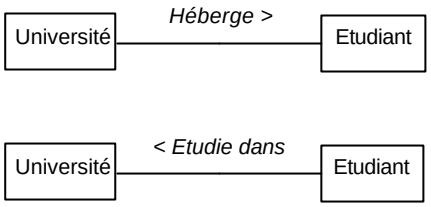


Figure 51 : Clarification de la nature d'une association par une forme verbale.

Il est possible de préciser le rôle d'une classe au sein d'une association : un nom de rôle peut être spécifié de part et d'autre de l'association. L'exemple suivant montre deux associations entre la classe **`Université`** et la classe **`Personne`**. Le diagramme précise que certaines personnes jouent le rôle d'étudiant, alors que d'autres personnes jouent le rôle d'enseignant. La deuxième association porte également un nom de rôle du côté de la classe **`Université`** pour indiquer que l'université joue le rôle d'employeur pour ses enseignants. Le nommage des rôles prend tout son intérêt lorsque plusieurs associations relient deux mêmes classes.

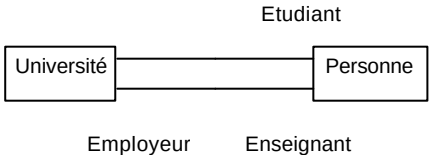


Figure 52 : Clarification du rôle des associations par une forme nominale.

Les rôles portent également une information de multiplicité qui précise le nombre d'instances qui participent à la relation. L'information de multiplicité apparaît dans les diagrammes de classes à proximité du rôle concerné.

Le tableau suivant résume les valeurs de multiplicité les plus courantes :

1	Un et un seul
0..1	Zéro ou un
M..N	De M à N (entiers naturels)
*****	De zéro à plusieurs

0..*	De zéro à plusieurs
1..*	D'un à plusieurs

Figure 53 : Valeurs de multiplicité conventionnelles.

La multiplicité peut également être exprimée au moyen de formules plus complexes. Par défaut, il n'y a pas de corrélation entre les valeurs ***** dans un même diagramme.

Le diagramme suivant donne un exemple de représentation des valeurs de multiplicité.

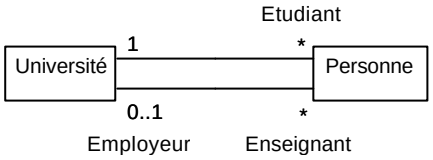


Figure 54 : Représentation du nombre d'instances qui participent aux associations.

Le diagramme précédent se lit de la manière suivante :

une université donnée regroupe de nombreuses personnes ; certaines jouent le rôle d'étudiant, d'autres le rôle d'enseignant. Un étudiant donné appartient à une seule université, un enseignant donné peut être en activité ou non.

L'agrégation

Une relation exprime une forme de couplage entre abstractions. La force de ce couplage dépend de la nature de la relation dans le domaine du problème. Par défaut, l'association exprime un couplage faible, les classes associées restant relativement indépendantes l'une de l'autre. L'agrégation est une forme particulière d'association qui exprime un couplage plus fort entre classes. Une des classes joue un rôle plus important que l'autre dans la relation. L'agrégation permet de représenter des relations de type maître et esclaves, tout et parties ou composé et composants.

Les agrégations représentent des connexions bidirectionnelles dissymétriques. Le concept d'agrégation est une notion purement logique, complètement indépendante des choix de représentation qui relèvent de la conception détaillée et non de la modélisation. Mathématiquement, l'agrégation est une relation transitive, non symétrique et réflexive.

L'exemple suivant montre qu'une personne peut s'occuper de plusieurs enfants. La relation est de nature dissymétrique dans le domaine considéré : l'adulte est responsable des enfants. La relation est également réflexive : certaines personnes

jouent le rôle de parent, d'autres jouent le rôle d'enfant. Une agrégation se représente comme une association, avec en plus un petit losange placé du côté de l'agrégat.

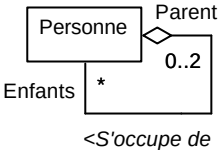


Figure 55 : Exemple d'agrégation réflexive.

L'agrégation favorise la propagation des valeurs d'attributs et des opérations de l'agrégat vers les composants. Lorsque la multiplicité de l'agrégat vaut **1**, la destruction de l'agrégat entraîne la destruction des composants.

L'exemple suivant présente le cas des voitures. Chaque voiture possède un moteur qui ne peut être partagé avec d'autres voitures. La destruction intégrale de la voiture entraîne la destruction du moteur.

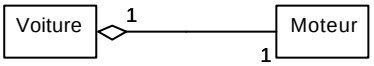


Figure 56 : La voiture est un tout qui contient un moteur.

Une agrégation non réflexive, dont la valeur de multiplicité vaut **1** du côté de l'agrégat, peut se réaliser par contenance physique.

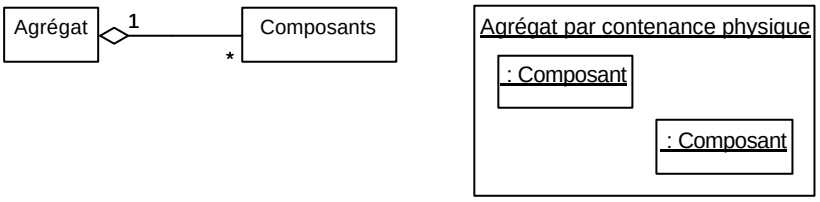


Figure 57 : Forme d'agrégation réalisable par contenance physique.

Lorsque cette multiplicité est supérieure à 1, la relation d'agrégation peut se réaliser par une forme idiomatique de type pointeur malin (*smart pointer*) : plusieurs pointeurs référencent le même objet tout en se synchronisant pour désallouer l'objet au moment où il ne sera plus référencé par aucun pointeur.

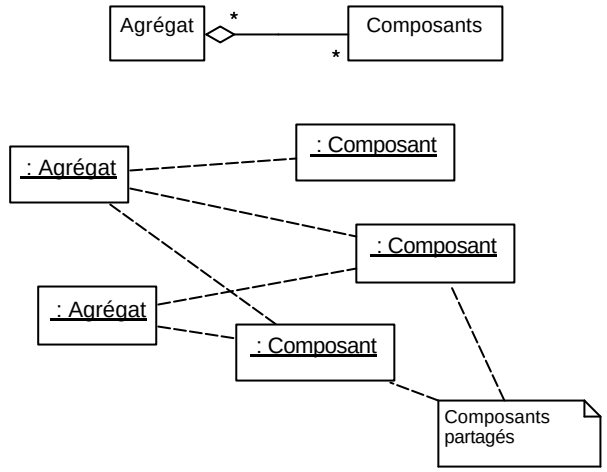


Figure 58 : Réalisation d'une agrégation par un pointeur malin.

Correspondances entre diagrammes de classes et diagrammes d'objets

Les diagrammes de classes et les diagrammes d'objets appartiennent à deux vues complémentaires du modèle. Un diagramme de classes montre une abstraction de la réalité, concentrée sur l'expression de la structure statique d'un point de vue général. Un diagramme d'objets représente plutôt un cas particulier, une situation concrète à un instant donné ; il exprime à la fois la structure statique et un comportement.

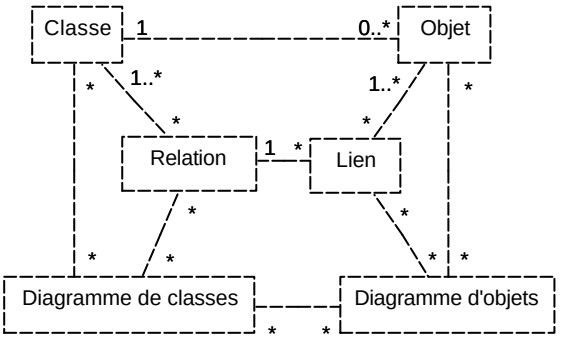


Figure 59 : Correspondances entre diagrammes de classes et diagrammes d'objets.

Les règles suivantes gouvernent la transition entre les deux types de diagramme :

- chaque objet est instance d'une classe et la classe de l'objet ne peut pas changer durant la vie de l'objet,
- certaines classes – appelées classes abstraites – ne peuvent pas être instanciées,
- chaque lien est instance d'une relation,
- les liens relient les objets, les relations relient les classes,
- un lien entre deux objets implique une relation entre les classes des deux objets,
- un lien entre deux objets indique que les deux objets se connaissent et qu'ils peuvent échanger des messages,
- les diagrammes d'objets qui contiennent des objets et des liens sont instances des diagrammes de classes qui contiennent des classes et des relations.

Les diagrammes de classes et les diagrammes d'objets doivent être cohérents les uns par rapport aux autres. Toutefois, il faut être conscient que le processus de modélisation objet n'est pas un processus linéaire, de sorte qu'il n'est pas souhaitable de construire un type de diagramme et ensuite d'en dériver intégralement l'autre type. Forcer la création d'un type de diagramme avant un autre limite la liberté de création. Pour prendre une image musicale, dissocier la construction des diagrammes d'objets de celle des diagrammes de classes, reviendrait à demander à un compositeur de considérer séparément la hauteur des notes et leur durée.

En pratique, les diagrammes d'objets et les diagrammes de classes se construisent en parallèle, par de nombreux allers et retours entre les deux représentations. Il n'y a aucune raison de définir les classes avant les objets. Il est vrai que chaque objet est instance d'une classe, mais la détermination de la classe peut très bien être postérieure à celle des objets. Le monde réel qui nous entoure contient des objets et non des classes : il semble donc naturel de trouver d'abord les objets puis d'en abstraire les classes. En fait, il n'y a pas de règle générale ; dans certains cas, la structure des classes est évidente, dans d'autres cas, les objets sont plus faciles à identifier que les classes.

Les hiérarchies de classes

Les hiérarchies de classes ou classifications permettent de gérer la complexité en ordonnant les objets au sein d'arborescences de classes d'abstraction croissante.

Généralisation et spécialisation

La généralisation et la spécialisation sont des points de vue portés sur les hiérarchies de classes.

La généralisation consiste à factoriser les éléments communs (attributs, opérations et contraintes) d'un ensemble de classes dans une classe plus générale appelée super-classe. Les classes sont ordonnées selon une hiérarchie ; une super-classe est une abstraction de ses sous-classes.

La généralisation est une démarche assez difficile car elle demande une bonne capacité d'abstraction. La mise au point d'une hiérarchie optimale est délicate et itérative. Les arbres de classes ne poussent pas à partir de leur racine. Au contraire, ils se déterminent en partant des feuilles car les feuilles appartiennent au monde réel alors que les niveaux supérieurs sont des abstractions construites pour ordonner et comprendre.

L'exemple suivant montre une hiérarchie des moyens de transport. La flèche qui symbolise la généralisation entre deux classes pointe vers la classe plus générale.

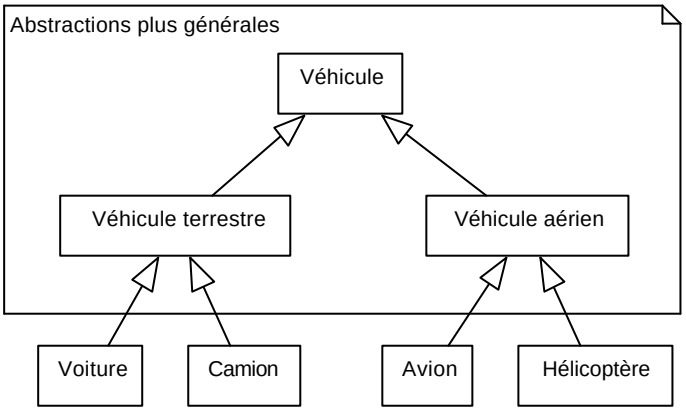


Figure 60 : Exemple de hiérarchie de classes construite par généralisation.

La spécialisation permet de capturer les particularités d'un ensemble d'objets non discriminés par les classes déjà identifiées. Les nouvelles caractéristiques sont représentées par une nouvelle classe, sous-classe d'une des classes existantes. La spécialisation est une technique très efficace pour l'extension cohérente d'un ensemble de classes.

L'exemple suivant montre une classification partielle des équipements de transmission, selon deux grandes familles : les systèmes continus et les systèmes discrets. Les dispositifs concrets sont ajoutés dans la hiérarchie par dérivation du parent le plus proche.

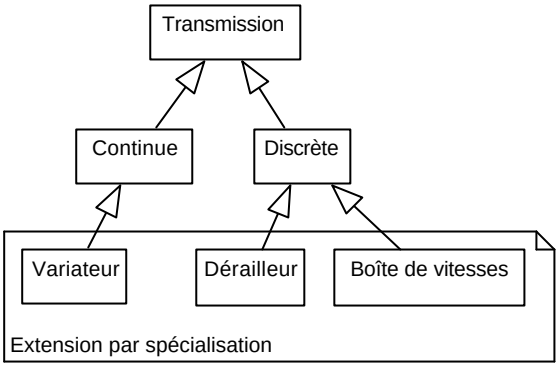


Figure 61 : Exemple d'extension par spécialisation.

La généralisation et la spécialisation sont deux points de vue antagonistes du concept de classification ; elles expriment dans quel sens une hiérarchie de classes est exploitée. Dans toute application réelle, les deux points de vue sont mis en œuvre simultanément. La généralisation est plutôt employée une fois que les éléments du domaine ont été identifiés afin de dégager une description détachée des solutions. La spécialisation, quant à elle, est à la base de la programmation par extension et de la réutilisation. Les nouveaux besoins sont encapsulés dans des sous-classes qui étendent harmonieusement les fonctions existantes.

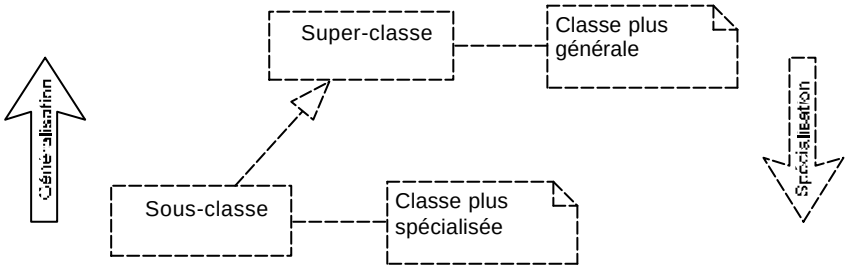


Figure 62 : La généralisation et la spécialisation offrent deux points de vue antagonistes sur une hiérarchie de classes.

L'élaboration d'une hiérarchie de classes demande de la part des développeurs des qualités et des compétences différentes selon le point d'entrée dans l'arborescence. L'identification des super-classes fait appel avant tout à la capacité d'abstraction, indépendamment de toutes connaissances techniques, alors que la réalisation des sous-classes demande surtout une expertise approfondie d'un domaine particulier.

En fait, la situation est tout à fait paradoxale ! Il est plutôt difficile de trouver les super-classes, mais les programmes écrits dans leurs termes sont plus simples à développer. Il est assez facile de trouver les sous-classes, mais difficile de les réaliser.

La généralisation ne porte aucun nom particulier ; elle signifie toujours : *est un* ou *est une sorte de*. La généralisation ne concerne que les classes, elle n'est pas instanciable en liens et, de fait, ne porte aucune indication de multiplicité. Dans l'exemple suivant, le lion est une sorte de carnivore et il n'est pas possible pour un lion d'être plusieurs fois un carnivore : un lion *est un* carnivore.

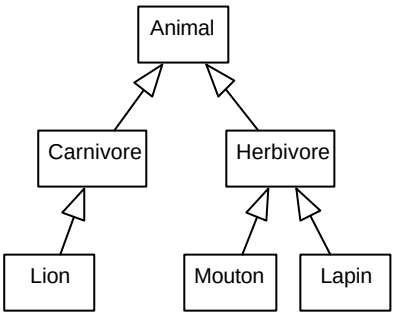


Figure 63 : La généralisation ne porte ni nom particulier ni valeur de multiplicité.

La généralisation est une relation non réflexive : une classe ne peut pas dériver d'elle-même.

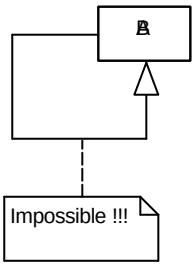


Figure 64 : La généralisation est une relation non réflexive.

La généralisation est une relation non symétrique : si une classe **B** dérive d'une classe **A**, alors la classe **A** ne peut pas dériver de la classe **B**.

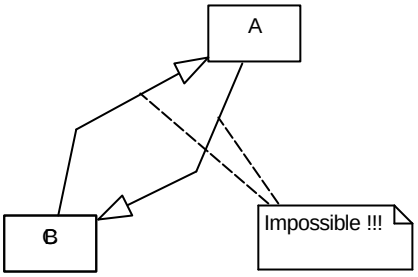


Figure 65 : La généralisation est une relation non symétrique.

La généralisation est par contre une relation transitive : si **C** dérive d'une classe **B** qui dérive elle-même d'une classe **A**, alors **C** dérive également de **A**.

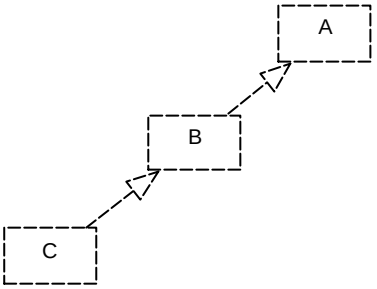


Figure 66 : La généralisation est une relation transitive.

Des ensembles aux classes

La notion de classe est très proche de la notion d'ensemble. La spécification d'une classe est une description abstraite, analogue à la description en compréhension d'un ensemble. Les objets instances d'une classe partagent des caractéristiques générales, exprimées dans la classe sous forme d'attribut, d'opération et de contrainte.

Ces caractéristiques constituent la propriété caractéristique de l'ensemble des instances. La propriété caractéristique d'un ensemble **X** est notée P (**X**). Dans les paragraphes suivants, le terme propriété caractéristique est appliqué directement à la classe et non à l'ensemble de ses instances. La figure suivante montre l'analogie entre une classe et un ensemble.

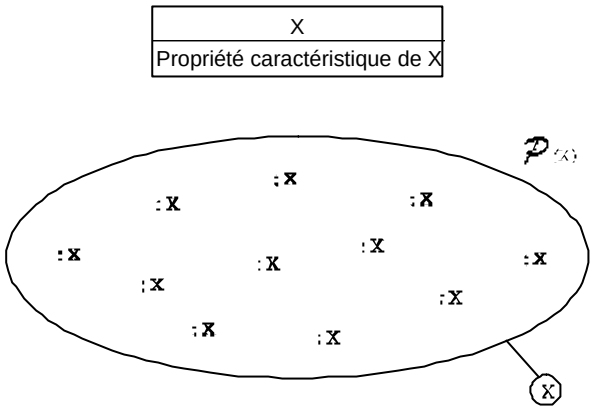


Figure 67 : Une classe donne une description abstraite d'un ensemble d'objets qui partagent des caractéristiques communes.

L'ensemble **X** peut être divisé en sous-ensembles, afin de distinguer par exemple des particularités supplémentaires partagées seulement par certains des éléments de **X**. Le diagramme suivant montre deux sous-ensembles de **X**, les ensembles **Y** et **Z**. Les propriétés caractéristiques des ensembles **Y** et **Z** sont des extensions de la propriété caractéristique de **X** :

$$P(X) \subseteq P(Y) \text{ et } P(X) \subseteq P(Z) \text{ et } P(Y) \cap P(Z) = P(X)$$

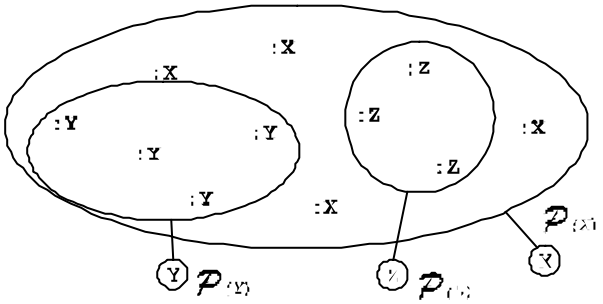


*Figure 68 : Les éléments des ensembles **Y** et **Z** sont d'abord des éléments de l'ensemble **X**. Les propriétés caractéristiques P (**Y**) et P (**Z**) englobent la propriété caractéristique P (**X**).*

Les classes et les sous-classes sont l'équivalent des ensembles et des sous-ensembles. La généralisation des classes correspond à la relation d'inclusion des ensembles. De ce fait, les objets instances d'une classe donnée sont décrits par la propriété caractéristique de leur classe, mais également par les propriétés

caractéristiques de toutes les classes parents de leur classe. Les sous-classes ne peuvent pas nier les propriétés caractéristiques de leurs classes parentes. La propriété caractéristique d'une sous-classe englobe la propriété caractéristique de toutes ses super-classes. Ce qui est vrai pour un objet instance d'une super-classe est vrai pour un objet instance d'une sous-classe. A l'image de l'inclusion dans les ensembles, la généralisation ordonne les objets au sein d'une hiérarchie de classes. Le diagramme suivant montre qu'il existe deux sortes d'éléments de **X** particuliers, respectivement décrits par les classes **Y** et **Z**.

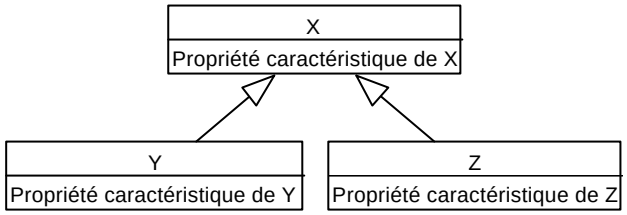


Figure 69 : La propriété caractéristique des sous-classes est une extension de la propriété caractéristique des super-classes.

Le diagramme suivant illustre un exemple concret. Il existe de très nombreux livres ; certains s'adressent tout particulièrement aux enfants, d'autres ont pour objectif l'enseignement. La classification n'est pas exhaustive : les livres qui ne s'adressent ni aux enfants, ni à l'enseignement ne sont pas distingués et appartiennent collectivement à la classe des livres. Les propriétés générales des livres, comme le nom de l'auteur ou le nombre de pages, sont définies dans la super-classe **`Livre`**. Chaque sous-classe reprend ces caractéristiques et peut en ajouter de nouvelles, comme la fourchette d'âge des lecteurs dans le cas du livre pour enfants.

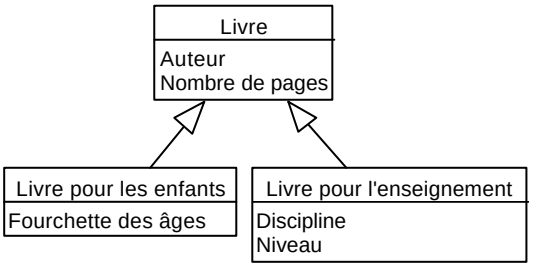


Figure 70 : Les livres pour enfants et ceux pour l'enseignement reprennent les caractéristiques générales des livres.

La généralisation – sous sa forme dite multiple – existe également entre arbres de classes disjoints. Dans l'exemple suivant, la classe **T** est issue de la combinaison des classes **Y** et **Z**.

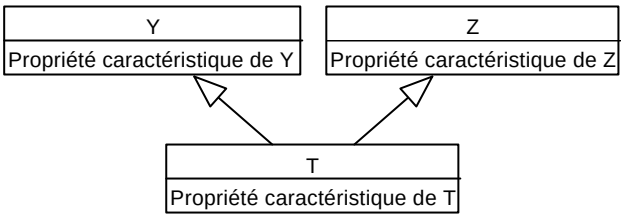


Figure 71 : Exemple de généralisation multiple à partir d'arbres de classes disjoints.

Dans le monde des ensembles, la situation précédente correspond à l'intersection de deux ensembles qui ne sont pas sous-ensembles du même sur-ensemble.

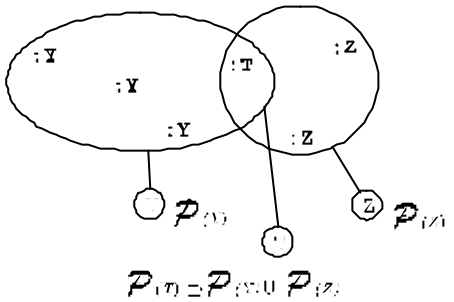


Figure 72 : Représentation ensembliste de la généralisation multiple entre classes sans ancêtre commun.

Le diagramme suivant illustre une situation particulièrement intéressante. Les ensembles **Y** et **Z** partitionnent l'ensemble **X**, de sorte que les objets de l'ensemble **X** appartiennent forcément à l'un de ses sous-ensembles. La propriété caractéristique P (**X**) ne décrit pas directement les éléments de **X**. P(**X**) est une description abstraite des éléments de **Y** et **Z** obtenue par une factorisation opérée sur P (**Y**) et P (**Z**).

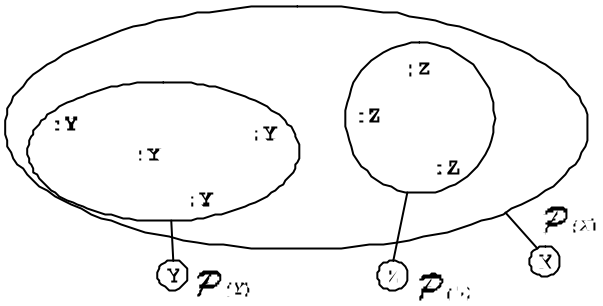


Figure 73 : L'ensemble ***X*** *est scindé en deux sous-ensembles disjoints* ***Y*** *et* ***Z*** *qui contiennent collectivement tous les objets de* ***X****. P (****X****) est une factorisation opérée sur P (****Y****) et P (****Z****).*

Dans le monde des classes, la situation précédente met en jeu une classe abstraite, c'est-à-dire une classe qui ne donne pas directement des objets. Elle sert en fait de spécification plus abstraite pour des objets instances de ses sous-classes. Le principal intérêt de cette démarche est de réduire le niveau de détails dans les descriptions des sous-classes. Le nom d'une classe abstraite est en italique dans les diagrammes de classes.

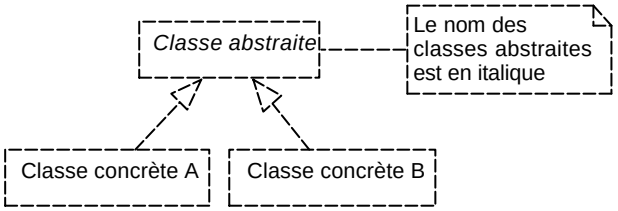


Figure 74 : Représentation graphique d'une classe abstraite.

Les classes abstraites facilitent l'élaboration de logiciels génériques, facilement extensibles par sous-classement. L'ensemble des mécanismes qui servent de charpente pour les fonctions des applications est construit à partir des éléments généraux fournis par les classes abstraites. Les spécificités et les extensions sont encapsulées dans des sous-classes concrètes.

Dans les exemples précédents, les sous-ensembles **Y** et **Z** sont disjoints. Dans l'exemple suivant, l'intersection de **Y** et **Z** est non nulle et définit l'ensemble **T**. **T** regroupe l'ensemble des objets qui appartiennent à la fois à la classe **Y** et à la classe **Z**. **T** est simultanément sous-ensemble de **Y** et de **Z**. La propriété caractéristique de l'ensemble **T** est l'union des propriétés caractéristiques des ensembles **Y** et **Z** et de la propriété caractéristique de **T** lui-même.

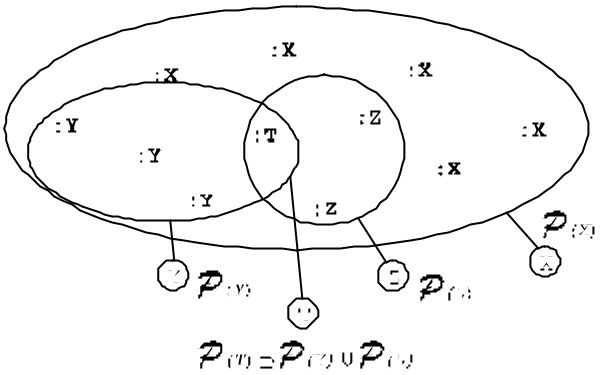


*Figure 75 : La propriété caractéristique de l'ensemble **T** – intersection des ensembles **Y** et **Z** – contient l'union des propriétés caractéristiques des ensembles **Y** et **Z**.*

En termes de classes, la situation précédente se représente de la manière suivante :

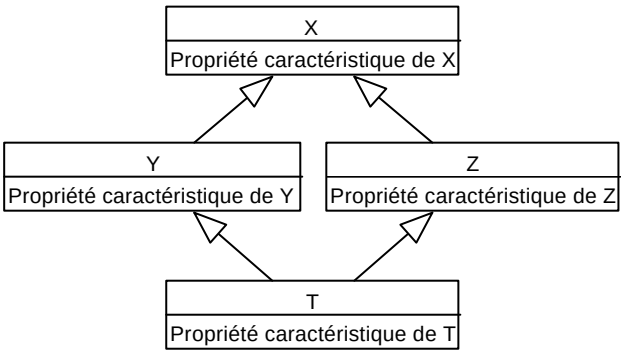


*Figure 76 : Exemple de généralisation en losange. Les objets instances de la classe **T** sont décrits simultanément par les classes **X**, **Y**, **Z** et **T**.*

Les objets de la classe **T** sont décrits une seule fois par la propriété caractéristique de **X**. Le nombre de branches de la généralisation n'est pas significatif pour la propagation des propriétés caractéristiques et ainsi **T** ne possède pas deux fois la propriété caractéristique de **X**.

Les objets sont instances d'une classe ou ne le sont pas ; il n'est pas possible d'être plusieurs fois instance de la même classe.

De la difficulté de classer

La classification n'est pas toujours une opération triviale. En effet, la détermination des critères de classification est difficile et dans certains cas il n'est pas possible de se déterminer. En 1755, bien avant UML, Jean-Jacques Rousseau, dans l'ouvrage *Discours sur l'origine et les fondements de l'inégalité parmi les hommes*, évoque le problème de la classification dans les termes suivants :

Chaque objet reçut d'abord un nom particulier, sans égard aux genres et aux espèces (...). Si un chêne s'appelait A, un autre chêne s'appelait B ; car la première idée que l'on tire de deux choses, c'est qu'elles ne sont pas la même ; et il faut souvent beaucoup de temps pour observer ce qu'elles ont de commun : de sorte que plus les connaissances étaient bornées, et plus le dictionnaire devint étendu. L'embarras de toute cette nomenclature ne put être levé facilement, car, pour ranger les êtres sous des dénominations communes et génériques, il en fallait connaître les propriétés et les différences ; il fallait des observations et des définitions, c'est-à-dire de l'histoire et de la métaphysique, beaucoup plus que les hommes de ce temps-là n'en pouvaient avoir.

Les classifications doivent avant tout bien discriminer les objets. Les bonnes classifications sont stables et extensibles. Il arrive qu'elles comportent des exceptions inclassables selon les critères retenus. Le grand panda, par exemple, fait partie de la famille des ours alors que le petit panda est plus proche des ratons laveurs. L'ornithorynque appartient à la famille des mammifères tout en étant ovipare.

Les classifications s'effectuent selon des critères dépendant du point de vue. Il n'y a donc pas une classification, mais des classifications, chacune adaptée à un usage donné.

Ainsi, pour les animaux, de nombreux critères peuvent être retenus :

- la station,
- le type de nourriture,
- la protection.

La détermination des critères pertinents et de l'ordre dans lequel ils doivent être appliqués n'est pas toujours facile. Une fois les critères arrêtés, il faut les suivre de manière cohérente et uniforme selon l'ordre déterminé.

L'ordre d'application des critères est souvent arbitraire, et conduit à des décompositions covariantes qui se traduisent par des parties de modèle isomorphes.

Dans l'exemple suivant, le critère de la station a été appliqué avant le critère de la nourriture. Sans informations supplémentaires, il est bien difficile de dire pourquoi ce choix a été effectué.

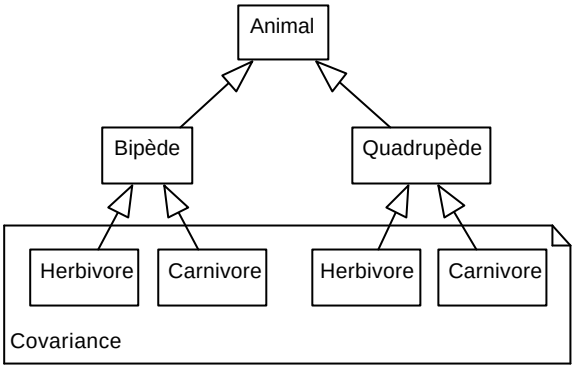


Figure 77 : Première forme de décomposition covariante.

Les animaux peuvent tout aussi bien être spécialisés d'abord par rapport à leur type de nourriture.

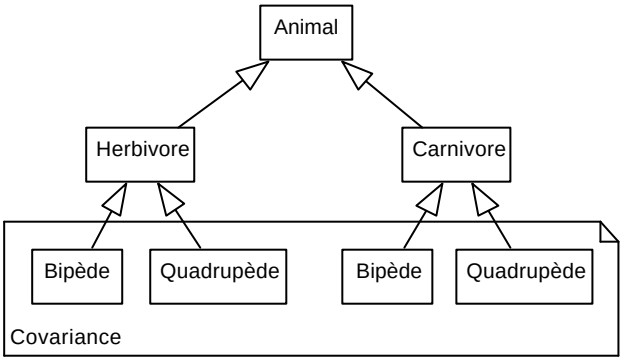


Figure 78 : Deuxième forme de décomposition covariante.

En fait, aucune des solutions précédentes n'est satisfaisante, car le phénomène de covariance induit des points de maintenance multiples dans le modèle.

La généralisation multiple apporte une solution élégante pour la construction de classifications comportant des critères indépendants, difficiles à ordonner. Les critères indépendants déterminent différentes dimensions de spécialisation et les classes concrètes sont obtenues par produit cartésien de ces différentes dimensions.

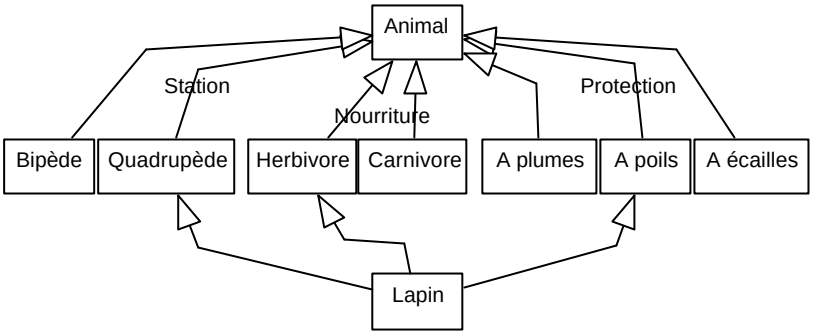


Figure 79 : Exemple de réduction de la covariance au moyen de la généralisation multiple.

Dans certains cas, la covariance existe du fait même de la nature du domaine de l'application. L'exemple suivant illustre ce propos : un pilote de moto d'enduro doit posséder une licence d'enduro et un pilote de moto de cross doit posséder une licence de cross. Cette forme de covariance n'est pas réductible car elle concerne deux hiérarchies distinctes.

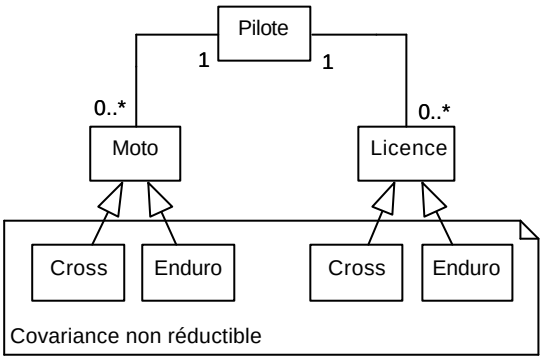


Figure 80 : Exemple de covariance non réductible.

Les classifications doivent également comporter des niveaux d'abstractions équilibrés. Dans l'exemple suivant, la décomposition n'est pas effectuée de manière homogène. Les deux premières sous-classes spécialisent selon la fonction du véhicule alors que la troisième sous-classe correspond à une marque de moto.

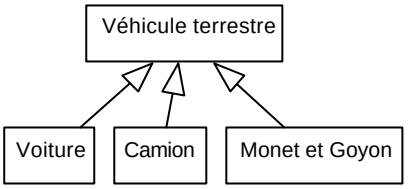


Figure 81 : Exemple de classification déséquilibrée.

L'énumération de toutes les marques n'est pas non plus une bonne idée car cela créé énormément de sous-classes. Il vaut mieux favoriser le critère le plus global car c'est lui qui donnera des sous-classes plus extensibles. En règle générale, il vaut mieux limiter le nombre de sous-classes à chaque niveau hiérarchique, quitte à augmenter le nombre d'objets par classe et à se réserver les attributs d'objets pour qualifier finement les objets.

La dérive vers un trop grand nombre de classes est illustrée dans l'exemple suivant. D'une part, le critère de la couleur est trop précis pour déterminer une classe et d'autre part, il génère trop de classes. Enfin, du fait du lien statique entre classe et instance, le modèle ne permet pas de changer la couleur d'une voiture ! Il faut en fait que le critère de génération d'une classe soit statique. La couleur doit être un attribut des véhicules, pas un critère de spécialisation.

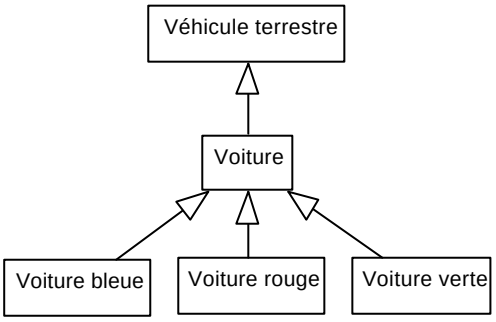


Figure 82 : Exemple de mauvais critère pour la détermination des classes.

La forme classique de la relation de généralisation de classes introduit un couplage statique très fort dans le modèle, non mutable, comme le lien entre un objet et sa classe. De ce point de vue, la généralisation n'est pas adaptée pour représenter les métamorphoses. L'exemple suivant ne représente pas correctement le cas du papillon qui passe successivement du stade de chenille, au stade de chrysalide puis au stade de lépidoptère.

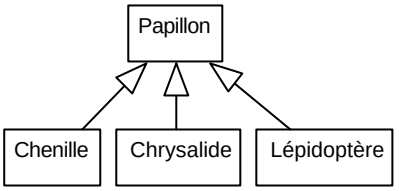


Figure 83 : Le modèle ci-dessus n'est pas adapté pour représenter les papillons, car la relation qui lie un objet à sa classe n'est pas mutable.

La solution pour la représentation des papillons consiste à sortir l'élément mutable. Dans notre cas, l'apparence d'un papillon donné est traduite par un lien vers un objet spécifique qui décrit son stade.

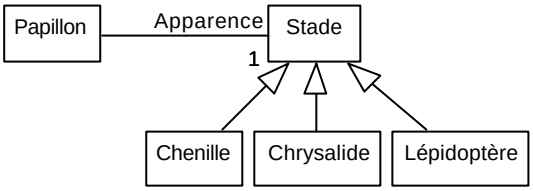


Figure 84 : Exemple d'extraction de l'élément mutable. L'objet qui matérialise l'apparence du papillon peut changer pendant son existence.

L'approche générale pour dynamiser une classification consiste, comme dans le cas de la mutabilité, à extraire le ou les éléments sujets à spécialisation. Cette approche permet également la classification multiple, au moyen d'une multiplicité de valeur illimitée.

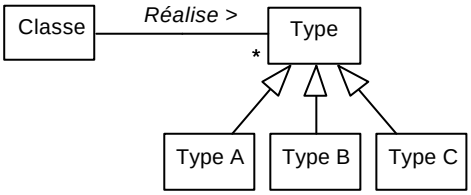


Figure 85 : Forme générale de la classification dynamique.

Le diagramme suivant montre un exemple de réalisation d'une classification selon la forme précédente, mais sans généralisation. Sur chaque livre est collée une gommette dont la couleur permet par de distinguer le propriétaire ou de préciser la nature de l'ouvrage.

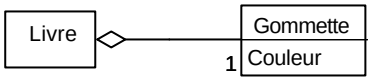


Figure 86 : Exemple de classification réalisée par un système de gommettes de couleur.

L'héritage

Il existe de nombreuses manières de réaliser la classification. En programmation objet, la technique la plus utilisée repose sur l'héritage entre classes.

Principe général

L'héritage est une technique offerte par les langages de programmation pour construire une classe à partir d'une ou plusieurs autres classes, en partageant des attributs, des opérations et parfois des contraintes, au sein d'une hiérarchie de classes. Les classes enfants héritent des caractéristiques de leurs classes parents ; les attributs et les opérations déclarés dans la classe parent, sont accessibles dans la classe enfant, comme s'ils avaient été déclarés localement.

L'héritage est utilisée pour satisfaire deux besoins distincts : la classification et la construction. Cette ambivalence de la relation d'héritage, qui peut à la fois classer et construire, est la source de beaucoup d'incohérences de programmation. Il en est de l'héritage comme de la conduite automobile ; il faut se tenir ni trop à droite ni trop à gauche, mais bien au milieu de la voie, sinon le risque d'accident est élevé. Il faut apprendre à utiliser correctement l'héritage, comme il faut apprendre à conduire les voitures.

En programmation, avec un langage objet comme C++, la classification se réalise très souvent par une relation d'héritage entre la classe plus générale et la classe plus spécifique. L'héritage propage les caractéristiques de la classe parent dans les classes enfants, de sorte que plusieurs classes peuvent partager une même description. De ce point de vue, l'héritage permet une description économique d'un ensemble de classes reliées par une relation de classification.

De nombreux programmeurs effectuent un amalgame entre la notion de classification impliquée par l'héritage et la composition virtuelle qui en résulte. Ces programmeurs recherchent avant tout une économie d'expression à court terme ; ils ont pris l'habitude d'intégrer des composants dans un composé en exploitant la propagation des caractéristiques réalisée automatiquement par la relation d'héritage. La construction de composants par héritage est une technique de programmation parfaitement respectable, à condition de clairement matérialiser le fait dans le programme. Lorsque le programmeur construit par héritage, il doit indiquer que la relation d'héritage concernée est utilisée pour la construction et non pour la classification. Le langage C++, par exemple, permet de distinguer

l'héritage pour la classification, de l'héritage pour la construction, par l'usage des mots-clés **`public`** et **`private`**.

```
class Truc :  private   Parent_Pour_Construction,
         public  Parent_Pour_Classification
{
...
} ;
```

Figure 87 : Distinction entre héritage pour la classification et héritage pour la construction.

La distinction doit être opérée visuellement dans les diagrammes de classes, comme le montre l'exemple suivant : un sémaphore n'est ni une liste chaînée, ni un compteur, mais peut se construire à partir de ces deux composants.

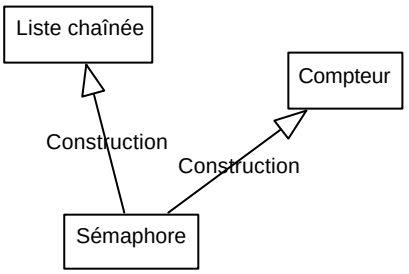


Figure 88 : Représentation graphique de l'héritage pour la construction.

Cette distinction se fait essentiellement en conception où, sauf indication contraire, l'héritage représente une relation de généralisation plutôt qu'une relation de composition.

L'héritage est entaché de contingences de réalisation et, en particulier, l'héritage n'effectue pas une union des propriétés caractéristiques des classes, mais plutôt une somme de ces propriétés. Ainsi, certaines caractéristiques peuvent être indûment dupliquées dans les sous-classes. L'héritage multiple doit donc être manié avec beaucoup de précautions car les techniques de réalisation de l'héritage peuvent induire des problèmes de collision de noms lors de la propagation des attributs et des opérations des classes parents vers les sous-classes.

L'exemple suivant illustre un conflit de nom, conséquence d'une relation d'héritage multiple réalisée par copie. Les super-classes définissent toutes les deux un attribut **A** de sorte que la classe **Z** possède deux attributs **A**. Si **A** représente exactement la même notion dans les classes **X** et **Y**, il n'y a pas de raison de propager deux attributs dans la sous-classe. En revanche, si **A** désigne

deux propriétés différentes, il serait judicieux d'en renommer une afin de pouvoir les distinguer. Il n'existe pas de réponse toute faite à ce problème : les langages objet diffèrent dans leur manière de le traiter.

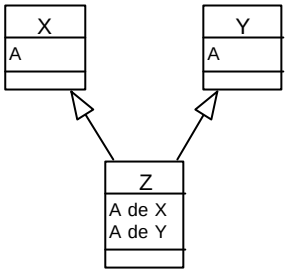


Figure 89 : Exemple de collision de noms. Le nom ***A*** *est défini dans les deux super-classes* ***X*** *et* ***Y****.*

Le conflit précédent apparaît également dans les formes de généralisation en losange, à la différence près qu'il n'y a vraiment pas de raisons de dupliquer l'attribut **A** dans la classe **Z**, étant donné que l'attribut **A** est unique dans la classe **T**. **Z** est une sorte de **T**, et non plusieurs fois une sorte de **T** ! Ici aussi, chaque langage objet apporte sa propre réponse à ce type de problème.

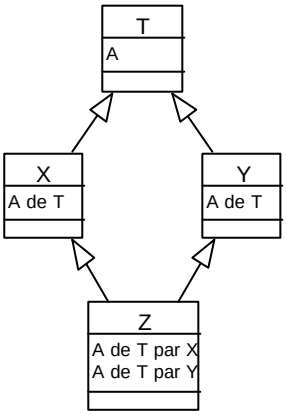


Figure 90 : Exemple de collision de noms dans le cas d'une forme d'héritage en losange.

Cette situation est suffisamment ennuyeuse pour que certains langages objet, comme Java ou Ada 95, n'offrent pas d'héritage multiple. Dans la pratique, l'héritage multiple peut être employé sans trop de soucis lorsque sa mise en œuvre a accompagné l'élaboration du modèle depuis le début. Par contre, il y a fort peu de chances pour que l'héritage multiple soit la manière d'effectuer une

fusion entre deux ensembles de classes construits de manière totalement indépendante. En conclusion : l'usage de l'héritage multiple doit être anticipé !

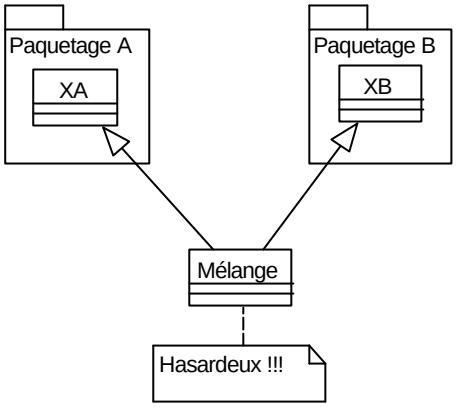


Figure 91 : L'héritage multiple n'est pas adapté pour la construction d'une classe par fusion de plusieurs classes issues de paquetages élaborés de manière indépendante.

La délégation

L'héritage n'est pas une nécessité absolue et peut toujours être remplacé par la délégation. La délégation présente l'avantage de réduire le couplage dans le modèle : d'une part, le client ne connaît pas directement le fournisseur, et d'autre part, le fournisseur peut être modifié en cours de route. Cette approche permet la mise en œuvre de la généralisation multiple avec les langages qui ne possèdent que l'héritage simple. Le diagramme suivant illustre le mécanisme de délégation ; le client communique avec une interface qui propage les questions à un ou plusieurs délégués.

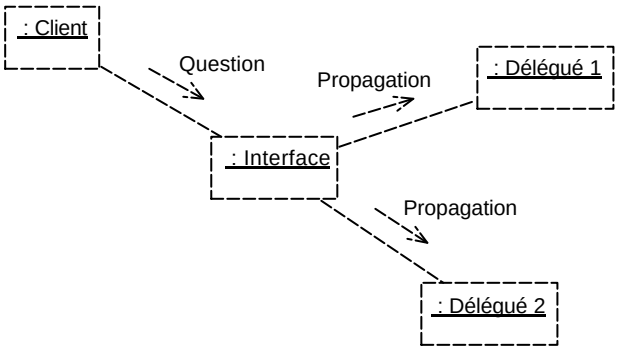


Figure 92 : Exemple de mécanisme de délégation. L'interface découple le client et les fournisseurs.

La délégation permet également de contourner le problème de la covariance, évoqué plus haut, au détriment toutefois de la propagation automatique des caractéristiques des super-classes vers les sous-classes. Le diagramme suivant illustre une construction à base de délégation qui peut remplacer la généralisation multiple.

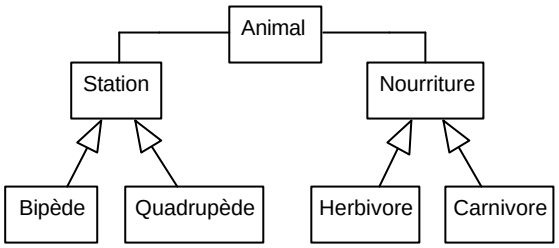


Figure 93 : Exemple de réduction de la covariance par la délégation.

Dans le système de fenêtres X Window, plus précisément dans la couche des *Intrinsics*, l'héritage est entièrement simulé à la main, grâce à la mise en œuvre de structures de données qui désignent les structures de données des parents. Les diagrammes suivants illustrent l'exemple de Douglas A. Young[7] pour la réalisation de l'héritage lors de la construction de *widgets*.

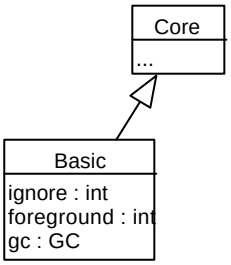


Figure 94 : Exemple d'héritage entre widgets dans le système de fenêtres X Window.

Le système X Window est entièrement réalisé en C, sans support direct pour l'héritage. Le diagramme suivant montre comment l'héritage est simulé à la main, en incorporant une description de la classe parent **`CoreClassPart`** dans la classe dérivée **`BasicClassRec`**. La classe **`BasicPart`** regroupe les variables

[7] Young D. A. 1990, *The X Window System, Programming and Applications with Xt*. Prentice Hall, Englewood Cliffs, New Jersey, pp 340-341.

d'instance et contient une référence vers la description de la classe **`BasicClassRec`**.

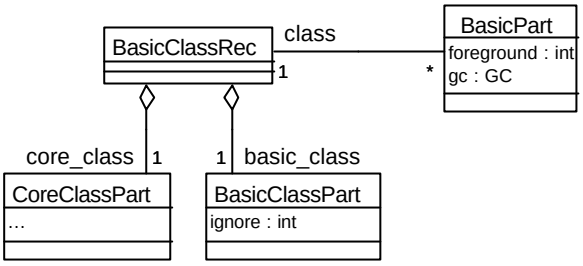


Figure 95 : Exemple de réalisation de l'héritage à la main par incorporation de la classe parent dans la classe dérivée.

Le principe de substitution

La classification propage l'état, le comportement et les contraintes. Il n'y a pas de demi-mesure : toutes les propriétés de la classe parent sont valables intégralement pour la classe enfant. Le besoin d'hériter partiellement est le signe que la relation d'héritage considérée ne réalise pas vraiment une relation de classification.

Le principe de substitution, énoncé originellement par Liskow[8], permet de déterminer si une relation d'héritage est bien employée pour la classification. Le principe de substitution affirme que :

il doit être possible de substituer n'importe quel objet instance d'une sous-classe à n'importe quel objet instance d'une super-classe sans que la sémantique du programme écrit dans les termes de la super-classe ne soit affectée.

L'illustration suivante montre un programme – symbolisé par un rectangle – qui fait référence à des objets de la classe parent. Si le principe de substitution est vérifié, tout objet instance de la classe **`CP`** doit pouvoir être remplacé par un objet instance de la classe **`CE`**, sans que cela n'affecte le programme écrit dans les termes de la classe **`CP`**.

[8] Liskow B. 1987, *Proceedings of OOPSLA'87.* ACM SIGPLAN Notices 23 (5), pp 17-34.

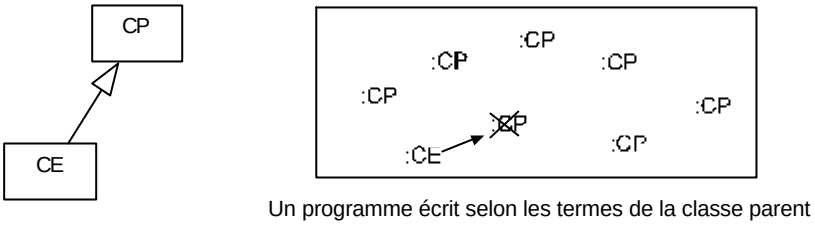


Figure 96 : Illustration du principe de substitution.

La notion de généralisation implique que la propriété caractéristique de la super-classe est incluse dans la propriété caractéristique de la sous-classe. Or, les compilateurs de certains langages ne sont pas capables de vérifier intégralement que cette condition est satisfaite dans le cas de l'héritage, car la syntaxe seule ne suffit pas toujours pour exprimer l'ensemble des propriétés. Les attributs et les opérations sont propagés automatiquement, mais les contraintes ne le sont pas nécessairement. Souvent, une contrainte est traduite par une forme de code particulière, implantée dans la réalisation d'une opération. Comme les langages objet permettent la redéfinition des opérations dans les sous-classes, les programmeurs peuvent involontairement introduire des incohérences entre la spécification d'une super-classe et la réalisation dans une des sous-classes. Ces incohérences concernent principalement les contraintes exprimées de manière programmée et non déclarative. L'adhérence au principe de substitution garantit qu'une relation d'héritage entre classes correspond bien à une généralisation. Malheureusement, la mise en œuvre du principe de substitution est du ressort du programmeur et non du compilateur et, comme le programmeur est humain, il n'est pas à l'abri d'une erreur. Sans respect du principe de substitution, le polymorphisme décrit dans le paragraphe suivant ne peut être mis en œuvre.

Le polymorphisme

Le terme polymorphisme décrit la caractéristique d'un élément qui peut prendre plusieurs formes, comme l'eau qui se trouve à l'état solide, liquide ou gazeux. En informatique, le polymorphisme désigne un concept de la théorie des types, selon lequel un nom d'objet peut désigner des instances de classes différentes issues d'une même arborescence.

Principe général

Les interactions entre objets sont écrites selon les termes des spécifications définies, non pas dans les classes des objets, mais dans leurs super-classes.

Ceci permet d'écrire un code plus abstrait, détaché des particularismes de chaque classe, et d'obtenir des mécanismes suffisamment généraux pour être encore valides dans le futur, quand seront créés de nouvelles classes.

Le terme polymorphisme désigne dans ce cas particulier le polymorphisme d'opération, c'est-à-dire la possibilité de déclencher des opérations différentes en réponse à un même message. Chaque sous-classe hérite de la spécification des opérations de ses super-classes, mais a la possibilité de modifier localement le comportement de ces opérations, afin de mieux prendre en compte les particularismes liés à un niveau d'abstraction donné. De ce point de vue, une opération donnée est polymorphe puisque sa réalisation peut prendre plusieurs formes.

Le polymorphisme est un mécanisme de découplage qui agit dans le temps. Les bénéfices du polymorphisme sont avant tout récoltés durant la maintenance. Le polymorphisme n'influence pas l'analyse, mais dépend de l'analyse : sa mise en œuvre efficace repose sur l'identification de mécanismes abstraits, applicables de manière uniforme à des objets instances de sous-classes différentes. Il ne faut pas penser l'analyse en termes de polymorphisme, il faut penser l'analyse en termes d'abstraction et ainsi, par effet de bord bénéfique de cette abstraction, rendre possible le polymorphisme.

Application

Le diagramme suivant représente une collection polymorphe, le zoo. Le zoo contient de nombreux animaux qui peuvent être soit des lions, soit des tigres, soit des ours. Le nom **`Animal`**, connu de la classe **`Zoo`**, décrit collectivement toutes les sortes d'animaux. Le logiciel, écrit au niveau d'abstraction du zoo, n'a pas besoin de connaître les détails propres à chaque animal.

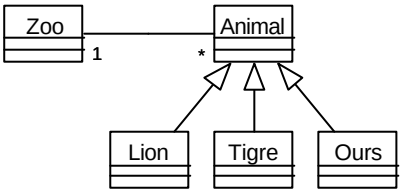


Figure 97 : Exemple de collection polymorphe.

Les animaux de l'exemple précédent savent tous dormir, mais chaque race a ses habitudes particulières. La spécification de l'animal dit que les animaux peuvent dormir. Les sous-classes particularisent l'opération **`Dormir()`** selon les goûts de chaque race. Le diagramme suivant montre comment chaque race d'animaux à l'habitude de dormir.

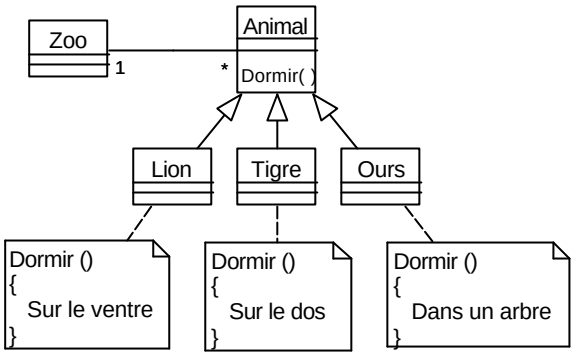


Figure 98 : Exemple de spécialisation d'une opération.

Les mécanismes généraux écrits selon la spécification du zoo n'ont pas besoin de connaître les goûts particuliers de chaque genre d'animal pour invoquer l'opération **`Dormir()`**. Ainsi, le soir venu, le gardien se promène au travers du zoo et informe chaque animal qu'il est temps de dormir.

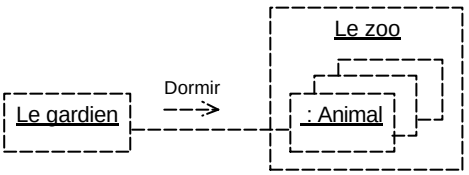


Figure 99 : Envoi du message ***`Dormir()`*** *à tous les animaux du zoo.*

En termes plus informatiques, ceci revient à visiter la collection **`Zoo`** en utilisant éventuellement un itérateur, et à envoyer le message **`Dormir`** à chaque animal.

Un itérateur est un objet associé à une collection qui permet d'en visiter tous ses éléments sans en dévoiler sa structure interne.

L'itérateur est dit actif lorsque le contrôle de l'itération est laissé à l'utilisateur, au moyen des quatre opérations suivantes :

- *Initialiser* qui permet de prendre en compte des éléments présents à un instant donné dans la collection,
- *Suivant* qui permet le passage à l'élément suivant,
- *Valeur* qui retourne l'élément courant,
- *Terminé* qui est vrai lorsque tous les éléments ont été visités.

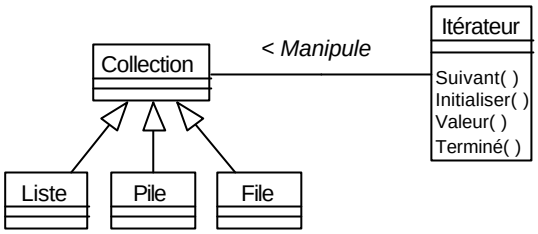


Figure 100 : Exemple d'itérateur actif.

Le fragment de code suivant illustre l'usage d'un itérateur sur le zoo.

La variable **`UnAnimal`** est polymorphe : elle peut contenir n'importe quel animal retourné par la fonction **`Visite.Valeur()`**. L'envoi du message **`Dormir()`** à l'animal contenu par la variable **`UnAnimal`** déclenche une manière spécifique de dormir qui dépend en fait de la sous-classe de l'animal.

```
Visite : Itérateur ;
UnAnimal : Animal ; -- variable polymorphe
...
Visite.Initialiser(leZoo);
while not Visite.Terminé()
loop
    UnAnimal := Visite.Valeur () ;
    UnAnimal.Dormir();
    Visite.Suivant();
end loop;
```

Figure 101 : Exemple de code pour endormir tous les animaux du zoo.

Le mécanisme qui endort les animaux du zoo est indépendant des animaux qui se trouvent réellement dans le zoo à un moment donné : l'effectif de chaque espèce n'est pas inscrit dans l'algorithme qui se contente d'exploiter un itérateur sur une collection. Le mécanisme ne dépend pas non plus de la sous-classe précise de l'animal courant : si de nouveaux animaux sont ajoutés dans le zoo, il n'est pas nécessaire de modifier le code qui endort les animaux existants pour endormir les nouveaux arrivants. Les particularités des nouveaux animaux sont encapsulées dans leur classe qui est rajoutée dans le modèle par dérivation de la classe **`Animal`** déjà existante. Pour prendre en compte un nouvel animal, il suffit donc de créer une sous-classe, de réaliser l'opération **`Dormir()`** dont il hérite, et éventuellement de recompiler.

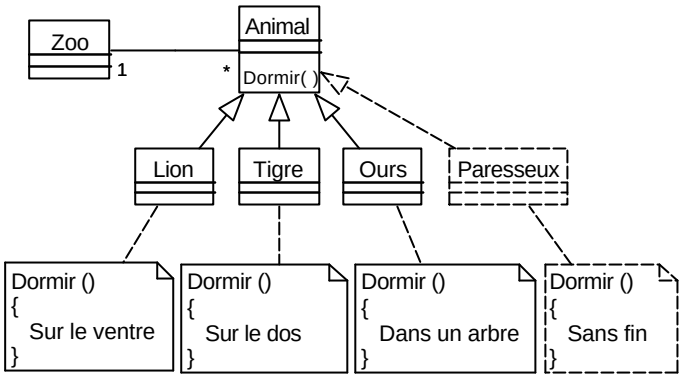


Figure 102 : Exemple de prise en compte d'un nouvel animal par ajout d'une sous-classe.

La recherche automatique du code à exécuter est le fruit de la liaison dynamique. Dans les langages traditionnels comme Pascal, le choix d'une opération est une activité complètement statique, effectuée à la compilation en fonction du type des variables. Dans les langages à liaison dynamique, le déclenchement d'une opération requiert une activité durant l'exécution pour retrouver la réalisation de l'opération qui correspond au message reçu. Sans liaison dynamique, le code précédent devrait être transformé de la façon suivante :

```
Visite : Itérateur ;
UnAnimal : Animal ; -- variable polymorphe
...
Visite.Initialiser(leZoo);
while not Visite.Terminé()
loop
    UnAnimal := Visite.Valeur () ;

    case UnAnimal.Classe ()
        when Lion
            -- Dormir sur le ventre
        When Tigre
            -- Dormir sur le dos
        When Ours
            -- Dormir dans un arbre
        When Paresseux
            -- Dormir sans fin
    end case;

    Visite.Suivant();
end loop;
```

Figure 103 : Sans liaison dynamique, le programmeur doit effectuer lui-même le choix de l'opération qui correspond au message **`Dormir()`**.

Cette solution sans liaison dynamique présente le désavantage d'introduire de nombreux points de maintenance dans le code. Chaque fois qu'un nouvel animal est ajouté dans le zoo, il faut ajouter une branche dans l'instruction **`case`** pour prendre en compte les spécificités du nouvel arrivant.

Le polymorphisme supprime ces points de maintenance et du même coup réduit considérablement l'usage des instructions à branchements multiples dans les mécanismes qui n'ont pas besoin de connaître explicitement la classe d'un objet. Les opérations du type **`Class_Of`**, comme celle employée dans l'exemple précédent (**`case UnAnimal.Classe()`**), qui retournent la classe d'un objet, doivent alors être utilisées avec circonspection, car leur usage va à l'encontre du polymorphisme. Un mécanisme écrit selon les termes d'une super-classe doit pouvoir ignorer les détails précisés dans les sous-classes.

L'exemple précédent montre également qu'il est possible d'employer un langage à liaison statique plutôt que dynamique après une analyse objet, mais que l'effort de réalisation est supérieur puisqu'il faut organiser manuellement le déclenchement des opérations.

Exploitation du principe de substitution

L'exemple précédent ne fonctionne que si chaque animal comprend le message **`Dormir()`**. Les mécanismes qui mettent en œuvre le polymorphisme manipulent des objets au travers des spécifications de leurs super-classes. Dans le cas d'un langage typé, le compilateur peut vérifier statiquement que les messages seront compris par un objet, soit parce qu'une opération est déclarée dans la classe même de l'objet, soit parce que cette opération est héritée d'une des super-classes. Cette vérification syntaxique ne suffit toutefois pas à garantir un bon comportement polymorphe ; il faut en plus que les réalisations des opérations dans les sous-classes soient conformes aux spécifications données dans les super-classes, en particulier aux contraintes qui les accompagnent. Pour que le polymorphisme fonctionne effectivement, il faut – au-delà de la syntaxe seule – que le principe de substitution soit vérifié.

L'exemple suivant illustre une violation du principe de substitution. Le programmeur décide de faire dériver l'autruche de l'oiseau pour obtenir les plumes et le bec, mais ne respecte pas la spécification de l'opération **`Voler()`** en réalisant un code qui ne fait pas voler l'autruche.

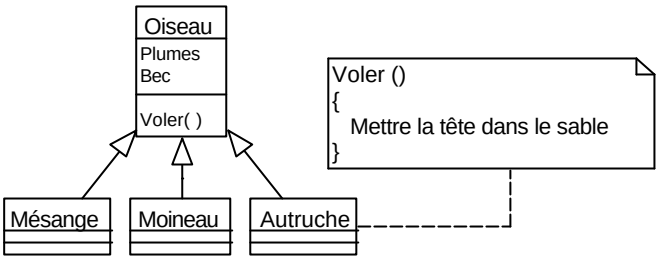


Figure 104 : Exemple de violation du principe de substitution.

La malversation précédente ne porte pas à conséquence tant que personne n'écrit de mécanisme général pour manipuler les oiseaux. En revanche, dès lors que des mécanismes généraux exploitent la spécification de la classe **`Oiseau`**, par exemple pour faire s'envoler tous les types d'oiseaux en cas de danger, l'incohérence introduite dans la classe **`Autruche`** se traduit au mieux par une autruche qui se fait manger et au pire par la situation la plus catastrophique qui puisse arriver en application de la loi de Murphy.

Le diagramme de collaboration suivant montre comment une petite autruche se fait manger par un gros chat !

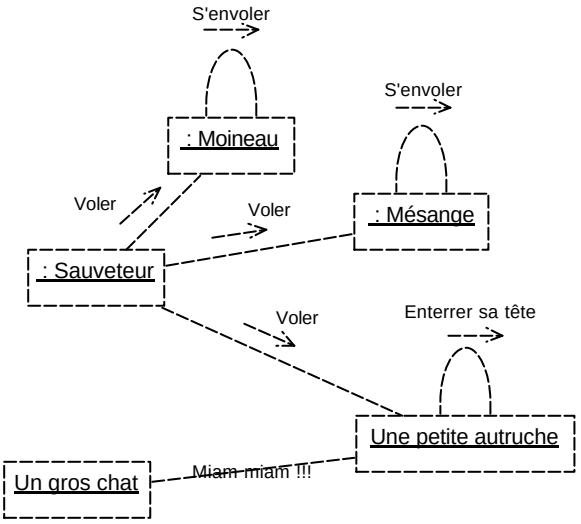


Figure 105 : Exemple de non respect du principe de substitution. La petite autruche ne se comporte pas comme un oiseau et se fait dévorer par le gros chat.

Déclenchement des opérations

Ce paragraphe a pour objectif de présenter les grandes lignes du mécanisme de liaison dynamique, indépendamment des particularités syntaxiques des langages de programmation. Les différentes étapes de l'exemple montrent les formes de déclenchement d'une opération en fonction de la classe de l'objet qui reçoit le message.

Le diagramme de classes suivant décrit la hiérarchie des classes qui sert de support à la discussion, et montre la définition et les réalisations de l'opération **`Z`** qui sera déclenchée. Les classes abstraites **`I`** et **`J`** ne sont pas instanciables.

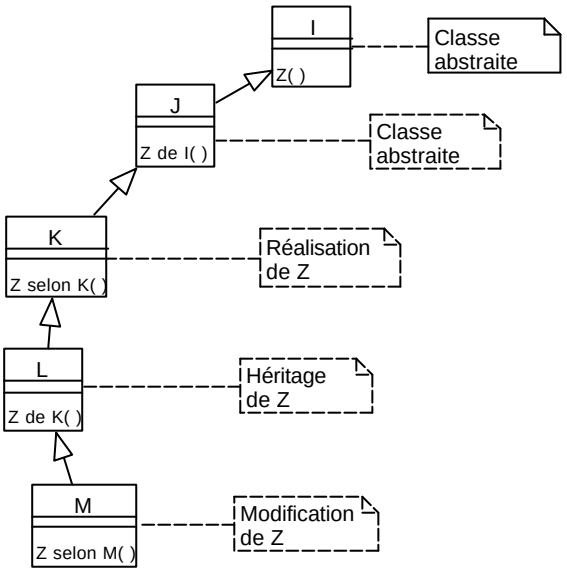


Figure 106 : Exemple de hiérarchie de classes. L'opération ***`Z`*** *est définie dans la classe de base* ***`I`****, puis réalisée dans la classe* ***`K`*** *et modifiée dans la classe* ***`M`****.*

Les objets de la classe **`Client`** possèdent des liens polymorphes vers des objets des classes **`K`**, **`L`** ou **`M`**.

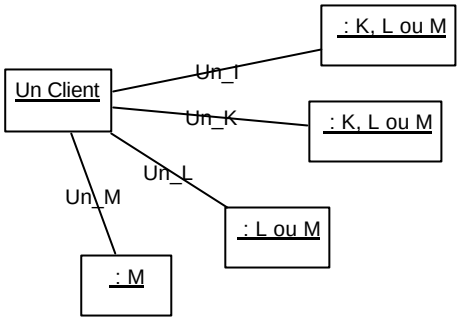


Figure 107 : Chaque objet de la classe ***Client*** *peut communiquer avec quatre objets instances des classes concrètes* ***K****,* ***L*** *ou* ***M****.*

Le lien **Un_I** est polymorphe et peut désigner un objet instance d'une classe concrète dérivée de **I** par exemple la sous-classe **K**. Le client peut ainsi manipuler un objet instance d'une sous-classe, au travers de la spécification d'une super-classe. Toutes les caractéristiques de la super-classe s'appliquent aux objets instances des sous-classes, en particulier le message **Z** peut être envoyé à l'objet **Un_K** le long du lien polymorphe **Un_I**.

Le code correspondant à l'opération **Z** est recherché à l'exécution en descendant l'arbre d'héritage jusqu'à la classe précise de l'objet. La classe **K** réalise l'opération **Z**, le message est compris et l'opération réalisée dans la classe **K** est exécutée.

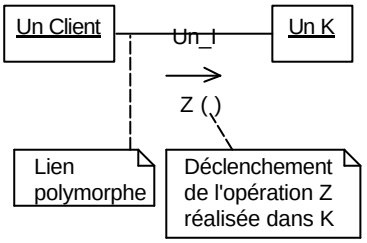


Figure 108 : Le lien ***Un_I*** *désigne un objet de la classe* ***K****. Le message est compris, car la classe* ***K*** *réalise l'opération* ***Z****.*

Le cas de figure suivant est proche du cas précédent, si ce n'est que la recherche s'effectue un cran plus bas dans la hiérarchie. La classe **L** n'ayant pas modifié la réalisation de l'opération **Z** héritée de la classe **K**, le même comportement que précédemment est déclenché.

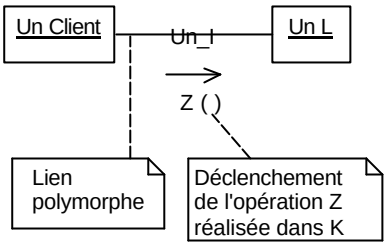


Figure 109 : La réalisation de l'opération ***Z*** *présente dans la classe* ***L*** *est héritée de la classe* ***K****. A l'exécution, le comportement réalisé dans la classe* ***K*** *sera déclenché.*

Toute classe intermédiaire dans l'arborescence des classes peut fournir une spécification pour manipuler les objets instances de ses sous-classes. A ce titre, le lien **Un_L** peut être exploité de manière polymorphe pour manipuler des objets instances de la classe **M**. Comme **M** modifie l'opération **Z**, l'expression **Un_L.Z()** déclenche le code réalisé dans la classe **M**.

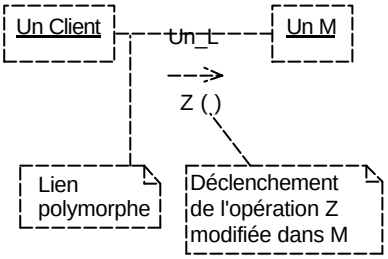


Figure 110 : Utilisation de la spécification d'une classe de niveau intermédiaire.

La liaison dynamique n'est nécessaire que pour rechercher automatiquement le code modifié dans les sous-classes et appelé depuis la spécification d'une super-classe. Lorsque la liaison dynamique n'est pas requise, les passages de messages peuvent être avantageusement remplacés par de simples appels de procédures résolus statiquement. Les langages comme C++ ou Ada 95 autorisent le programmeur à choisir le type de liaison au coup par coup, opération par opération, selon les besoins. En Eiffel, le programmeur n'a pas à se soucier du choix de la liaison car le compilateur est suffisamment intelligent pour reconnaître les liaisons qui seront toujours statiques.

Dans l'exemple suivant, la liaison peut être statique d'une part parce que le lien **Un_K** désigne un objet de classe **K** et d'autre part parce que l'opération **Z** est réalisée dans cette classe.

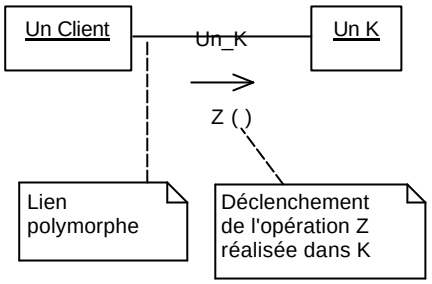


Figure 111 : Situation résoluble statiquement.

Le code qui exploite le polymorphisme se caractérise par la quasi-absence d'instructions à branchements multiples (**`case`**, **`switch`**). Ceci est dû au fait que les branchements sont réalisés de manière implicite – en fonction de la classe de l'objet destinataire du message – lors de la recherche de l'opération à exécuter en réponse à la réception d'un message.

Influence du typage

Le polymorphisme existe dans les environnements typés comme dans les environnements non typés, mais seuls les environnements typés garantissent une exécution sans surprise des programmes – à condition toutefois de respecter le principe de substitution. La notion de typage permet au compilateur de vérifier statiquement que les messages envoyés à un objet seront compris lors de l'exécution. En Ada 95, en Eiffel, en C++ ou en Java, les messages sont toujours compris par le destinataire du fait du typage fort. En l'absence de typage fort, n'importe quel message peut être envoyé à n'importe quel objet. En Smalltalk, un message peut ne pas être compris et l'expéditeur doit donc se préparer à cette éventualité. Ceci n'enlève rien à la vertu de la liaison dynamique, toujours utile pour trouver le code le plus adapté en fonction de la classe. Mais à la différence des langages typés, la recherche peut être infructueuse.

Il importe de ne pas confondre le polymorphisme avec la surcharge des opérations, parfois appelée polymorphisme ad hoc. Certains langages autorisent l'emploi d'un même nom pour désigner une famille d'opérations, avec un profil de paramètres différent pour chaque opération. La surcharge est un moyen élégant d'offrir la notion de signature variable, sans tomber dans le travers des opérations à nombre de paramètres variable comme en C. La surcharge est toujours résolue statiquement par les compilateurs, elle n'a rien à voir avec la liaison dynamique.

Les heureux parents reconnaîtront le comportement de leur progéniture dans la classe suivant. L'opération **`Manger()`** est surchargée : elle existe sous trois formes différentes, identifiables statiquement par le compilateur en fonction de la signature.

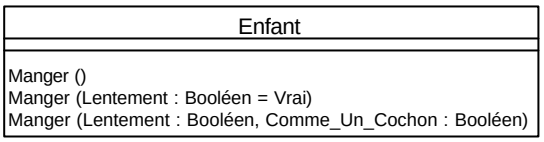


Figure 112 : Exemple d'une opération surchargée. La classe ***`Enfant`*** *propose trois opérations* ***`Manger()`*** *différentes, identifiables statiquement par leur signature.*

3

La notation UML

La notation UML est une fusion des notations de Booch, OMT, OOSE et d'autres notations. UML est conçue pour être lisible sur des supports très variés comme les tableaux blancs, le papier, les nappes de restaurants, les écrans d'ordinateurs, les impressions en noir et blanc, etc. Les concepteurs de la notation ont recherché avant tout la simplicité ; UML est intuitive, homogène et cohérente. Les symboles embrouillés, redondants ou superflus ont été éliminés en faveur d'un meilleur rendu visuel.

UML se concentre sur la description des artefacts du développement de logiciel, plutôt que sur la formalisation du processus de développement lui-même : elle peut ainsi être utilisée pour décrire les éléments logiciels, obtenus par l'application de différents processus de développement. UML n'est pas une notation fermée : elle est générique, extensible et configurable par l'utilisateur. UML ne recherche pas la spécification à outrance : il n'y a pas une représentation graphique pour tous les concepts imaginables ; en cas de besoins particuliers, des précisions peuvent être apportées au moyen de mécanismes d'extension et de commentaires textuels. Une grande liberté est donnée aux outils pour le filtrage et la visualisation d'information. L'usage de couleurs, de dessins et d'attributs graphiques particuliers est laissé à la discrétion de l'utilisateur.

Ce chapitre propose un survol de la sémantique des éléments de modélisation d'UML, décrits de manière précise dans le document UML 1.0, Semantics[9]. Cette présentation a pour objectif d'introduire les principaux concepts de modélisation et de montrer leur articulation au sein de la notation UML. Les éléments de visualisation et les éléments de modélisation sont présentés conjointement, en se

[9] Rational Software Corporation 1997, *UML Semantics V 1.0*.

servant de la notation comme d'un support pour faciliter la présentation de la sémantique.

UML définit neuf sortes de diagrammes pour représenter les différents points de vue de modélisation. L'ordre de présentation de ces différents diagrammes ne reflète pas un ordre de mise en œuvre dans un projet réel, mais simplement une démarche pédagogique qui essaie de minimiser les prérequis et les références croisées.

Les diagrammes d'UML

Un diagramme donne à l'utilisateur un moyen de visualiser et de manipuler des éléments de modélisation. Les différents types de diagrammes d'UML sont présentés dans l'extrait du métamodèle suivant.

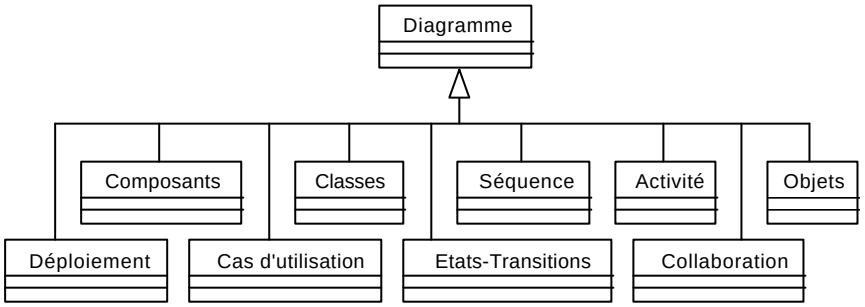


Figure 113 : Différents types de diagrammes définis par UML.

Un diagramme contient des attributs de placement et de rendu visuel qui ne dépendent que du point de vue. La plupart des diagrammes se présentent sous la forme de graphes, composés de sommets et d'arcs. Les diagrammes contiennent des éléments de visualisation qui représentent des éléments de modélisation éventuellement issus de paquetages distincts, même en l'absence de relations de visibilité entre ces paquetages[10].

Les diagrammes peuvent montrer tout ou partie des caractéristiques des éléments de modélisation, selon le niveau de détail utile dans le contexte d'un diagramme donné. Les diagrammes peuvent également rassembler des informations liées entre elles, pour montrer par exemple les caractéristiques héritées par une classe.

Voici, par ordre alphabétique, la liste des différents diagrammes :

[10] Car l'utilisateur peut tout voir !

- *les diagrammes d'activités* qui représentent le comportement d'une opération en termes d'actions,
- *les diagrammes de cas d'utilisation* qui représentent les fonctions du système du point de vue de l'utilisateur.
- *les diagrammes de classes* qui représentent la structure statique en termes de classes et de relations,
- *les diagrammes de collaboration* qui sont une représentation spatiale des objets, des liens et des interactions,
- *les diagrammes de composants* qui représentent les composants physiques d'une application,
- *les diagrammes de déploiement* qui représentent le déploiement des composants sur les dispositifs matériels,
- *les diagrammes d'états-transitions* qui représentent le comportement d'une classe en terme d'états,
- *les diagrammes d'objets* qui représentent les objets et leurs relations et correspondent à des diagrammes de collaboration simplifiés, sans représentation des envois de message,
- *les diagrammes de séquence* qui sont une représentation temporelle des objets et de leurs interactions.

Les diagrammes de collaboration et les diagrammes de séquence sont tous deux appelés diagrammes d'interaction. Les diagrammes d'états-transitions sont également appelés *Statecharts*[11], nom donné par leur auteur, David Harel.

Concepts de base

Il est pratique de représenter la sémantique des éléments de modélisation d'UML selon le formalisme d'UML.

Ce type de représentation récursive pose cependant le problème *de l'œuf et de la poule*, principalement lors de l'apprentissage. C'est pourquoi il est conseillé, en première lecture, de considérer les diagrammes comme des exemples de la notation, plutôt que de chercher à approfondir leur compréhension

Quoi qu'il en soit, les diagrammes montrent des vues simplifiées du métamodèle afin de rendre le texte accessible au plus grand nombre de lecteurs.

---

[11] Harel, D. 1987. *Statecharts : A Visual Formalism for Complex Systems*. Science of Computer Programming vol. 8.

Les éléments communs

Les éléments sont les briques de base d'UML. Les éléments comprennent les éléments de modélisation et les éléments de visualisation. Les éléments de modélisation représentent les abstractions du système en cours de modélisation. Les éléments de visualisation procurent des projections textuelles ou graphiques qui permettent la manipulation des éléments de modélisation.

Les éléments sont regroupés en paquetages qui contiennent ou référencent les éléments de modélisation. Un modèle est une abstraction d'un système, représenté par une arborescence de paquetages.

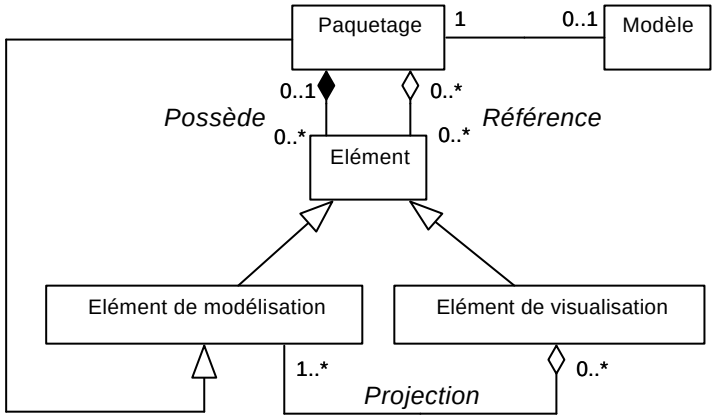


Figure 114 : Extrait du métamodèle. Représentation des deux grandes familles d'éléments qui forment le contenu des modèles.

Les mécanismes communs

UML définit un petit nombre de mécanismes communs qui assurent l'intégrité conceptuelle de la notation. Ces mécanismes communs comprennent les stéréotypes, les étiquettes, les notes, les contraintes, la relation de dépendance et les dichotomies **`(type, instance)`** et **`(type, classe)`**. Chaque élément de modélisation possède une spécification qui contient la description unique et détaillée de toutes les caractéristiques de l'élément.

Les stéréotypes, les étiquettes et les contraintes permettent l'extension d'UML. Les stéréotypes spécialisent les classes du métamodèle, les étiquettes étendent les attributs des classes du métamodèle et les contraintes étendent la sémantique du métamodèle.

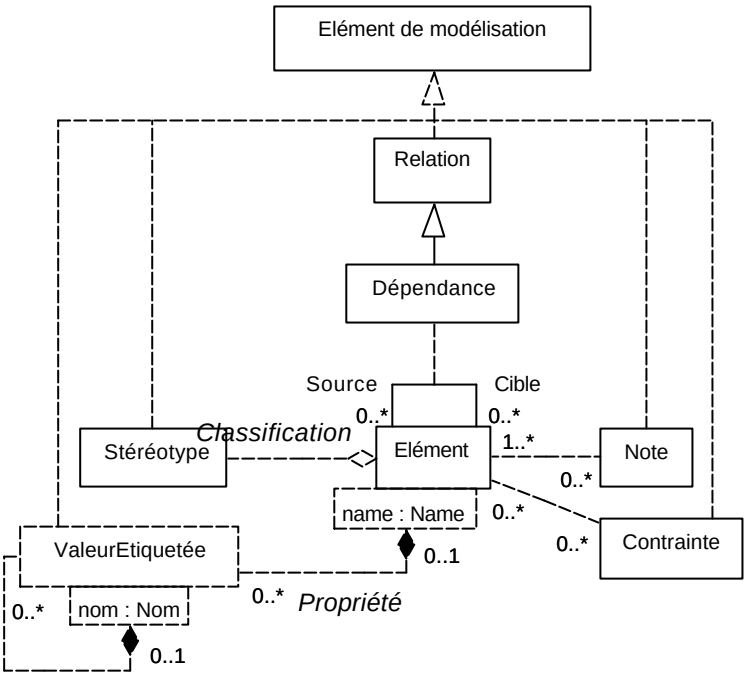


Figure 115 : Extrait du métamodèle. Représentation des mécanismes communs.

Les stéréotypes

Les stéréotypes font partie des mécanismes d'extensibilité, prévus dans UML. Chaque élément de modélisation d'UML a au plus un stéréotype, lorsque la sémantique de base de l'élément est insuffisante. Un stéréotype permet la métaclassification d'un élément d'UML. Il permet aux utilisateurs (méthodologistes, constructeurs d'outils, analystes et concepteurs) d'ajouter de nouvelles classes d'éléments de modélisation, en plus du noyau prédéfini par UML. Les stéréotypes facilitent également l'unification de concepts proches, comme les sous-systèmes ou les catégories, qui sont exprimés au moyen de stéréotypes de paquetage.

Les stéréotypes permettent l'extension contrôlée des classes du métamodèle, par les utilisateurs d'UML. Un élément spécialisé par un stéréotype **S** est sémantiquement équivalent à une nouvelle classe du métamodèle, nommée elle aussi **S**. A l'extrême, toute la notation aurait pu être construite à partir des deux classes **`Truc`** et **`Stéréotype`**, les autres concepts en dérivant par stéréotypage de la classe **`Truc`**.

Les concepteurs d'UML ont recherché l'équilibre entre les classes incluses d'origine, et les extensions apportées par les stéréotypes. Ainsi, seuls les concepts fondamentaux ont été exprimés sous la forme de classes distinctes. Les

autres concepts, dérivables de ces concepts de base, ont été traités comme des stéréotypes.

Les étiquettes

Une étiquette est une paire **(nom, valeur)** qui décrit une propriété d'un élément de modélisation. Les propriétés permettent l'extension des attributs des éléments du métamodèle. Une étiquette modifie la sémantique de l'élément qu'elle qualifie.

Les notes

Une note est un commentaire attaché à un ou plusieurs éléments de modélisation. Par défaut, les notes ne véhiculent pas de contenu sémantique. Toutefois, une note peut être transformée en contrainte au moyen d'un stéréotype, et dans ce cas, elle modifie la sémantique des éléments de modélisation auxquels elle est attachée.

Les contraintes

Une contrainte est une relation sémantique quelconque entre éléments de modélisation. UML ne spécifie pas de syntaxe particulière pour les contraintes, qui peuvent ainsi être exprimées en langage naturel, en pseudo-code, par des expressions de navigation ou par des expressions mathématiques.

La relation de dépendance

La relation de dépendance définit une relation d'utilisation unidirectionnelle entre deux éléments de modélisation, appelés respectivement source et cible de la relation. Les notes et les contraintes peuvent également être source d'une relation de dépendance.

Dichotomies (type, instance) et (type, classe)

De nombreux éléments de modélisation présentent une dichotomie **(type, instance)**, dans laquelle le type dénote l'essence de l'élément, et l'instance avec ses valeurs correspond à une manifestation de ce type.

De même, la dichotomie **(type, classe)** correspond à la séparation entre la spécification d'un élément qui est énoncé par le type et la réalisation de cette spécification qui est fournie par la classe.

Les types primitifs

Les types primitifs regroupent toutes les abstractions situées en marge d'UML. Ces types élémentaires ne sont pas des éléments de modélisation et ne possèdent donc ni stéréotypes, ni étiquettes, ni contraintes.

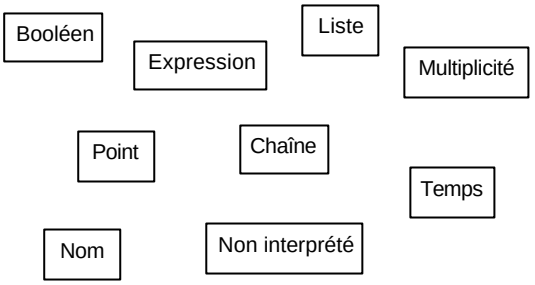


Figure 116 : Extrait du métamodèle. Représentation des types primitifs d'UML.

Les types suivants forment les types primitifs d'UML :

- *Booléen.* Un booléen est un type énuméré qui comprend les deux valeurs **`Vrai`** et **`Faux`**.
- *Expression.* Une expression est une chaîne de caractères dont la syntaxe est hors de la portée d'UML.
- *Liste.* Une liste est un conteneur dont les parties sont ordonnées et peuvent être indexées.
- *Multiplicité.* Une multiplicité est un ensemble non vide d'entiers positifs, étendu par un lexème particulier, le caractère **`*`**, qui signifie que la multiplicité est illimitée. La syntaxe de la multiplicité est donnée par la règle suivante :

  ```
  multiplicité ::= [intervalle | nombre]
                       {',' multiplicité}
  intervalle ::= nombre " .. " nombre
  nombre ::= nombre_positif | nom | '*'
  ```

- *Nom.* Un nom est une chaîne de caractères qui permet de désigner un élément. Des noms composés peuvent être formés à partir de noms simples, selon la règle suivante :

  ```
  nom_composé ::= nom_simple { '.' nom_composé }
  ```

  Le nom d'un élément peut être qualifié par le nom du paquetage qui le contient ou qui le référence, selon la syntaxe suivante :

  ```
  nom_qualifié ::= qualificateur "::" nom_simple
  qualificateur ::= nom_de_paquetage
            { "::" qualificateur }
  ```

- *Point.* Un point est un tuple **`(x, y, z)`** qui décrit une position dans l'espace. Par défaut, la composante **`z`** prend la valeur **`0`**.

- *Chaîne*. Une chaîne est une suite de caractères, désignée par un nom.
- *Temps.* Un temps est une chaîne qui représente un temps absolu ou relatif et dont la syntaxe est hors de la portée d'UML.
- *Non-interprété.* Un non-interprété est un truc[12] dont la signification dépend du domaine.

Les paquetages

Les paquetages offrent un mécanisme général pour la partition des modèles et le regroupement des éléments de modélisation. Chaque paquetage est représenté graphiquement par un dossier.

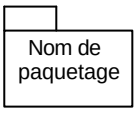


Figure 117 : Représentation des paquetages.

Les paquetages divisent et organisent les modèles de la même manière que les répertoires organisent les systèmes de fichiers. Chaque paquetage correspond à un sous-ensemble du modèle et contient, selon le modèle, des classes, des objets, des relations, des composants ou des nœuds, ainsi que les diagrammes associés.

La décomposition en paquetages n'est pas l'amorce d'une décomposition fonctionnelle ; chaque paquetage est un regroupement d'éléments selon un critère purement logique. La forme générale du système (l'architecture du système) est exprimée par la hiérarchie de paquetages et par le réseau de relations de dépendance entre paquetages.

Les stéréotypes `<<Catégorie>>` et `<<Sous-système>>` permettent de distinguer, si besoin est, les paquetages de la vue logique et les paquetages de la vue de réalisation (ces vues sont détaillées dans le quatrième chapitre).

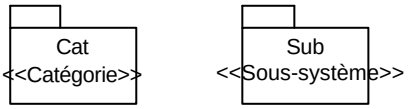


Figure 118 : Les paquetages peuvent être stéréotypés, pour distinguer par exemple les catégories de la vue logique et les sous-systèmes de la vue de réalisation.

[12] UML parle de *blob*.

Un paquetage définit un espace de nom, de sorte que deux éléments différents, contenus par deux paquetages différents, peuvent porter le même nom.

Un paquetage peut contenir d'autres paquetages, sans limite du niveau d'emboîtement. Un niveau donné peut contenir un mélange de paquetages et d'autres éléments de modélisation, de la même manière qu'un répertoire peut contenir des répertoires et des fichiers.

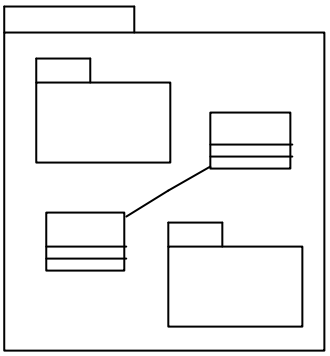


Figure 119 : Exemple de représentation mixte. Un paquetage contient des éléments de modélisation, accompagnés éventuellement d'autres paquetages.

Chaque élément appartient à un paquetage. Le paquetage de plus haut niveau est le paquetage racine de l'ensemble d'un modèle. Une classe contenue par un paquetage peut également apparaître dans un autre paquetage sous la forme d'un élément importé, au travers d'une relation de dépendance entre paquetages.

Les imports entre paquetages se représentent dans les diagrammes de classes, les diagrammes de cas d'utilisation et les diagrammes de composants, au moyen d'une relation de dépendance stéréotypée, orientée du paquetage client vers le paquetage fournisseur.

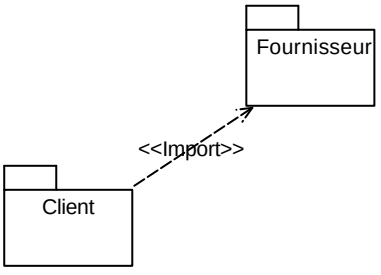


Figure 120 : Représentation des imports entre deux paquetages au moyen d'une relation de dépendance stéréotypée.

Une relation de dépendance entre deux paquetages signifie qu'au moins une classe du paquetage client utilise les services offerts par au moins une classe du paquetage fournisseur. Toutes les classes contenues par un paquetage ne sont pas nécessairement visibles de l'extérieur du paquetage.

L'opérateur **`::`** permet de désigner une classe définie dans un contexte différent du contexte courant. L'expression **`Zoo::Kangourou`** désigne la classe **`Kangourou`** définie dans le paquetage **`Zoo`**.

Un paquetage est un regroupement d'éléments de modélisation, mais aussi une encapsulation de ces éléments. A l'image des classes, les paquetages possèdent une interface et une réalisation. Chaque élément contenu par un paquetage possède un paramètre qui signale si l'élément est visible ou non à l'extérieur du paquetage. Les valeurs prises par le paramètre sont : **`public`** ou **`implementation`** (privé).

Dans le cas des classes, seules celles indiquées comme publiques apparaissent dans l'interface du paquetage qui les contient ; elles peuvent alors être utilisées par les classes membres des paquetages clients. Les classes de réalisation ne sont utilisables qu'au sein du paquetage qui les contient.

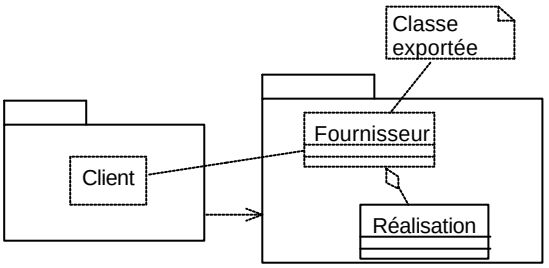


Figure 121 : Seules les classes exportées sont visibles à l'extérieur des paquetages.

Les relations de dépendance entre paquetages entraînent des relations d'obsolescence entre les éléments de modélisation contenus dans ces paquetages.

Pour des raisons de compilation lors de la réalisation, il est fortement conseillé que les dépendances entre paquetages ne forment que des graphes acycliques. Il faut donc éviter la situation suivante :

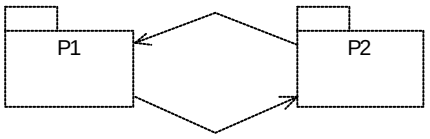


Figure 122 : Exemple de dépendance circulaire entre paquetages, à éviter.

De même, il convient d'éviter les dépendances circulaires transitives, illustrées ci-dessous :

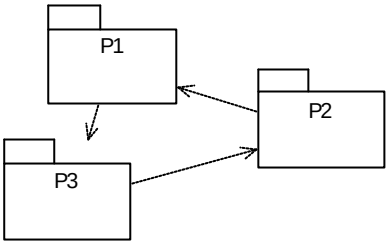


Figure 123 : Exemple de dépendance circulaire transitive, à éviter.

En règle générale, les dépendances circulaires peuvent être réduites en éclatant un des paquetages incriminés en deux paquetages plus petits, ou en introduisant un troisième paquetage intermédiaire.

Certains paquetages sont utilisés par tous les autres paquetages. Ces paquetages regroupent par exemple des classes de base comme des ensembles, des listes, des files, ou encore des classes de traitement des erreurs. Ces paquetages possèdent une propriété qui les désigne comme paquetages globaux. Il n'est pas nécessaire de montrer les relations de dépendances entre ces paquetages et leurs utilisateurs ce qui limite la charge graphique des diagrammes.

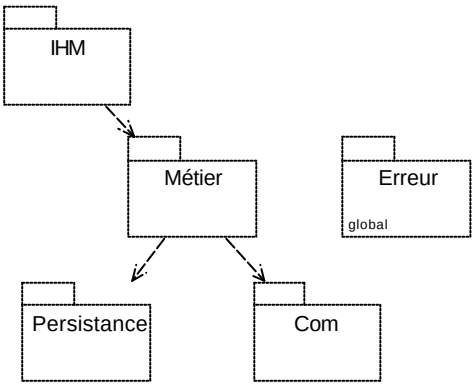


Figure 124 : Afin de réduire la charge graphique, les dépendances envers les paquetages visibles globalement ne sont pas portées dans les diagrammes.

Les diagrammes de classes

Les diagrammes de classes expriment de manière générale la structure statique d'un système, en termes de classes et de relations entre ces classes. De même qu'une classe décrit un ensemble d'objets, une association décrit un ensemble de liens ; les objets sont instances des classes et les liens sont instances des relations. Un diagramme de classes n'exprime rien de particulier sur les liens d'un objet donné, mais décrit de manière abstraite les liens potentiels d'un objet vers d'autres objets.

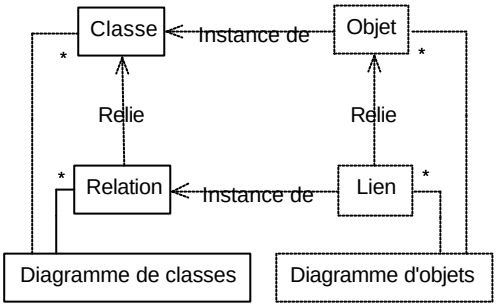


Figure 125 : Extrait du métamodèle. Les diagrammes de classes représentent les classes et les relations.

Les classes

Les classes sont représentées par des rectangles compartimentés. Le premier compartiment contient le nom de la classe. Le nom de la classe doit permettre de comprendre ce que la classe est, et non ce qu'elle fait. Une classe n'est pas une fonction, une classe est une description abstraite – condensée – d'un ensemble d'objets du domaine de l'application. Les deux autres compartiments contiennent respectivement les attributs et les opérations de la classe.

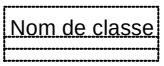


Figure 126 : Représentation graphique des classes.

Une classe contient toujours au moins son nom. Lorsque le nom n'a pas encore été trouvé, ou n'est pas encore arrêté, il est conseillé de faire figurer un nom générique facile à identifier, comme **`A_Définir`**. Les compartiments d'une classe peuvent être supprimés lorsque leur contenu n'est pas pertinent dans le contexte d'un diagramme. La suppression des compartiments reste purement visuelle, elle ne signifie pas qu'il n'y a pas d'attribut ou d'opération.

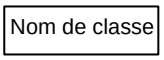


Figure 127 : Représentation graphique simplifiée par suppression des compartiments.

Le rectangle qui symbolise la classe peut également contenir un stéréotype et des propriétés. UML définit les stéréotypes de classe suivants :

- **`<<signal>>`**, une occurrence remarquable qui déclenche une transaction dans un automate,
- **`<<interface>>`**, une description des opérations visibles,
- **`<<métaclasse>>`**, la classe d'une classe, comme en Smalltalk,
- **`<<utilitaire>>`**, une classe réduite au concept de module et qui ne peut être instanciée.

Les propriétés désignent toutes les valeurs attachées à un élément de modélisation, comme les attributs, les associations et les étiquettes. Une étiquette est une paire **`(attribut, valeur)`** définie par l'utilisateur ; elle permet de préciser par exemple des informations de génération de code, d'identification ou de références croisées.

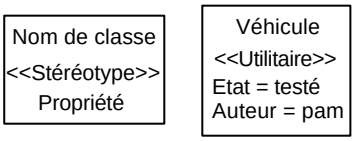


Figure 128 : Les compartiments de classes peuvent contenir un stéréotype et des propriétés, définis librement par l'utilisateur.

Les attributs et les opérations

Les attributs et les opérations peuvent être montrés de manière exhaustive ou non dans les compartiments des classes. Par convention, le premier compartiment contient les attributs et le deuxième compartiment contient les opérations.

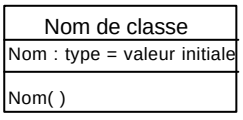


Figure 129 : Visualisation des attributs et des opérations dans les compartiments de la classe.

La syntaxe retenue pour la description des attributs est la suivante :

```
Nom_Attribut : Type_Attribut = Valeur_Initiale
```

Cette description peut être complétée progressivement, lors de la transition de l'analyse vers la conception.

Il arrive que des propriétés redondantes soit spécifiées lors de l'analyse des besoins. Les attributs dérivés offrent une solution pour allouer des propriétés à des classes, tout en indiquant clairement que ces propriétés sont dérivées d'autres propriétés déjà allouées. Dans l'exemple suivant, la classe **`Rectangle`** possède un attribut **`Longueur`**, un attribut **`Largeur`**, ainsi qu'un attribut dérivé **`Surface`**, qui peut être construit à partir des deux autres attributs.

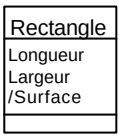


Figure 130 : Exemple d'attribut dérivé. La surface d'un rectangle peut être déterminée à partir de la longueur et de la largeur.

Par la suite, en conception, l'attribut dérivé **`/Surface`** sera transformé en une opération **`Surface()`** qui encapsulera le calcul de surface. Cette transformation peut toutefois être effectuée sans attendre, dès lors que la nature dérivée de la propriété surface a été détectée.

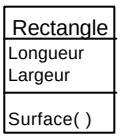


Figure 131 : Transformation de l'attribut dérivé en une opération de calcul de la surface.

La syntaxe retenue pour la description des opérations est la suivante :

```
Nom_Opération
(Nom_Argument : Type_Argument = Valeur_Par_Défaut,...)
   : Type_Retourné
```

Toutefois, étant donné la longueur de la spécification, les arguments des opérations peuvent être supprimés dans les graphiques.

Visibilité des attributs et des opérations

UML définit trois niveaux de visibilité pour les attributs et les opérations :

- *public* qui rend l'élément visible à tous les clients de la classe,
- *protégé* qui rend l'élément visible aux sous-classes de la classe,

- *privé* qui rend l'élément visible à la classe seule.

L'information de visibilité ne figure pas toujours de manière explicite dans les diagrammes de classes, ce qui ne veut pas dire que la visibilité n'est pas définie dans le modèle. Par défaut, le niveau de visibilité est symbolisé par les caractères **+**, **#** et **-**, qui correspondent respectivement aux niveaux **`public`**, **`protégé`** et **`privé`**.

Certains attributs et certaines opérations peuvent être visibles globalement, dans toute la portée lexicale de la classe. Ces éléments, également appelés variables et opérations de classe, sont représentés comme un objet par un nom souligné. La notation se justifie par le fait qu'une variable de classe apparaît comme un objet partagé par les instances d'une classe. Par extension, les opérations de classe sont aussi soulignées.

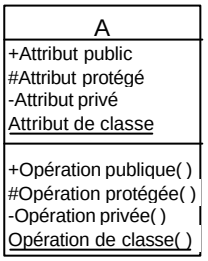


Figure 132 : Représentation des différents niveaux de visibilité des attributs et des opérations.

Les interfaces

Une interface utilise un type pour décrire le comportement visible d'une classe, d'un composant (décrits plus loin dans l'ouvrage) ou d'un paquetage. Une interface est un stéréotype d'un type. UML représente les interfaces au moyen de petits cercles reliés par un trait à l'élément qui fournit les services décrits par l'interface.

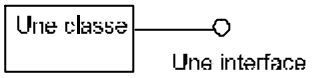


Figure 133 : Représentation d'une interface au moyen d'un petit cercle relié à la classe qui fournit effectivement les services.

Les interfaces peuvent également se représenter au moyen de classes stéréotypées.

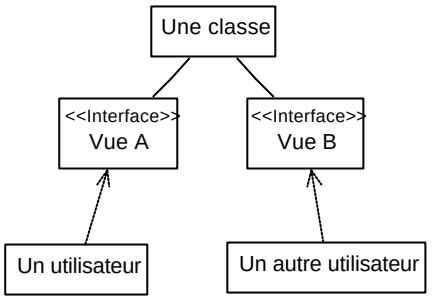


Figure 134 : Exemple de représentation des interfaces au moyen de classes stéréotypées.

Une interface fournit une vue totale ou partielle d'un ensemble de services offerts par un ou plusieurs éléments. Les dépendants d'une interface utilisent tout ou partie des services décrits dans l'interface.

Les classes paramétrables

Les classes paramétrables sont des modèles de classes. Elles correspondent aux classes génériques d'Eiffel et aux *templates* de C++. Une classe paramétrable ne peut être utilisée telle quelle. Il convient d'abord de l'instancier, afin d'obtenir une classe réelle qui pourra à son tour être instanciée afin de donner des objets. Lors de l'instanciation, les paramètres effectifs personnalisent la classe réelle obtenue à partir de la classe paramétrable. Les classes paramétrables permettent de construire des collections universelles, typées par les paramètres effectifs.

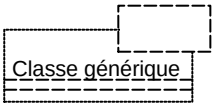


Figure 135 : Représentation graphique des classes paramétrables.

Ce type de classes n'apparaît généralement pas en analyse, sauf dans le cas particulier de la modélisation d'un environnement de développement. Les classes paramétrables sont surtout utilisées en conception détaillée, pour incorporer par exemple des composants réutilisables. La figure suivante représente l'instanciation d'une table générique, pour réaliser un annuaire de personnes. Le paramètre générique formel apparaît dans le rectangle pointillé de la classe paramétrable alors que le paramètre générique effectif est accolé au nom de la classe obtenue par instanciation. L'instanciation est matérialisée par une flèche pointillée, orientée de l'instance vers la classe paramétrable.

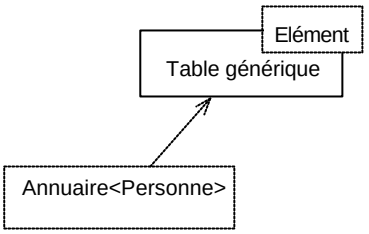


Figure 136 : Exemple de représentation de l'instanciation d'une classe paramétrable.

Les classes utilitaires

Il est parfois utile de regrouper des éléments (les fonctions d'une bibliothèque mathématique par exemple) au sein d'un module, sans pour autant vouloir construire une classe complète. La classe utilitaire permet de représenter de tels modules et de les manipuler graphiquement comme les classes conventionnelles.

Les classes utilitaires ne peuvent pas être instanciées car elles ne sont pas des types de données. Il ne faut pas pour autant les confondre avec les classes abstraites qui ne peuvent pas être instanciées car elles sont des spécifications pures (voir le paragraphe qui traite de la généralisation). En C++, une classe utilitaire correspond à une classe qui ne contient que des membres statiques (fonctions et données).

Le stéréotype **`<<Utilitaire>>`** spécialise les classes en classes utilitaires. UML propose également une représentation graphique alternative, avec une bordure grisée.

Figure 137 : Exemple de représentation graphique d'une classe utilitaire.

Les associations

Les associations représentent des relations structurelles entre classes d'objets. Une association symbolise une information dont la durée de vie n'est pas négligeable par rapport à la dynamique générale des objets instances des classes associées. La plupart des associations sont binaires, c'est-à-dire qu'elles connectent deux classes. Les associations se représentent en traçant une ligne entre les classes associées.

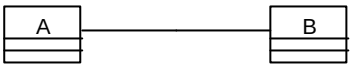


Figure 138 : Les associations se représentent en traçant une ligne entre les classes associées.

Les associations peuvent être représentées par des traits rectilignes ou obliques, selon les préférences de l'utilisateur. L'usage recommande de se tenir le plus possible à un seul style de représentation des traits afin de simplifier la lecture des diagrammes au sein d'un projet.

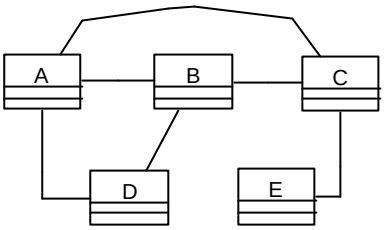


Figure 139 : Les associations peuvent être représentées par des traits rectilignes ou obliques selon les préférences de l'utilisateur.

Arité des associations

La plupart des associations sont dites binaires car elles relient deux classes. Des arités supérieures peuvent cependant exister et se représentent alors au moyen d'un losange sur lequel arrivent les différentes composantes de l'association.

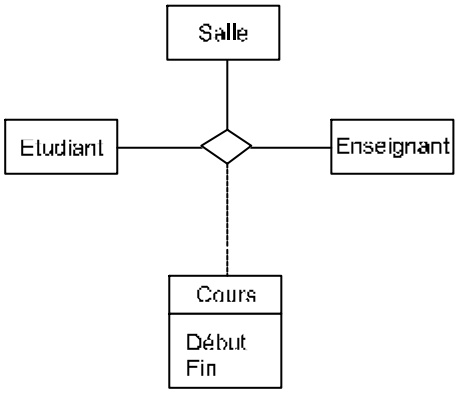


Figure 140 : Exemple de relation ternaire qui matérialise un cours.

Les associations n-aires peuvent généralement se représenter en promouvant l'association au rang de classe et en ajoutant une contrainte qui exprime que les multiples branches de l'association s'instancient toutes simultanément, en un

même lien. Dans l'exemple suivant, la contrainte est exprimée au moyen d'un stéréotype qui précise que la classe **`Cours`** réalise une association ternaire.

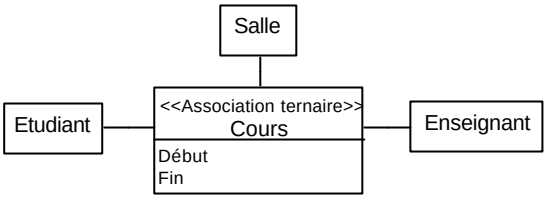


Figure 141 : Représentation d'une association ternaire au moyen d'une classe avec stéréotype.

Comme pour les associations binaires, les extrémités d'une association n-aire sont appelées rôles et peuvent porter un nom. La difficulté de trouver un nom différent à chaque rôle d'une association n-aire est souvent le signe d'une association d'arité inférieure.

Nommage des associations

Les associations peuvent être nommées ; le nom de l'association apparaît alors en italique, au milieu de la ligne qui symbolise l'association, plus précisément dessus, au-dessus ou au-dessous.

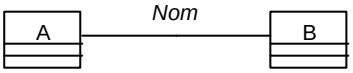


Figure 142 : Les associations peuvent être nommées afin de faciliter la compréhension des modèles.

Sans en faire une règle systématique, l'usage recommande de nommer les associations par une forme verbale, soit active comme *travaille pour*, soit passive comme *est employé par.*

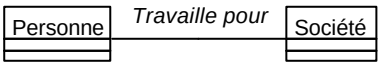


Figure 143 : Nommage d'une association par une forme verbale active.

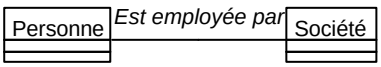


Figure 144 : Nommage d'une association par une forme verbale passive.

Dans les deux cas, le sens de lecture du nom peut être précisé au moyen d'un petit triangle dirigé vers la classe désignée par la forme verbale et placé à proximité du nom de l'association.

Dans un but de simplification, le petit triangle peut être remplacé par les signes **<** et **>** disponibles dans toutes les fontes.

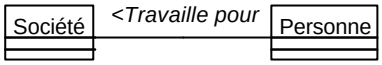


Figure 145 : Précision du sens d'application du nom d'une association.

Les associations entre classes expriment principalement la structure statique ; de ce fait, le nommage par des formes verbales qui évoquent plutôt le comportement se trouve un peu en porte-à-faux avec l'esprit général des diagrammes de classes. Le nommage des extrémités des relations permet de clarifier les diagrammes, comme le nommage des associations, mais d'une manière plus passive, plus en phase avec la tonalité statique des diagrammes de classes.

Nommage des rôles

L'extrémité d'une association est appelée rôle. Chaque association binaire possède deux rôles, un à chaque extrémité. Le rôle décrit comment une classe voit une autre classe au travers d'une association. Un rôle est nommé au moyen d'une forme nominale. Visuellement, le nom d'un rôle se distingue du nom d'une association, car il est placé près d'une extrémité de l'association être en italique.

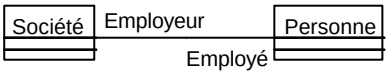


Figure 146 : Le nommage des rôles exprime comment une classe voit une autre classe au travers d'une association. Dans l'exemple ci-dessus, la personne voit la société comme son employeur, et la société voit la personne comme son employé.

Le nommage des associations et le nommage des rôles ne sont pas exclusifs l'un de l'autre ; toutefois, l'usage mélange rarement les deux constructions. Il est fréquent de commencer par décorer une association par un verbe, puis de se servir plus tard de ce verbe pour construire un substantif verbal qui nomme le rôle qui convient.

Lorsque deux classes sont reliées par une seule association, le nom des classes suffit souvent à caractériser le rôle ; le nommage des rôles prend tout son intérêt lorsque plusieurs associations relient deux classes. Dans ce cas, il n'y a aucune corrélation par défaut entre les objets qui participent aux deux relations. Chaque association exprime un concept distinct. Dans l'exemple suivant, il n'y a pas de relation entre les passagers et le pilote.

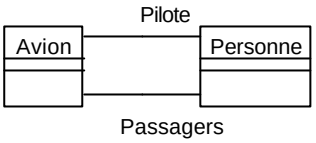


Figure 147 : Exemple d'associations multiples entre un avion et des personnes.

La présence d'un grand nombre d'associations entre deux classes peut être suspecte. Chaque association augmente le couplage entre les classes associées et un fort couplage peut être le signe d'une mauvaise décomposition. Il est également fréquent que les débutants visualisent plusieurs fois la même association en nommant chaque association avec le nom d'un des messages qui circulent entre les objets instances des classes associées.

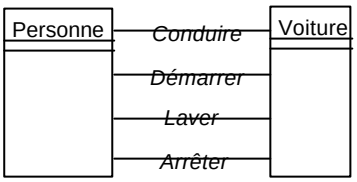


Figure 148 : Exemple de confusion entre association et messages. Une seule association est suffisante pour permettre à une personne de conduire, démarrer, laver et arrêter sa voiture.

Multiplicité des associations

Chaque rôle d'une association porte une indication de multiplicité qui montre combien d'objets de la classe considérée peuvent être liés à un objet de l'autre classe. La multiplicité est une information portée par le rôle, sous la forme d'une expression entière bornée.

1	Un et un seul
0..1	Zéro ou un
M..N	De M à N (entiers naturels)
*****	De zéro à plusieurs
0..*	De zéro à plusieurs
1..*	D'un à plusieurs

Figure 149 : Valeurs de multiplicité conventionnelles.

L'exemple suivant montre que plusieurs personnes travaillent pour une même société. La multiplicité de valeur **1** du côté de la classe **Société** n'est pas très réaliste car elle signifie que toutes les personnes ont un emploi. La multiplicité de valeur **0..*** du côté de la classe **Personne** signifie qu'une société emploie de zéro à plusieurs personnes.

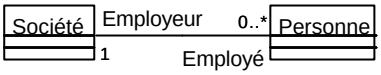


Figure 150 : Ce modèle supprime le chômage ! Chaque personne travaille pour une société, chaque société emploie de zéro à plusieurs personnes.

Une valeur de multiplicité supérieure à **1** implique une collection d'objets. Cette collection n'est pas bornée dans le cas d'une valeur *****, qui signifie que plusieurs objets participent à la relation, sans préjuger du nombre maximum d'objets. Le terme collection, plus général que liste ou ensemble, est employé afin d'éviter toute supposition sur la structure de données qui contient les objets.

Les valeurs de multiplicité expriment des contraintes liées au domaine de l'application, valables durant toute l'existence des objets. Les multiplicités ne doivent pas être considérées durant les régimes transitoires, comme lors de la création ou de la destruction des objets. Une multiplicité de valeur **1** indique qu'en régime permanent, un objet possède obligatoirement un lien vers un autre objet ; pour autant, il ne faut pas essayer d'en déduire le profil de paramètres des constructeurs et systématiquement prévoir un paramètre de la classe de l'objet lié. Les valeurs de multiplicité n'impliquent rien de particulier sur l'ordre de création des objets. L'exemple suivant montre deux classes connectées par une association de multiplicité **1** vers **1**, sans rien supposer sur le profil de paramètres des constructeurs d'objets. Dans l'exemple suivant, il n'est pas nécessaire de disposer d'une voiture pour construire un moteur et inversement.

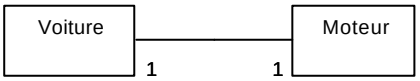


Figure 151 : La multiplicité d'une association est le reflet d'une contrainte du domaine.

La détermination de valeurs de multiplicité optimales est très importante pour trouver le bon équilibre, d'une part, entre souplesse et possibilité d'extension, et d'autre part, entre complexité et efficacité. En analyse, seule la valeur de la multiplicité est réellement importante. En revanche, en conception, il faut choisir des structures de données (pile, file, ensemble...) pour réaliser les collections qui correspondent aux multiplicités de type **1..*** ou **0..***. La surestimation des valeurs de multiplicité induit un surcoût en taille de stockage et en vitesse de recherche. De même, une multiplicité qui commence à zéro implique que les

diverses opérations doivent contenir du code pour tester la présence ou l'absence de liens dans les objets.

Les valeurs de multiplicité sont souvent employés pour décrire manière générique les associations. Les formes les plus courantes sont les associations **1** vers **1**, **1** vers **N** et **N** vers **N** représentées dans la figure suivante.

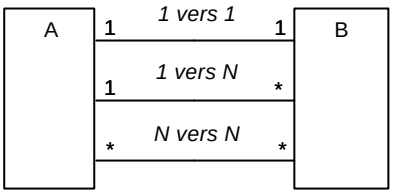


Figure 152 : Description générique des associations selon les valeurs de multiplicité.

Placement des attributs selon les valeurs de multiplicité

La réification des associations prend tout son intérêt pour les associations **N** vers **N**. Pour les associations **1** vers **1**, les attributs de l'association peuvent toujours être déplacés dans une des classes qui participent à l'association. Pour les associations **1** vers **N**, le déplacement est généralement possible vers la classe du côté **N** ; toutefois, il est fréquent de promouvoir l'association au rang de classe pour augmenter la lisibilité ou en raison de la présence d'associations vers d'autres classes. L'exemple suivant illustre les différentes situations.

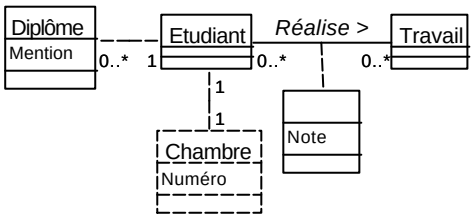


Figure 153 : Exemples de placement des attributs selon les valeurs de multiplicité.

L'association entre la classe **Etudiant** et la classe **Travail** est de type **N** vers **N**. La classe **Travail** décrit le sujet, la solution apportée par l'étudiant n'est pas conservée.

Dans le cas des contrôles de connaissances, chaque étudiant compose individuellement sur un travail donné et la note obtenue ne peut être stockée ni *dans* un étudiant en particulier (car il effectue de nombreux travaux dans sa carrière), ni *dans* un travail donné (car il y a autant de notes que d'étudiants). La

note est un attribut de la relation entre la classe des étudiants et la classe des travaux.

En fin d'année, chaque étudiant reçoit un diplôme avec une mention qui dépend de la performance individuelle de l'étudiant. La relation entre le diplôme et l'étudiant est personnalisée car un diplôme ne concerne qu'un étudiant donné. La mention devient un attribut du diplôme. La mention n'est pas stockée *dans* l'étudiant pour deux raisons : elle ne qualifie pas un étudiant et un étudiant peut obtenir plusieurs diplômes. Chaque étudiant possède une chambre et une chambre n'est pas partagée par plusieurs étudiants. L'association entre les étudiants et les chambres est du type **`1`** vers **`1`**. Le numéro est un attribut de la classe **`Chambre`** puisqu'un numéro caractérise une chambre.

Contraintes sur les associations

Toutes sortes de contraintes peuvent être définies sur une relation ou sur un groupe de relations. La multiplicité présentée dans le paragraphe précédent est une contrainte sur le nombre de liens qui peuvent exister entre deux objets. Les contraintes se représentent dans les diagrammes par des expressions placées entre accolades.

La contrainte **`{ordonnée}`** peut être placée sur le rôle pour spécifier qu'une relation d'ordre décrit les objets placés dans la collection ; dans ce cas, le modèle ne spécifie pas comment les éléments sont ordonnés, mais seulement que l'ordre doit être maintenu durant l'ajout et ou la suppression des objets par exemple.

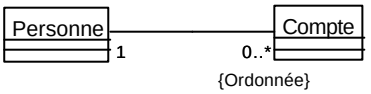


*Figure 154 : La contrainte **`{Ordonnée}`** indique que la collection des comptes d'une personne est ordonnée.*

La contrainte **`{Sous-ensemble}`** indique qu'une collection est incluse dans une autre collection. L'exemple suivant montre que les délégués de parents d'élèves sont des parents d'élèves.

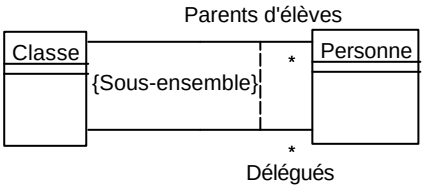


*Figure 155 : La contrainte **`{Sous-ensemble}`** indique que les objets qui participent à la relation **`Délégués`** participent également à la relation **`Parents d'élèves`**.*

La contrainte **`{ou-exclusif}`** précise que, pour un objet donné, une seule association parmi un groupe d'associations est valide. Cette contrainte évite l'introduction de sous-classes artificielles pour matérialiser l'exclusivité.

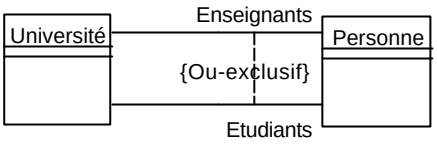


Figure 156 : Introduction d'une contrainte ***`{Ou-exclusif}`*** *pour distinguer deux associations mutuellement exclusives.*

Les associations peuvent également relier une classe à elle-même, comme dans le cas des structures récursives. Ce type d'association est appelé association réflexive. Le nommage des rôles prend à nouveau toute son importance pour distinguer les instances qui participent à la relation. L'exemple suivant montre la classe des personnes et la relation qui unie les parents et leurs enfants en vie.

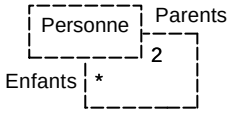


Figure 157 : Toute personne possède de zéro à deux parents, et de zéro à plusieurs enfants. Le nommage des rôles est essentiel à la clarté du diagramme.

Les classes-associations

Une association peut être représentée par une classe pour ajouter, par exemple, des attributs et des opérations dans l'association. Une classe de ce type, appelée parfois classe associative ou classe-association, est une classe comme les autres et peut à ce titre participer à d'autres relations dans le modèle. La notation utilise une ligne pointillée pour attacher une classe à une association. Dans l'exemple suivant, l'association entre les classes **`A`** et **`B`** est représentée par la classe **`C`**, qui elle-même est associée à la classe **`D`**.

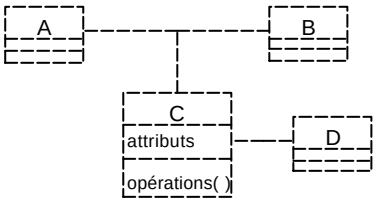


Figure 158 : Exemple de classe-association.

Une association qui contient des attributs sans participer à des relations avec d'autres classes est appelée association attribuée. Dans ce cas, la classe rattachée à l'association ne porte pas de nom spécifique.

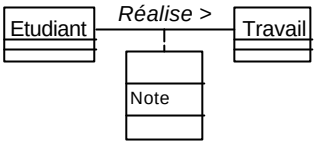


Figure 159 : Représentation d'une association attribuée.

Restriction des associations

La restriction (UML parle de *qualification*) d'une association consiste à sélectionner un sous-ensemble d'objets parmi l'ensemble des objets qui participent à une association. La restriction est réalisée au moyen d'un tuple d'attributs particuliers (appelé clé) et utilisé conjointement avec un objet de la classe de départ. La clé est représentée sur le rôle de la classe de départ, dans un compartiment rectangulaire. La clé appartient pleinement à l'association et non aux classes associées.

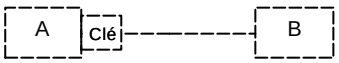


Figure 160 : Restriction d'une association, au moyen d'un attribut particulier appelé clé.

Chaque instance de la classe **`A`** accompagnée de la valeur de la clé, identifie un sous-ensemble des instances de **`B`** qui participent à l'association. La restriction réduit la multiplicité de l'association. La paire **`(instance de A, valeur de clé)`** identifie un sous-ensemble des instances de **`B`**. Très souvent, la multiplicité est réduite à la valeur **`1`**.

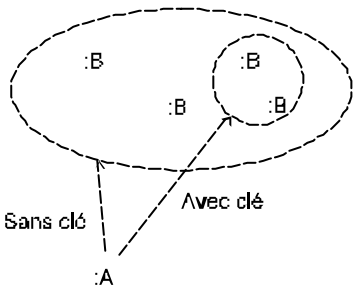


Figure 161 : Une restriction réduit le nombre d'instances qui participent à une association.

La restriction d'une association peut être opérée en combinant les valeurs des différents attributs qui forment la clé.

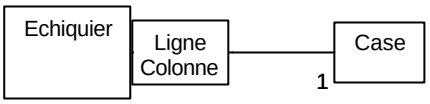


Figure 162 : Combinaison d'une ligne et d'une colonne pour identifier une case de l'échiquier.

Les agrégations

Une agrégation représente une association non symétrique dans laquelle une des extrémités joue un rôle prédominant par rapport à l'autre extrémité. Quelle que soit l'arité l'agrégation ne concerne qu'un seul rôle d'une association.

L'agrégation se représente en ajoutant un petit losange du côté de l'agrégat.

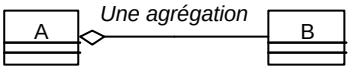


Figure 163 : Représentation des agrégations.

Les critères suivants impliquent une agrégation :

- une classe fait partie d'une autre classe ;
- les valeurs d'attributs d'une classe se propagent dans les valeurs d'attributs d'une autre classe ;
- une action sur une classe implique une action sur une autre classe ;
- les objets d'une classe sont subordonnés aux objets d'une autre classe.

L'inverse n'est pas toujours vrai : l'agrégation n'implique pas nécessairement les critères évoqués plus haut. Dans le doute, les associations sont préférables. De manière générale, il faut toujours choisir la solution qui implique le couplage le plus faible.

Les agrégations peuvent être multiples, comme les associations. Tant qu'aucun choix de réalisation n'a été effectué, il n'y a aucune contrainte particulière sur les valeurs de multiplicité que peuvent porter les rôles d'une agrégation. Cela signifie, en particulier, que la multiplicité du côté de l'agrégat peut être supérieure à 1. Ce type d'agrégation correspond par exemple au concept de copropriétaire. Le diagramme suivant montre que des personnes peuvent être copropriétaires des mêmes immeubles.

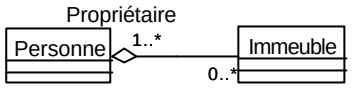


Figure 164 : Exemple d'agrégat multiple.

La notion d'agrégation ne suppose aucune forme de réalisation particulière. La contenance physique est un cas particulier de l'agrégation, appelé composition.

La composition

Les attributs constituent un cas particulier d'agrégation réalisée par valeur : ils sont physiquement contenus par l'agrégat. Cette forme d'agrégation est appelée composition, et se représente dans les diagrammes par un losange de couleur noire.

La composition implique une contrainte sur la valeur de la multiplicité du côté de l'agrégat : elle ne peut prendre que les valeurs **`0`** ou **`1`**. La valeur **`0`** du côté du composant correspond à un attribut non renseigné.

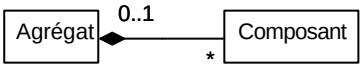


Figure 165 : Représentation graphique de la composition.

La composition et les attributs sont sémantiquement équivalents ; leurs représentations graphiques sont interchangeables. La notation par composition s'emploie dans un diagramme de classes lorsqu'un attribut participe à d'autres relations dans le modèle. La figure suivante illustre l'équivalence entre la notation par attribut et la notation par composition.

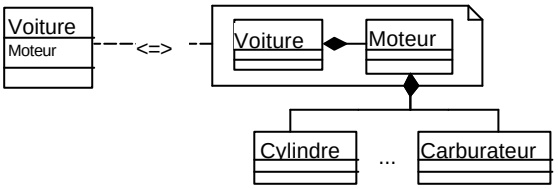


Figure 166 : Une agrégation par valeur est sémantiquement équivalente à un attribut ; elle permet de montrer comment un attribut participe à d'autres relations dans le modèle.

Les classes réalisées par composition sont également appelées classes composites. Ces classes fournissent une abstraction de leurs composants.

La navigation

Les associations décrivent le réseau de relations structurelles qui existent entre les classes et qui donnent naissance aux liens entre les objets, instances de ces classes. Les liens peuvent être vus comme des canaux de navigation entre les objets. Ces canaux permettent de se déplacer dans le modèle et de réaliser les formes de collaboration qui correspondent aux différents scénarios.

Par défaut, les associations sont navigables dans les deux directions. Dans certains cas, seule une direction de navigation est utile ; ceci se représente par une flèche portée par le rôle vers lequel la navigation est possible. L'absence de flèche signifie que l'association est navigable dans les deux sens. Dans l'exemple suivant, les objets instances de **A** voient les objets instances de **B**, mais les objets instances de **B** ne voient pas les objets instances de **A**.

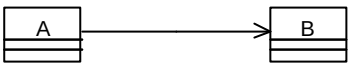


*Figure 167 : L'association entre les classes **A** et **B** est uniquement navigable de **A** vers **B**.*

Une association navigable uniquement dans un sens peut être vue comme une demi-association. Cette distinction est souvent réalisée pendant la conception, mais peut très bien apparaître en analyse, lorsque l'étude du domaine révèle une dissymétrie des besoins de communication.

Expressions de navigation

UML définit un pseudo-langage pour représenter les chemins au sein des diagrammes de classes. Ce langage très simple définit des expressions qui servent par exemple à préciser des contraintes.

La partie gauche de l'expression désigne un ensemble d'objets, éventuellement réduit à un singleton. Un nom de classe désigne l'ensemble des objets instances de la classe.

La syntaxe des expressions de navigation est donnée par les quatre règles suivantes :

- `cible ::= ensemble '.' sélecteur`

Le sélecteur correspond soit à un nom d'attribut des objets de l'ensemble, soit à un nom d'association de la classe des objets, soit à un nom de rôle opposé sur un lien qui concerne les objets de l'ensemble. La cible est un ensemble de valeurs ou d'objets, dont le nombre dépend de la multiplicité de l'ensemble et de l'association.

Dans le cas d'une association, l'expression retourne une collection d'objets qui contient le nombre d'éléments spécifiés par la multiplicité du rôle. L'expression **`UnePersonne.Enfants`** désigne toutes les enfants d'une personne donnée.

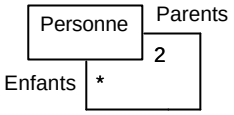


Figure 168 : L'expression ***`UnePersonne.Enfants`*** *désigne tous les enfants d'une personne donnée.*

- `cible ::= ensemble '.' '~' sélecteur`

Le sélecteur correspond à un nom de rôle placé du côté de l'ensemble. La cible est un ensemble d'objets, obtenu par navigation dans le sens opposé au nom de rôle.

Dans l'exemple précédent, les parents d'une personne sont désignés par l'expression **`UnePersonne.~Enfants`**.

- `cible ::= ensemble '[' expression_booléenne ']'`

L'expression booléenne est construite à partir des objets contenus par l'ensemble et à partir des liens et valeurs accessibles par ces objets. La cible est un ensemble d'objets qui vérifient l'expression. La cible est un sous-ensemble de l'ensemble de départ.

Dans l'exemple précédent, l'expression booléenne **`UnePersonne.Enfants [âge>=18ans]`** désigne tous les enfants majeurs d'une personne donnée.

- `cible ::=`

  `ensemble '.' sélecteur '[' valeur_de_clé ']'`

Le sélecteur désigne une association qui partitionne l'ensemble à partir de la valeur de la clé. La cible est un sous-ensemble de l'ensemble défini par l'association.

Dans l'exemple suivant, une restriction est opérée par l'ajout de clés sur l'association, ce qui a pour effet de réduire la multiplicité. L'expression générale **`UnePersonne.Enfant[UnPrénom]`** identifie un enfant donné de manière non ambiguë puisque chaque enfant d'une même famille a un prénom différent.

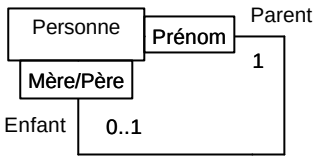


Figure 169 : Les valeurs de clés peuvent également être utilisées dans les expressions de navigation.

Les expressions précédentes se prêtent particulièrement bien à la formalisation des pré- et post-conditions attachées aux spécifications des opérations. Dans l'exemple précédent, les opérations de création des liens familiaux doivent associer correctement les parents et leurs enfants. Ceci peut s'exprimer sous la forme de la contrainte suivante.

```
{UnePersonne =
UnePersonne.Enfant[UnPrénom].(Parent[Mère] ou
Parent[Père])}
```

ou encore, en naviguant en sens inverse :

```
{UnePersonne =
UnePersonne.Enfant[UnPrénom].(~Enfant[Mère] ou
~Enfant[Père])}
```

ou enfin, en posant :

```
{papa = UnePersonne.Parent[Père]}
```

et

```
{papi = UnePersonne.(Parent[Père] ou
Parent[Mère]).Parent[Père]}
```

alors

```
{papi.Parent[Père] =
papa.Parent[Père].Parent[Père]}
```

ce qui revient à dire, comme chacun sait, que le papa de papi est le papi de papa !

La généralisation

UML emploie le terme généralisation pour désigner la relation de classification entre un élément plus général et un élément plus spécifique. En fait, le terme généralisation désigne un point de vue porté sur un arbre de classification. Par exemple, un animal est un concept plus général qu'un chat, un chien ou un raton laveur. Inversement, un chat est un concept plus spécialisé qu'un animal.

L'élément plus spécifique peut contenir des informations qui lui sont propres, à condition de rester totalement cohérent avec la description de l'élément plus général. La généralisation s'applique principalement aux classes, aux paquetages et aux cas d'utilisation.

Dans le cas des classes, la relation de généralisation exprime le fait que les éléments d'une classe sont aussi décrits par une autre classe (en fait par le type d'une autre classe). La relation de généralisation signifie *est un* ou *est une sorte de*. Un chat *est un* animal, il s'agit de généralisation. Un chat *a* deux oreilles, il ne s'agit pas de généralisation, mais de composition.

La relation de généralisation se représente au moyen d'une flèche qui pointe de la classe plus spécialisée vers la classe plus générale. La tête de la flèche est réalisée par un petit triangle vide, ce qui permet de la distinguer d'une flèche ouverte, symbole de la propriété de navigation des associations. Dans l'exemple suivant, la classe **`Animal`** est une abstraction des classes **`Chat`**, **`Chien`** et **`Raton laveur`**.

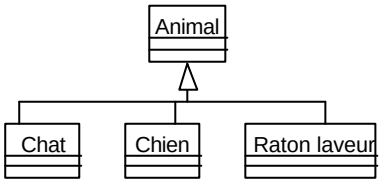


Figure 170 : Représentation de la relation de généralisation entre classes.

Dans l'exemple précédent, la classe **`Animal`** est appelée super-classe et les classes **`Chat`**, **`Chien`** et **`Raton laveur`** sont dites sous-classes de la classe **`Animal`**.

Dans le cas de sous-classes multiples, les flèches peuvent être agrégées en une seule, comme précédemment, ou représentées de manière indépendante, comme suit. Les deux formes de flèches qui symbolisent la généralisation ne véhiculent aucune sémantique particulière.

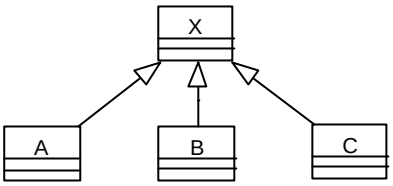


Figure 171 : Représentation de la généralisation au moyen de flèches obliques.

Les attributs, les opérations, les relations et les contraintes définies dans les super-classes sont hérités intégralement dans les sous-classes. En programmation, la relation de généralisation est très souvent réalisée en utilisant la relation d'héritage entre classes, proposé par les langages objet. L'héritage est une manière de réaliser la classification, mais ce n'est pas la seule. La relation de généralisation définie par UML est plus abstraite que la relation d'héritage telle qu'elle existe dans les langages de programmation objet comme C++. L'héritage est une relation statique ; elle induit un couplage très fort entre classes, et se révèle peu adaptée à la notion de classification dynamique ou à la notion de métamorphose. En analyse, il vaut mieux parler de généralisation ou de classification, et se préoccuper plus tard, lors de la conception, de la manière de réaliser la généralisation.

Les classes peuvent avoir plusieurs super-classes ; dans ce cas, la généralisation est dite multiple et plusieurs flèches partent de la sous-classe vers les différentes super-classes. La généralisation multiple consiste à fusionner plusieurs classes en une seule classe. Les super-classes n'ont pas nécessairement un ancêtre commun. Dans l'exemple suivant, la classe **`Tapis volant`** possède deux ancêtres complètement disjoints, les classes **`Tapis`** et **`Véhicule`**. La forme de généralisation qui regroupe les classes **`Véhicule`**, **`Véhicule terrestre`**[13], **`Véhicule Aérien`** et **`Tapis volant`**, est appelée généralisation en diamant ou en losange.

[13] Pour se convaincre que les tapis sont bien des véhicules terrestres, il suffit d'observer mes enfants...

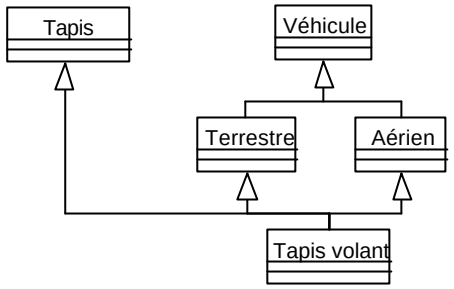


Figure 172 : Exemple de généralisation multiple.

Une classe peut être spécialisée selon plusieurs critères simultanément. Chaque critère de la généralisation est indiquée dans le diagramme en associant un discriminant à la relation de généralisation. Lorsque les flèches sont agrégées, le discriminant n'apparaît qu'une seule fois. L'absence de discriminant est considérée comme un cas particulier de généralisation.

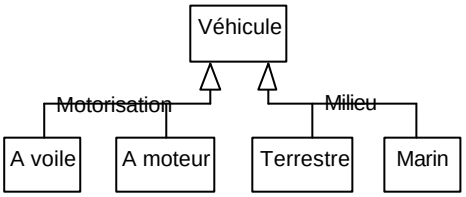


Figure 173 : Exemple de généralisation selon des critères indépendants.

Différentes contraintes peuvent être appliquées aux relations de généralisation, pour distinguer par exemple les formes de généralisation exclusives des formes inclusives. Les contraintes sur les relations de généralisation se représentent au moyen d'expressions entre parenthèses, directement attachées aux généralisations agrégées ou associées à une ligne pointillé qui relie les relations de généralisation concernées.

Par défaut, la généralisation symbolise une décomposition exclusive, c'est-à-dire qu'un objet est au plus instance d'une des sous-classes. La contrainte **`{Disjoint}`** ou **`{Exclusif}`** indique qu'une classe descendante d'une classe **A** peut être descendante d'une seule sous-classe de **A**.

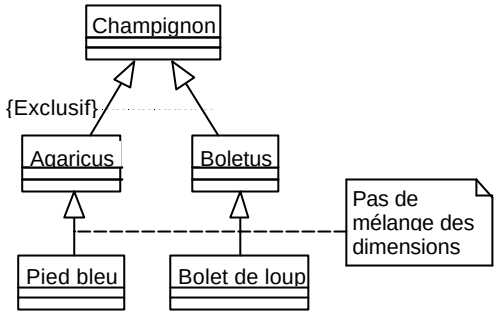


Figure 174 : Exemple de classification exclusive.

La contrainte **{Chevauchement}** ou **{Inclusif}** indique qu'une classe descendante d'une classe **A** appartient au produit cartésien des sous-classes de la classe **A**. Un objet concret est alors construit à partir d'une classe obtenue par mélange de plusieurs super-classes.

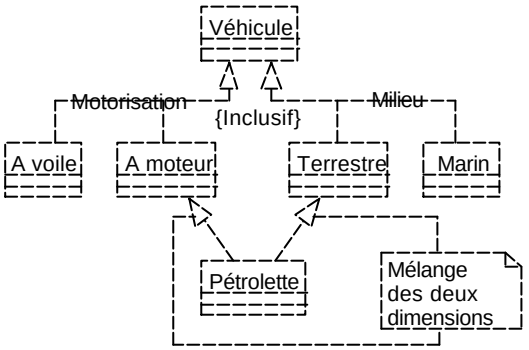


Figure 175 : Exemple de classification inclusive.

La contrainte **{Complète}** indique que la généralisation est terminée et qu'il n'est pas possible de rajouter des sous-classes. Inversement, la contrainte **{Incomplète}** désigne une généralisation extensible.

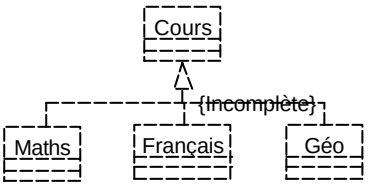


Figure 176 : La contrainte ***{Incomplète}*** *signale qu'il peut exister d'autres cours ; la classification est extensible.*

Il ne faut pas confondre la contrainte **`{Incomplète}`** avec une vue incomplète. Une contrainte véhicule un contenu sémantique qui affecte le modèle. Une vue incomplète ne signifie pas que le modèle est incomplet.

De manière générale, une vue partielle se symbolise par la présence de points d'élision qui se substituent aux éléments de modélisation dans les différents diagrammes. L'exemple suivant montre partiellement la généralisation de la classe **`Cours`**.

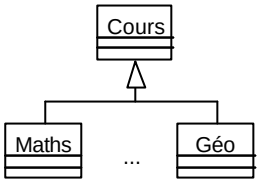


Figure 177 : Exemple de vue incomplète. Les points d'élision caractérisent la vue et non le modèle.

Les classes abstraites

Les classes abstraites ne sont pas instanciables directement ; elles ne donnent pas naissance à des objets, mais servent de spécification plus générale – de type – pour manipuler les objets instances d'une (ou plusieurs) de leurs sous-classes.

Les classes abstraites forment une base pour les logiciels extensibles. L'ensemble des mécanismes généraux est décrit dans les termes des spécifications des classes abstraites, sans tenir compte des spécificités rassemblées dans les classes concrètes. Les nouveaux besoins, les extensions et les améliorations sont concentrés dans de nouvelles sous-classes, dont les objets peuvent être manipulés de manière transparente par les mécanismes déjà en place.

Une classe est désignée comme abstraite au moyen de la propriété booléenne **`Abstraite`** définie pour tous les éléments généralisables (les types, les paquetages et les stéréotypes). Par convention, le nom des classes abstraites est en italique.

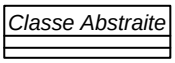


Figure 178 : Représentation graphique d'une classe abstraite.

La propriété abstraite peut également être appliquée à une opération afin d'indiquer que le corps de l'opération doit être défini explicitement dans les sous-classes.

Introduction au métamodèle

De nombreux éléments de modélisation présentent une dichotomie commune **(essence, manifestation)**, par laquelle un type représente l'essence d'une abstraction et l'instance fournit une manifestation concrète. Il existe également une dichotomie **(spécification, réalisation)** par laquelle des classes et des types primitifs réalisent des types.

Un type spécifie un domaine de valeurs et un ensemble d'opérations applicables à ces valeurs. Une classe réalise un type : elle fournit la représentation des attributs et la réalisation des opérations (les méthodes). Cette distinction se propage avec les sous-classes de sorte que la spécification donnée par un type est valable pour toutes les sous-classes et qu'une sous-classe peut réaliser plusieurs types.

Le diagramme de classes ci-dessous représente la dichotomie **(essence, manifestation)** et la dichotomie **(spécification, réalisation)**.

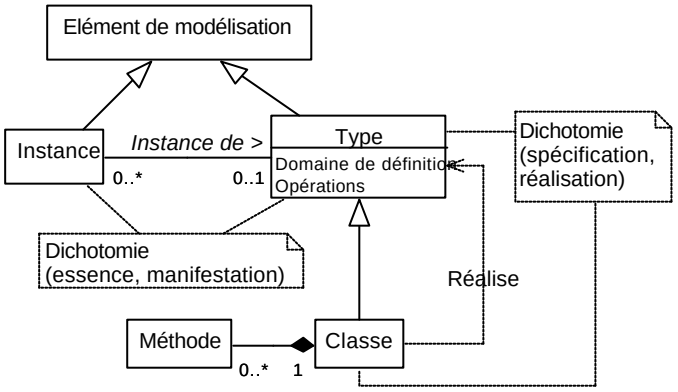


Figure 179 : Extrait du métamodèle. Une instance manifeste un type, une classe réalise un type.

La classe **Type** comprend les sous-classes suivantes :

- le *type primitif* qui correspond à un type qui n'est pas réalisé par une classe, comme les entiers ou les types énumérés,
- la *classe* qui fournit la réalisation d'un type,
- le *cas d'utilisation* qui est une séquence d'actions effectuées par un acteur et un système.

Le diagramme suivant représente les différentes sortes de types proposés par UML.

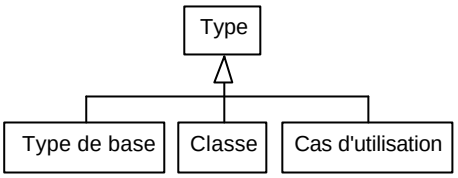


Figure 180 : Extrait du métamodèle. Représentation des différentes sortes de types.

La classe **`Classe`** comprend les sous-classes suivantes :

- la *classe active* qui réifie un ou plusieurs flots d'exécution,
- le *signal* qui est un événement nommé,
- le *composant* qui est un élément réutilisable contenant les constituants physiques des éléments de modélisation,
- le *nœud* qui est un dispositif physique sur lequel des composants peuvent être déployés pour leur exécution.

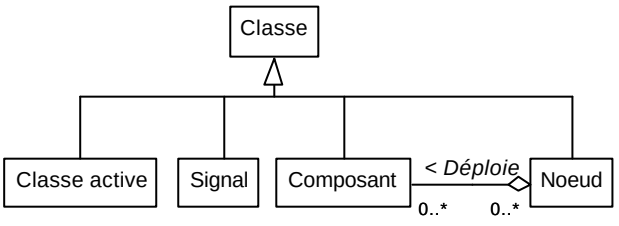


Figure 181 : Extrait du métamodèle. Représentation des différentes sortes de classes.

Les relations

UML définit cinq sortes de relations. La plus générale est appelée relation de dépendance et s'applique de manière uniforme à tous les éléments de modélisation. L'association et la généralisation concernent tous les types, en particulier les classes et les cas d'utilisation. Les deux dernières, les transitions et les liens, s'appliquent à certains éléments de modélisation du comportement.

Ce paragraphe se limite à la description des relations statiques entre classes, présentées dans le diagramme suivant. Les liens et les transitions sont présentés plus loin dans ce chapitre.

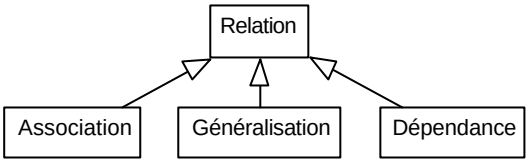


Figure 182 : Extrait du métamodèle. Représentation des relations entre classes.

L'association

L'association spécifie une connexion sémantique bidirectionnelle entre types. Une association possède au moins deux rôles qui décrivent la part prise par les types participant à l'association.

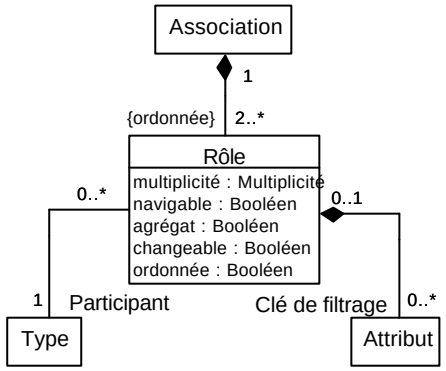


Figure 183 : Extrait du métamodèle. Représentation des principales caractéristiques des associations.

Chaque rôle comprend les attributs suivants :

- *Multiplicité*. Spécifie le nombre d'instances qui participent à la relation.
- *Navigable*. Précise si les liens, instances de l'association, sont navigables dans la direction du rôle considéré.
- *Agrégat*. Spécifie si les instances du type associé au rôle sont le tout dans une relation du genre **`(tout, parties)`**. Seul un des rôles d'une association peut posséder un attribut d'agrégation positionné à **`Vrai`**. Si, de plus, la multiplicité de ce rôle vaut **`1`**, alors le tout possède les parties et la destruction du tout entraîne la destruction des parties. Si la multiplicité est supérieure à **`1`**, plusieurs instances jouent le rôle du tout et partagent les parties ; la destruction d'une des instances du tout n'entraîne pas la

destruction des parties. La destruction de toutes les instances du tout entraîne la destruction des parties.

- *Changeable*. Précise si la sémantique de l'association est préservée lorsqu'une instance, du type qui participe au rôle, est remplacée par une autre instance.
- *Ordonnée*. S'applique lorsque la valeur de la multiplicité est supérieure à **1**, pour signifier que les instances sont ordonnées.

Un rôle d'association peut également comprendre un tuple d'attributs dont la conjonction de valeurs réalise une partition de l'ensemble des objets de la classe associée.

La généralisation

La généralisation spécifie une relation de classification par laquelle une instance d'un sous-type peut être substituée à une instance d'un super-type. Un super-type peut avoir plusieurs sous-types et un sous-type peut avoir plusieurs super-types.

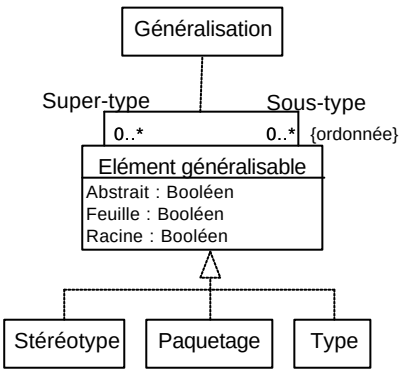


Figure 184 : Extrait du métamodèle. Représentation de la relation de généralisation et des éléments généralisables.

Tout élément généralisable possède les attributs booléens suivants :

- *Abstrait*. La valeur **Vrai** précise que l'élément ne peut être instancié directement.
- *Feuille*. La valeur **Vrai** précise que l'élément ne peut avoir des sous-types.
- *Racine*. La valeur **Vrai** précise que l'élément ne peut avoir des super-types.

UML définit trois sortes d'éléments généralisables :

- *Stéréotype.* Les stéréotypes permettent la classification d'un élément de modélisation et éventuellement la définition d'une représentation graphique particulière. Un stéréotype ne peut pas avoir de sous-type.
- *Paquetage.* Les paquetages offrent un mécanisme général pour le regroupement des éléments (à la fois de modélisation et de visualisation). Un paquetage ne peut pas avoir de super-type.
- *Type.* Un type spécifie un domaine de définition et les opérations applicables à ce domaine.

La dépendance

La dépendance est une relation d'utilisation unidirectionnelle entre éléments (de modélisation et de visualisation). La relation de dépendance relie des éléments au sein d'un même modèle.

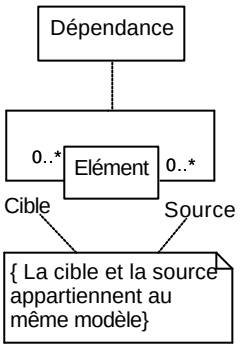


Figure 185 : Extrait du métamodèle. Représentation des relations de dépendance.

UML définit également une relation de trace entre éléments qui appartiennent à des modèles différents. La trace faciliter peut servir notamment à représenter l'histoire des constructions présentes dans les différents modèles. Cette relation de trace est une relation de dépendance stéréotypée.

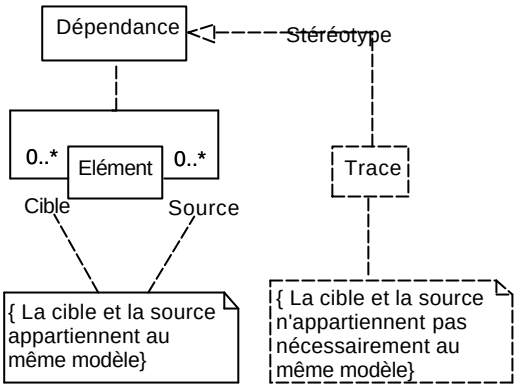


Figure 186 : Extrait du métamodèle. La relation de trace est une relation de dépendance stéréotypée.

Les cas d'utilisation

Les cas d'utilisation (*use cases*) ont été formalisés par Ivar Jacobson[14]. Les cas d'utilisation décrivent sous la forme d'actions et de réactions, le comportement d'un système du point de vue d'un utilisateur. Ils permettent de définir les limites du système et les relations entre le système et l'environnement.

Les cas d'utilisation viennent combler un manque des premières méthodes objet, comme OMT-1 et Booch 91, qui ne proposaient pas de technique pour la détermination des besoins. En ce sens, les cas d'utilisation associés aux techniques objet permettent une approche complète pour l'ensemble du cycle de vie, depuis le cahier des charges jusqu'à la réalisation.

Un cas d'utilisation est une manière spécifique d'utiliser un système. C'est l'image d'une fonctionnalité du système, déclenchée en réponse à la stimulation d'un acteur externe.

Intérêt des cas d'utilisation

La détermination et la compréhension des besoins sont souvent difficiles car les intervenants sont noyés sous de grandes quantités d'information. Les besoins sont souvent exprimés de manière non structurée, sans forte cohérence, de sorte que les cahiers des charges sont de longues litanies de paragraphes qui contiennent des expressions du genre :

[14] Jacobson I. 1992, *Object-Oriented Software Engineering, A Use Case Driven Approach*, Addison-Wesley.

- le système devra faire...
- le système devrait faire...
- le système fera éventuellement...
- il faut absolument que...
- il serait intéressant de...

Fréquemment, des besoins se contredisent, des oublis sont commis, des imprécisions subsistent, et l'analyse du système part sur de mauvaises bases. En conséquence, le cahier des charges initial est flou et en constante évolution. Lorsque les besoins se précisent ou évoluent – ce qui est toujours le cas – il devient très difficile d'apprécier l'impact et le coût d'une modification.

Les cas d'utilisation recentrent l'expression des besoins sur les utilisateurs, en partant du point de vue très simple qui veut qu'un système est avant tout construit pour ses utilisateurs. La structuration de la démarche s'effectue par rapport aux interactions d'une seule catégorie d'utilisateurs à la fois ; cette partition de l'ensemble des besoins réduit considérablement la complexité de la détermination des besoins.

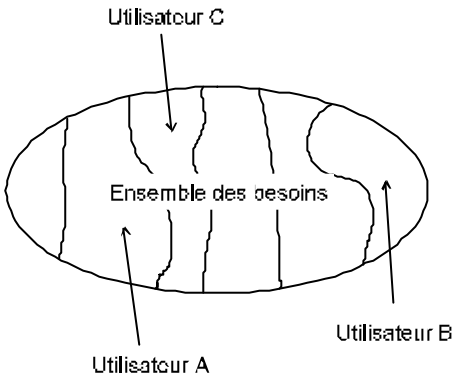


Figure 187 : Les cas d'utilisation partitionnent l'ensemble des besoins d'un système.

Le formalisme des cas d'utilisation – basé sur le langage naturel – est accessible sans formation particulière des utilisateurs, qui peuvent ainsi exprimer leurs attentes et leurs besoins en communiquant facilement avec les experts du domaine et les informaticiens. La terminologie employée dans la rédaction des cas d'utilisation est celle employée par les utilisateurs dans leur vie de tous les jours, de sorte que l'expression des besoins s'en trouve grandement facilitée. Les informaticiens, quant à eux, sont constamment ramenés par les cas d'utilisation vers les préoccupations premières des utilisateurs ; ils évitent ainsi de dériver vers des constructions techniquement séduisantes, mais pas forcément

nécessaires aux utilisateurs. Les cas d'utilisation focalisent l'effort de développement sur les vrais besoins.

Les cas d'utilisation permettent aux utilisateurs de structurer et d'articuler leurs désirs ; ils les obligent à définir la manière dont ils voudraient interagir avec le système, à préciser quelles informations ils entendent échanger et à décrire ce qui doit être fait pour obtenir le résultat escompté. Les cas d'utilisation concrétisent le futur système dans une formalisation proche de l'utilisateur ; ils favorisent la définition d'un cahier des charges qui reflète réellement les besoins, même en l'absence d'un système à critiquer.

Le modèle des cas d'utilisation

Le modèle des cas d'utilisation comprend les acteurs, le système et les cas d'utilisation eux-mêmes. L'ensemble des fonctionnalités d'un système est déterminé en examinant les besoins fonctionnels de chaque acteur, exprimés sous forme de familles d'interactions dans les cas d'utilisation. Les acteurs se représentent sous la forme de petits personnages qui déclenchent des cas d'utilisation ; ces derniers sont représentés par des ellipses contenues par le système.

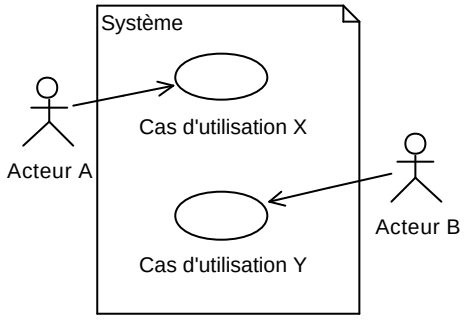


Figure 188 : Représentation des acteurs au moyen de petits personnages.

Un acteur représente un rôle joué par une personne ou une chose qui interagit avec un système. Les acteurs se déterminent en observant les utilisateurs directs du système, ceux qui sont responsables de son exploitation ou de sa maintenance, ainsi que les autres systèmes qui interagissent avec le système en question. La même personne physique peut jouer le rôle de plusieurs acteurs (vendeur, client). D'autre part, plusieurs personnes peuvent jouer le même rôle, et donc agir comme le même acteur (tous les clients). Le nom de l'acteur décrit le rôle joué par l'acteur.

Dans l'exemple suivant, Monsieur Schmoldu, garagiste de son état, passe le plus clair de son temps dans le rôle du mécanicien, mais peut à l'occasion jouer le rôle

de vendeur. Le dimanche, il joue le rôle de client et entretient sa voiture personnelle.

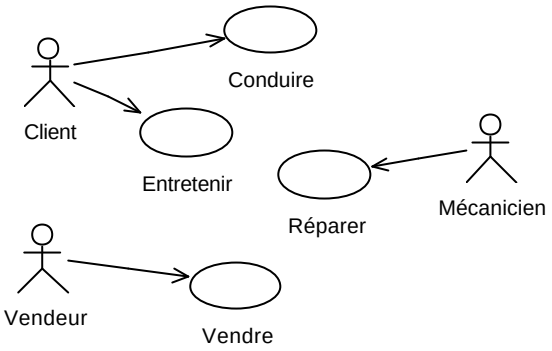


Figure 189 : Une même personne physique peut jouer le rôle de plusieurs acteurs.

Les candidats acteurs se recrutent parmi les utilisateurs, les clients, les partenaires, les fournisseurs, les vendeurs, les autres systèmes, en résumé, les personnes et les choses extérieures a un système qui interagissent avec lui en échangeant de l'information (en entrée et en sortie). La détermination des acteurs permet de préciser les limites du système de manière progressive : floues au départ, elles se précisent au fur et à mesure de l'élaboration des différents cas d'utilisation. Cette activité de délimitation est extrêmement importante car elle sert de base contractuelle où est signalé ce qui doit être fait, ce qui fait partie du système à développer et ce qui n'en fait pas partie. Elle précise également les éléments intangibles, que l'équipe de développement ne peut pas modifier.

Il existe quatre grandes catégories d'acteurs :

- *Les acteurs principaux.* Cette catégorie regroupe les personnes qui utilisent les fonctions principales du système. Dans le cas d'un distributeur de billets, il s'agit des clients.
- *Les acteurs secondaires.* Cette catégorie regroupe les personnes qui effectuent des tâches administratives ou de maintenance. Dans le cas d'un distributeur de billets, il s'agit de la personne qui recharge la caisse contenue dans le distributeur.
- *Le matériel externe.* Cette catégorie regroupe les dispositifs matériels incontournables qui font partie du domaine de l'application et qui doivent être utilisés. Il ne s'agit donc pas de l'ordinateur sur lequel s'exécute l'application, mais des autres dispositifs matériels périphériques. Dans le cas d'un distributeur de billets, ce peut être l'imprimante.

- *Les autres systèmes.* Cette catégorie regroupe les systèmes avec lesquels le système doit interagir. Dans le cas d'un distributeur de billet, le système du groupement bancaire qui gère un parc de distributeurs apparaît comme un acteur.

Une fois identifiés, les acteurs doivent être décrits d'une manière claire et concise, en trois ou quatre lignes maximum. Lorsqu'il y a beaucoup d'acteurs dans un système, il est conseillé de les regrouper par catégories afin de faciliter la navigation dans le modèle des cas d'utilisation.

Les cas d'utilisation se déterminent en observant et en précisant, acteur par acteur, les séquences d'interaction – les scénarios – du point de vue de l'utilisateur. Ils se décrivent en termes d'informations échangées et d'étapes dans la manière d'utiliser le système. Un cas d'utilisation regroupe une famille de scénarios d'utilisation selon un critère fonctionnel. Les cas d'utilisation sont des abstractions du dialogue entre les acteurs et le système : ils décrivent des interactions potentielles, sans entrer dans les détails de chaque scénario.

Les cas d'utilisation doivent être vus comme des classes dont les instances sont les scénarios. Chaque fois qu'un acteur interagit avec le système, le cas d'utilisation instancie un scénario ; ce scénario correspond au flot de messages échangés par les objets durant l'interaction particulière qui correspond au scénario. L'analyse des besoins par les cas d'utilisation s'accommode très bien d'une approche itérative et incrémentale.

La portée des cas d'utilisation dépasse largement la définition des seuls besoins du système. En effet, les cas d'utilisation interviennent tout au long du cycle de vie, depuis le cahier des charges jusqu'aux tests, en passant par l'analyse, la conception, la réalisation et la rédaction de la documentation pour l'utilisateur. De ce point de vue, il est possible de naviguer vers les classes et les objets qui collaborent pour satisfaire un besoin, puis vers les tests qui vérifient que le système s'acquitte correctement de sa tâche.

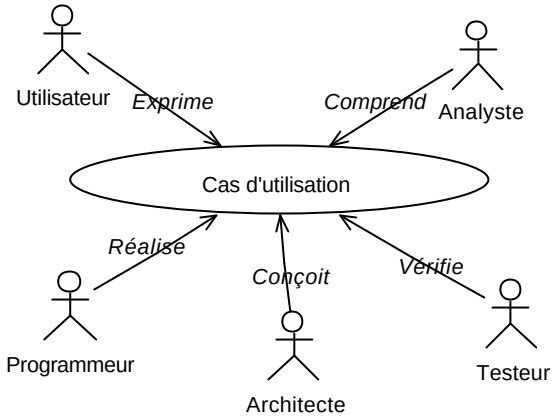


Figure 190 : Les cas d'utilisation servent de fil conducteur pour l'ensemble du projet.

Les relations entre cas d'utilisation

Les diagrammes de cas d'utilisation représentent les cas d'utilisation, les acteurs et les relations entre les cas d'utilisation et les acteurs. UML définit trois type de relations entre acteurs et cas d'utilisation :

- *La relation de communication.* La participation de l'acteur est signalée par une flèche entre l'acteur et le cas d'utilisation. Le sens de la flèche indique l'initiateur de l'interaction.

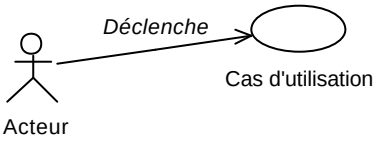


Figure 191 : Représentation du déclenchement d'un cas d'utilisation par un acteur.

- *La relation d'utilisation.* Une relation d'utilisation entre cas d'utilisation signifie qu'une instance du cas d'utilisation source comprend également le comportement décrit par le cas d'utilisation destination.

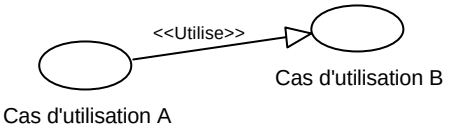


Figure 192 : Représentation de la relation d'utilisation au moyen d'une relation de généralisation stéréotypée.

- *La relation d'extension.* Une relation d'extension entre cas d'utilisation signifie que le cas d'utilisation source étend le comportement du cas d'utilisation destination.

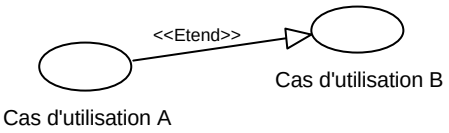


Figure 193 : Représentation de la relation d'extension au moyen d'une relation de généralisation stéréotypée.

Le diagramme suivant donne un exemple de mise en œuvre des différentes relations entre cas d'utilisation. Le virement par Minitel est une extension du virement effectué en agence. Dans les deux cas, le client doit être identifié.

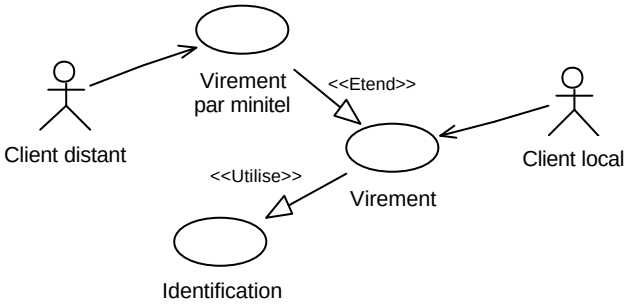


Figure 194 : Exemple de mise en œuvre des différentes relations entre cas d'utilisation.

Construction des cas d'utilisation

Un cas d'utilisation possède une valeur ajoutée qui procure à l'utilisateur un résultat appréciable. Un cas d'utilisation doit avant tout être simple, intelligible, décrit de manière claire et concise. Le nombre d'acteurs qui interagissent avec le cas d'utilisation est très limité : en règle générale, il n'y a qu'un seul acteur par cas d'utilisation.

Lors de la construction des cas d'utilisation, il faut se demander :

- quelles sont les tâches de l'acteur ?
- quelles informations l'acteur doit-il créer, sauvegarder, modifier, détruire ou simplement lire ?
- l'acteur devra-t-il informer le système des changements externes ?
- le système devra-t-il informer l'acteur des conditions internes ?

Les cas d'utilisation peuvent être présentés aux travers de vues multiples : un acteur avec tous ses cas d'utilisation, un cas d'utilisation avec tous ses acteurs, etc.

Les cas d'utilisation peuvent également être groupés selon leurs séquences de déclenchement types, c'est-à-dire en fonction de leurs enchaînements, ou encore en fonction des différents points de vue, souvent corrélés aux grandes catégories d'acteurs (client, fournisseur, maintenance...).

L'étape suivante consiste à classer les cas d'utilisation selon leur acuité par rapport à l'activité de la société, leur intérêt pour les clients, la nature des risques, la simplicité de leur mise en œuvre ou leur impact sur les processus déjà en place.

Règles de mise en œuvre des cas d'utilisation

Un cas d'utilisation décrit non seulement une fonctionnalité ou une motivation, mais aussi une interaction entre un acteur et un système sous la forme d'un flot d'événements. La description de l'interaction se concentre sur ce qui doit être fait dans le cas d'utilisation, pas sur la manière de le faire.

Un cas d'utilisation doit rester simple. Il ne faut pas hésiter à fragmenter un cas d'utilisation si les interactions deviennent trop complexes, trop enchevêtrées, ou encore si des parties se révèlent être indépendantes. Il faut également veiller à ne pas mélanger les cas d'utilisation ; les acteurs et les interactions d'un autre cas d'utilisation ne doivent pas être mentionnés dans la description. Enfin, les cas d'utilisation doivent éviter d'employer des expressions floues et imprécises, construites à partir de mots comme beaucoup, plutôt, suffisamment, assez, etc.

La description d'un cas d'utilisation comprend les éléments suivants :

- *le début du cas d'utilisation*, autrement dit l'événement qui déclenche le cas d'utilisation ; il doit être clairement stipulé dans la phrase suivante : « Le cas d'utilisation débute quand **X** se produit. » ;
- *la fin du cas d'utilisation*, autrement dit l'événement qui cause l'arrêt du cas d'utilisation ; il doit être clairement identifié et documenté par la phrase : « Quand **Y** se produit, le cas d'utilisation est terminé. » ;
- *l'interaction entre le cas d'utilisation et les acteurs* qui décrit clairement ce qui est dans le système et ce qui est hors du système ;

- *les échanges d'informations* qui correspondent aux paramètres des interactions entre le système et les acteurs. Une formulation non informatique est recommandée, par exemple : « L'utilisateur se connecte au système et donne son nom et son mot de passe. » ;
- *La chronologie et l'origine des informations* qui décrivent quand le système a besoin d'informations internes ou externes et quand il les enregistre à l'intérieur ou à l'extérieur ;
- *Les répétitions de comportement* qui peuvent être décrites au moyen de pseudo-code, avec des constructions du type :

```
boucle
   -- quelque chose
fin de boucle
```

ou

```
pendant que
   -- autre chose
fin pendant
```

- *Les situations optionnelles* qui doivent être représentées de manière uniforme dans tous les cas d'utilisation ; les différentes options doivent apparaître clairement selon la formulation :

  « L'acteur choisit l'un des éléments suivants, éventuellement plusieurs fois de suite :

  a) choix **X**

  b) choix **Y**

  c) choix **Z**

  puis l'acteur continue en... »

Ces points ne suffisent pas pour obtenir un bon cas d'utilisation. Il est également primordial de trouver le bon niveau d'abstraction, c'est-à-dire la quantité de détails qui doivent apparaître, et de faire la distinction entre un cas d'utilisation et un scénario. Il n'y a malheureusement pas de réponse toute faite, de sorte que l'appréciation du niveau d'abstraction repose grandement sur l'expérience. Les réponses apportées aux deux interrogations suivantes peuvent néanmoins servir de gabarit :

- est-il possible d'exécuter une activité donnée indépendamment des autres, ou faut-il toujours l'enchaîner avec une autre activité ? Deux activités qui s'enchaînent toujours font probablement partie du même cas d'utilisation ;
- est-il judicieux de regrouper certaines activités en vue de les documenter, de les tester ou de les modifier ? Si oui, ces activités font sûrement partie du même cas d'utilisation.

Lorsqu'un cas d'utilisation devient trop complexe (plus de dix pages par exemple), il est possible de le découper en cas d'utilisation plus petits. Cette opération n'est toutefois effectuée que lorsque les cas d'utilisation ont été décrits en détail, après examen des principaux scénarios.

Processus d'élaboration des cas d'utilisation

Les cas d'utilisation permettent le travail en groupe, à condition de définir un guide de style pour la rédaction. Ce guide contient une description de la mise en page des cas d'utilisation, du niveau de détails, ainsi qu'un modèle de cas d'utilisation qui sera repris par tous les intervenants.

L'équipe commence par identifier grossièrement les cas d'utilisation. Les principales étapes de chaque cas d'utilisation sont décrites en une phrase ou deux, en distinguant le cas nominal des cas alternatifs et exceptionnels. Une fois ce premier travail réalisé, des groupes issus de l'équipe se chargent d'approfondir la compréhension et la description d'un cas d'utilisation particulier. Pour ce faire, chaque groupe identifie les scénarios et les conditions de différenciation entre scénarios à partir de la connaissance du domaine et de rencontres avec les utilisateurs.

Un scénario est un chemin particulier au travers de la description abstraite et générale fournie par le cas d'utilisation. Les scénarios traversent le cas d'utilisation, en suivant le chemin nominal ainsi que tout chemin alternatif et exceptionnel. L'explosion combinatoire fait qu'il est bien souvent impossible de dérouler tous les scénarios.

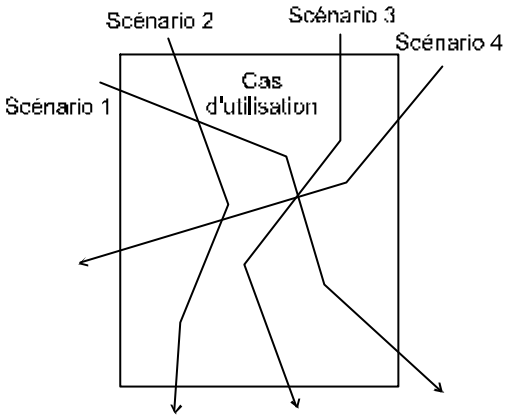


Figure 195 : Un cas d'utilisation décrit (de manière abstraite) une famille de scénarios.

Lorsque le contenu du cas d'utilisation a été validé par le déroulement des scénarios, les étapes stabilisées doivent être décrites de manière plus détaillée. A chaque étape correspond alors un paragraphe qui décrit ce qui se passe dans l'étape, en tenant compte a priori de tous les scénarios qui traversent l'étape. A ce stade, tous les besoins identifiés dans les scénarios doivent être pris en compte par un cas d'utilisation.

Il est fortement conseillé de synchroniser l'évolution des cas d'utilisation par des réunions régulières entre groupes de l'équipe. Ceci présente le double avantage d'harmoniser le niveau de détail des cas d'utilisation et de permettre la réallocation des scénarios orphelins, en mal de cas d'utilisation.

Ces réunions permettent également de faire valider les cas d'utilisation d'un groupe par les autres groupes. Des critères simples, issus de l'expérience, aident à distinguer un bon cas d'utilisation d'un moins bon. Il faut par exemple se poser les questions suivantes :

- existe-t-il une brève description qui donne une vraie image du cas d'utilisation ?
- les conditions de démarrage et d'arrêt du cas d'utilisation sont-elles bien cernées ?
- les utilisateurs sont-ils satisfaits par la séquence d'interactions entre l'acteur et le cas d'utilisation ?
- existe-t-il des étapes communes avec d'autres cas d'utilisation ?
- les mêmes termes employés dans des cas d'utilisation différents ont-ils bien la même signification ?
- est-ce que toutes les alternatives sont prises en compte par le cas d'utilisation ?

Les derniers pièges à éviter

Les cas d'utilisation ne sont pas la seule manière de documenter les besoins d'un système. Les techniques traditionnelles, à base de descriptions textuelles informelles, peuvent très bien accompagner les cas d'utilisation. Il est alors judicieux d'établir des références croisées entre les deux types de documentation et d'étudier les points suivants :

- est-ce que tous les besoins identifiés de manière informelle ont été alloués à un cas d'utilisation ?
- est-ce qu'un même besoin a été alloué à plusieurs cas d'utilisation ?
- existe-t-il des cas d'utilisation qui ne sont pas référencés par des besoins informels ?

D'autre part, le maquettage à base de constructeurs d'interface utilisateur est très souvent le meilleur moyen d'aider l'utilisateur final à articuler ses désirs. L'examen des interactions entre l'utilisateur et la maquette procure alors une source non négligeable de scénarios, et donc de moyens de valider le modèle des cas d'utilisation.

La principale difficulté d'emploi liée aux cas d'utilisation réside plus dans la détermination du niveau de détail, que dans la formulation des cas d'utilisation. Il faut se rappeler que les cas d'utilisation sont une abstraction d'un ensemble de comportements fonctionnellement liés au sein d'un cas d'utilisation. La présence d'un grand nombre de détails par exemple, est le signe d'un scénario plus que d'un cas d'utilisation.

Dans le même ordre d'idée, un grand nombre de cas d'utilisation est le signe d'un manque d'abstraction. Dans n'importe quel système, quelle que soit sa complexité et sa taille, il y a relativement peu de cas d'utilisation, mais beaucoup de scénarios. Typiquement, un système moyen comprend de dix à vingt cas d'utilisation ; un grand nombre de cas d'utilisation signifie que l'essence du système n'a pas été comprise. Un cas d'utilisation décrit ce que l'utilisateur veut fondamentalement faire avec le système. Le nom des cas d'utilisation peut être un indicateur, les associations (objet, opérations) ou (fonction, données) sont fortement suspectes.

Une autre difficulté consiste à résister à la tentation de trop décrire le comportement interne du système, au détriment de l'interaction entre l'acteur et le système. Il est vrai que le cas d'utilisation doit décrire ce que l'utilisateur échange avec le système et les grandes étapes dans l'élaboration de ces échanges ; mais cela ne va pas jusqu'à expliquer comment le système réalise ces échanges. Le cas d'utilisation est un outil d'analyse utilisé pour déterminer ce que l'utilisateur attend du système, c'est-à-dire le *quoi* et le *quoi faire* du système. Dès lors que la description fait appel au *comment*, il ne s'agit plus d'analyse mais de conception.

La transition vers les objets

Les cas d'utilisation recentrent le développement sur les besoins de l'utilisateur ; ils aident donc à construire le bon système (*Build the right system* comme diraient nos amis anglo-saxons). Les cas d'utilisation ne définissent pas pour autant la bonne manière de faire le système (*Build the system right* comme diraient toujours nos amis) ni la forme de l'architecture du logiciel ; ils disent ce qu'un système doit faire, pas comment il doit le faire.

La vue des cas d'utilisation est une description fonctionnelle des besoins, structurée par rapport à un acteur. Le passage à l'approche objet s'effectue en associant une collaboration à chaque cas d'utilisation. Une collaboration décrit des objets du domaine, les connexions entre ces objets et les messages échangés par les objets. Chaque scénario, instance du cas d'utilisation réalisé par la collaboration, se représente par une interaction entre les objets décrits dans le contexte de la collaboration.

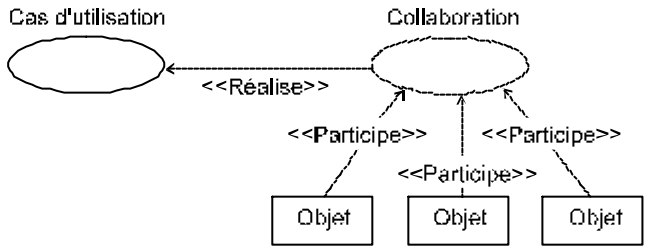


Figure 196 : La transition vers l'objet est effectuée en réalisant un cas d'utilisation par une collaboration.

La réalisation d'un cas d'utilisation par une collaboration est un instant crucial de la modélisation ; c'est le moment du virage vers l'objet. Une décomposition qui suit directement la forme des cas d'utilisation conduit à une approche structurée classique, avec tous les défauts des structures calquées sur les fonctions.

Une approche objet réalise un cas d'utilisation au moyen d'une collaboration entre objets. Les scénarios, instances du cas d'utilisation, sont représentés par des diagrammes d'interaction (diagrammes de collaboration et diagrammes de séquence).

La figure suivante illustre le choix de décomposition après l'étude des cas d'utilisation.

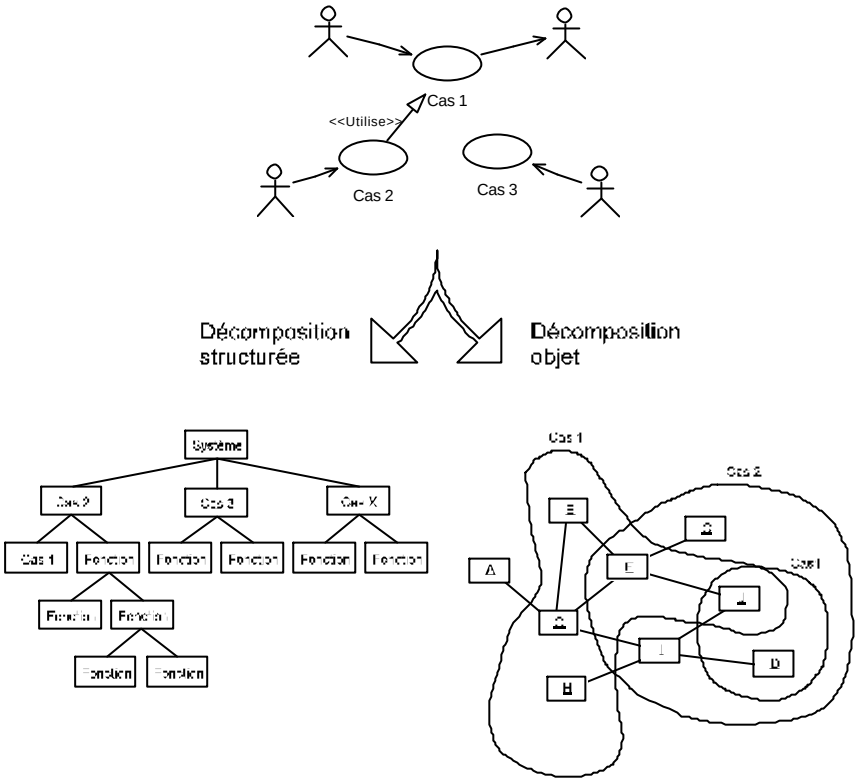


Figure 197 : Représentation du virage vers l'objet après une étude des cas d'utilisation.

Les diagrammes d'objets

Les diagrammes d'objets, ou diagrammes d'instances, montrent des objets et des liens. Comme les diagrammes de classes, les diagrammes d'objets représentent la structure statique. La notation retenue pour les diagrammes d'objets est dérivée de celle des diagrammes de classes ; les éléments qui sont des instances sont soulignés.

Les diagrammes d'objets s'utilisent principalement pour montrer un contexte, par exemple avant ou après une interaction, mais également pour faciliter la compréhension des structures de données complexes, comme les structures récursives.

Représentation des objets

Chaque objet est représenté par un rectangle qui contient soit le nom de l'objet, soit le nom et la classe de l'objet (séparés par un double point), soit seulement la classe de l'objet (l'objet est alors dit anonyme). Le nom seul correspond à une modélisation incomplète dans laquelle la classe de l'objet n'a pas encore été précisée. La classe seule évite l'introduction de noms inutiles dans les diagrammes, tout en permettant l'expression de mécanismes généraux, valables pour de nombreux objets. Le diagramme suivant illustre les trois possibilités de représentation.

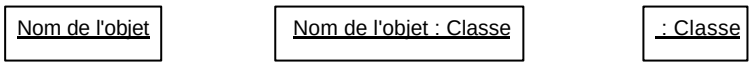


Figure 198 : Représentations graphiques d'un objet ; le nom ou la classe de l'objet peuvent être supprimés du diagramme.

Le nom de la classe peut contenir le chemin complet, composé à partir des noms des différents paquetages englobants séparés par des doubles deux points, comme dans l'exemple suivant :

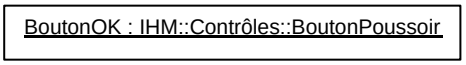


Figure 199 : Exemple de représentation du chemin d'accès complet de la classe de l'objet.

Le stéréotype de la classe peut être repris dans le compartiment de l'objet, soit sous la forme textuelle (entre guillemets au dessus du nom de l'objet), soit sous la forme graphique (dans le coin haut droit), soit encore au moyen d'une représentation graphique particulière qui se substitue au symbole de l'objet. Il n'existe pas de stéréotype d'objet ; le stéréotype qui figure dans un objet est toujours le stéréotype de la classe de l'objet.

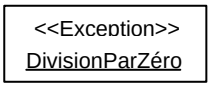


Figure 200 : Exemple de représentation du stéréotype de la classe dans le symbole d'un objet.

Les rectangles qui symbolisent les objets peuvent également comporter un deuxième compartiment qui contient les valeurs des attributs. Le type des attributs est déjà décrit dans la classe, de sorte qu'il n'est plus nécessaire de le faire figurer dans les représentations d'objets. Le diagramme suivant représente

un objet anonyme de la classe **`Voiture`**, muni d'un attribut **`Couleur`** dont la valeur est **`rouge`**.

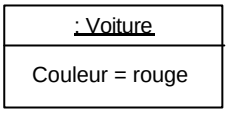


Figure 201 : Exemple d'objet anonyme de la classe ***`Voiture`****, muni d'un attribut* ***`Couleur`*** *dont la valeur est* ***`rouge`****.*

Représentation des liens

Les objets sont reliés par des liens qui sont instances des relations entre les classes des objets considérés. La représentation concrète d'une structure par des objets est souvent plus parlante que celle abstraite par des classes, surtout dans le cas de structures récursives.

Le diagramme d'objets suivant montre une partie de la structure générale des voitures. Chaque voiture possède un moteur et quatre roues (roue de secours exclue !).

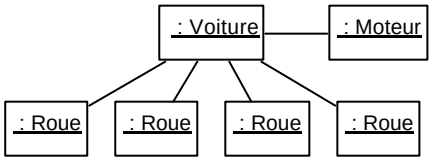


Figure 202 : Exemple de diagramme d'objets.

Le diagramme d'objets précédent est instance du diagramme de classes suivant.

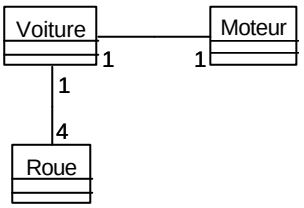


Figure 203 : Exemple de diagramme de classes. Représentation générale de la situation particulière montrée dans la figure précédente.

Les liens instances des associations réflexives peuvent relier un objet à lui-même. Dans ce cas, le lien est représenté par une boucle attachée à un seul objet. Le diagramme suivant représente deux liens, instances de la même association

réflexive. Le premier lien montre qu'Etienne est le patron de Jean-Luc, le deuxième lien montre que Denis est son propre patron.

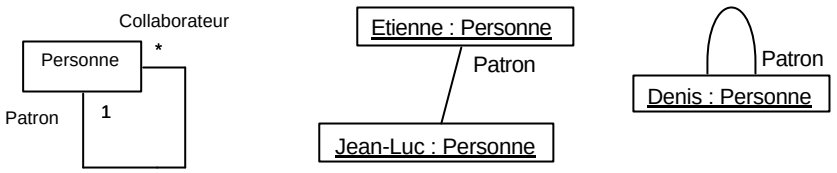


Figure 204 : Exemple de liens instances d'une relation réflexive. Etienne est le patron de Jean-Luc, Denis est son propre patron.

La plupart des liens sont binaires. Il existe toutefois certains liens d'arité supérieure, par exemple ceux qui correspondent aux relations ternaires. La représentation des liens ternaires peut se combiner avec les autres éléments de la notation : le diagramme suivant représente une famille de liens ternaires, instances d'une association ternaire de multiplicité **`N`** du côté de la classe **`Etudiant`**. Cette notation présente l'avantage de lever les ambiguïtés inhérentes à la représentation de la multiplicité des associations d'arité supérieure à 2.

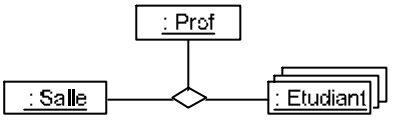


Figure 205 : Exemple de combinaison d'éléments de notation pour représenter de manière condensée des liens ternaires multiples.

Les objets composites

Les objets composés de sous-objets peuvent être représentés au moyen d'un objet composite, afin de réduire la complexité des diagrammes. Les objets composites se présentent comme des objets classiques, si ce n'est que les attributs sont remplacés par des objets, soit sous une forme textuelle soulignée, soit sous une forme graphique. Le diagramme suivant reprend la forme graphique des objets composites.

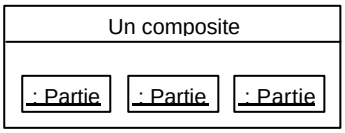


Figure 206 : Représentation graphique des objets composites.

Les objets composites sont instances de classes composites, c'est-à-dire de classes construites à partir d'autres classes par la plus forte forme d'agrégation. Le diagramme suivant représente une classe composite **`Fenêtre`**.

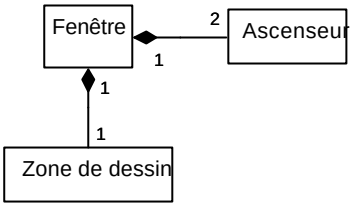


Figure 207 : Exemple d'une classe composite ***`Fenêtre`*** *qui contient une zone de dessin et deux ascenseurs.*

Le diagramme d'objets suivant est instance du diagramme de classe précédent : il représente la forme générale des objets composites **`Fenêtre`**, du point de vue des objets.

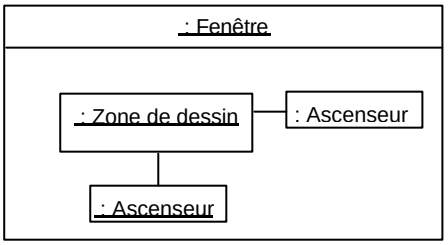


Figure 208 : Représentation d'un objet composite, instance de la classe composite décrite dans le diagramme précédent.

Similitudes avec les diagrammes de classes

Les décorations qui figurent dans les diagrammes de classes peuvent en grande partie être reportées dans les diagrammes d'objets, afin de faciliter la compréhension de l'interaction. Ceci concerne toutes les caractéristiques des associations (le nom, le nom des rôles, l'agrégation, la composition et la navigation), à l'exception de la multiplicité qui se représente de manière explicite par les liens. Le diagramme d'objets suivant se distingue graphiquement d'un diagramme de classe car les noms d'objets sont soulignés.

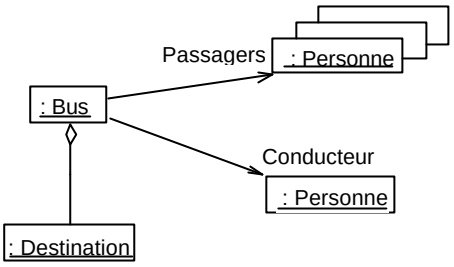


Figure 209 : Exemple de diagramme d'objets, décoré comme un diagramme de classes.

Les valeurs des clés de restriction des associations peuvent également être rajoutées dans les diagrammes d'objets. Le diagramme suivant représente les liens de parenté entre Lara, Jonathan, Roxane, Anne et Pierre-Alain.

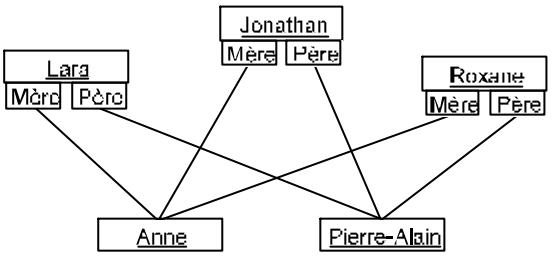


Figure 210 : Exemple de représentation des valeurs des clés de restriction des associations.

Les diagrammes de collaboration

Les diagrammes de collaboration montrent des interactions entre objets, en insistant plus particulièrement sur la structure spatiale statique qui permet la mise en collaboration d'un groupe d'objets. Les diagrammes de collaboration expriment à la fois le contexte d'un groupe d'objets (au travers des objets et des liens) et l'interaction entre ces objets (par la représentation des envois de messages). Les diagrammes de collaboration sont une extension des diagrammes d'objets.

Représentation des interactions

Le contexte d'une interaction comprend les arguments, les variables locales créées pendant l'exécution, ainsi que les liens entre les objets qui participent à l'interaction.

Une interaction est réalisée par un groupe d'objets qui collaborent en échangeant des messages. Ces messages sont représentés le long des liens qui relient les objets, au moyen de flèches orientées vers le destinataire du message.

Le diagramme suivant représente une cabine d'ascenseur qui demande à une porte de s'ouvrir.

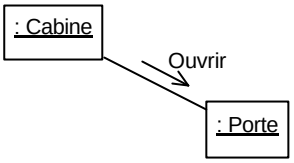


Figure 211 : Exemple d'interaction entre deux objets. La cabine demande à la porte de s'ouvrir.

Dans un diagramme de collaboration, le temps n'est pas représenté de manière implicite, comme dans un diagramme de séquence, de sorte que les différents messages sont numérotés pour indiquer l'ordre des envois.

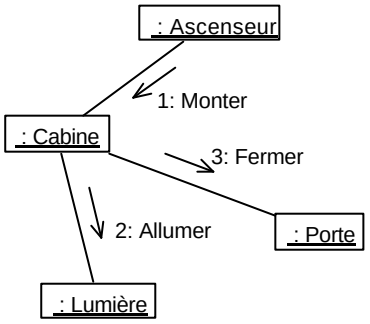


Figure 212 : Exemple de représentation de l'ordre des envois de message.

Les diagrammes de collaboration montrent simultanément les interactions entre les objets et les relations structurelles qui permettent ces interactions. Le diagramme suivant représente le contexte d'un mécanisme pour annuler une opération de destruction. Avant de déclencher l'opération de destruction de l'objet **B**, l'objet **A** effectue une copie locale de **B**, de sorte que si l'opération de destruction venait à être annulée, l'objet **B** puisse être restitué tel qu'il était avant le déroulement de l'interaction.

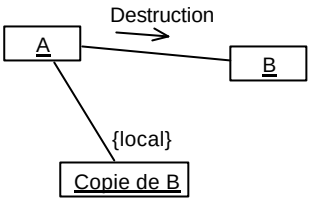


*Figure 213 : Représentation d'une interaction entre trois objets. L'objet **B** a été copié localement par l'objet **A**, dans le but de permettre l'annulation éventuelle de la destruction de **B**.*

Les objets et les liens créés ou détruits au cours d'une interaction peuvent respectivement porter les contraintes **{nouveau}** ou **{détruit}**. Les objets créés, puis détruits au sein de la même interaction, sont identifiés par la contrainte **{transitoire}**.

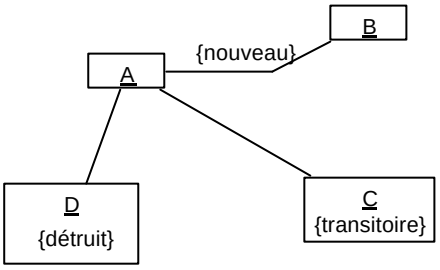


Figure 214 : Représentation de la création et de la destruction des objets et des liens.

La notation permet de représenter de manière condensée une famille de liens, instances d'une même association. Cette approche est particulièrement intéressante lorsque le groupe d'objets considéré est traité de manière uniforme, en étant par exemple destination d'un même message. L'exemple suivant montre un instituteur qui demande à tous ses élèves de se lever ; l'itération est indiquée par le caractère ***** placé devant le message.

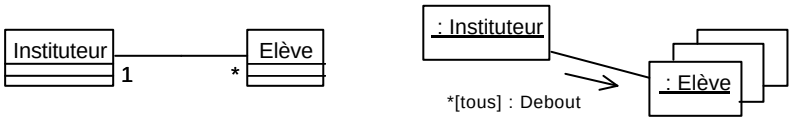


Figure 215 : Représentation condensée d'une famille de liens, instances d'une même association.

La place de l'utilisateur

La notation permet de faire figurer un acteur dans un diagramme de collaboration afin de représenter le déclenchement des interactions par un élément externe au système. Grâce à cet artifice, l'interaction peut être décrite de manière plus abstraite, sans entrer dans les détails des objets de l'interface utilisateur. Le premier message de l'interaction est envoyé par l'acteur, représenté soit par le symbole graphique des acteurs du modèle des cas d'utilisation, soit par un objet muni d'un stéréotype qui précise sa qualité d'acteur. Le diagramme suivant montre un fragment d'interaction ; elle correspond à un appel de cabine d'ascenseur par une personne.

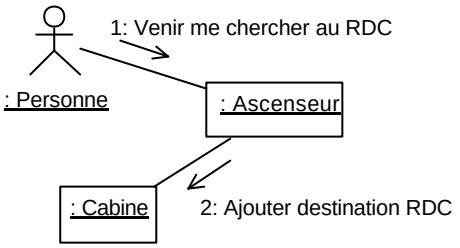


Figure 216 : Exemple de déclenchement d'une interaction par un acteur.

Les objets actifs

Les objets qui possèdent le flot de contrôle sont dits d'actifs. Un objet actif peut activer un objet passif pour le temps d'une opération, en lui envoyant un message. Une fois le message traité, le flot de contrôle est restitué à l'objet actif. Dans un environnement multitâche, plusieurs objets peuvent être actifs simultanément. Un objet actif se représente par un rectangle dont la bordure est plus épaisse que celle des objets passifs.

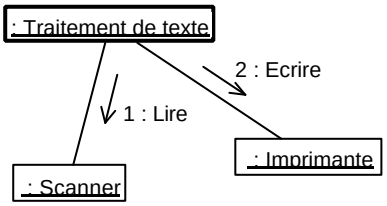


Figure 217 : Représentation des objets actifs. Le cadre des objets actifs est plus épais que celui des objets passifs.

Représentation des messages

Un message se représente par une flèche placée à proximité d'un lien et dirigée vers l'objet destinataire du message. Un lien sert de support de transmission pour le message. Un message déclenche une action dans l'objet destinataire.

Les messages respectent la forme générale suivante :

```
synchronisation séquence ':' résultat ':=' nom
   arguments
```

Le message, ses arguments et valeurs de retour, son rang au sein de l'interaction, et diverses autres informations comme le degré d'emboîtement ou la synchronisation sont précisés lors de l'envoi.

Synchronisation

Le point de synchronisation d'un message est exprimé sous la forme d'une séquence d'envoi de message, terminée par le caractère **/**. Tous les messages référencés dans cette liste doivent avoir été envoyés pour valider l'envoi du message courant.

La syntaxe d'un point de synchronisation prend la forme suivante :

```
synchronisation ::= rang {',' synchronisation} '/'

rang ::= [entier | nom de flot d'exécution]{'.' rang}
```

L'entier représente le rang de l'envoi de message au sein de l'emboîtement englobant. Le nom identifie un flot d'exécution parallèle au sein d'un emboîtement. Ainsi, l'envoi de message **3.1.3** suit immédiatement l'envoi **3.1.2** au sein de l'emboîtement **3.1**, alors que l'envoi **3.1.a** est effectué simultanément à l'envoi **3.1.b**.

Dans l'exemple suivant, le message **Message** est envoyé lorsque les envois **A.1** et **B.3** ont été satisfaits.

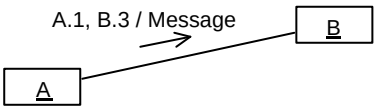


Figure 218 : Exemple de représentation de la synchronisation entre flots d'exécution parallèles.

Séquence

La séquence indique le niveau d'emboîtement de l'envoi de message au sein de l'interaction. La séquence est constituée d'une suite de termes séparés par des points. Chaque séquence possède la syntaxe suivante :

```
séquence ::= rang [récurrence]
```

La récurrence représente l'itération et les branchements conditionnels ; elle prend les formes suivantes :

```
récurrence ::= '*' '[' clause d'itération ']' bloc
```

ou

```
récurrence ::= '[' clause de condition ']' bloc
```

La clause d'itération est optionnelle ; elle est exprimée dans un format libre :

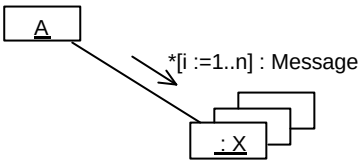


Figure 219 : Exemple de représentation de l'itération.

La notation de l'itération sous-entend l'envoi séquentiel des messages contenus par le bloc. L'envoi parallèle (également appelé diffusion) est matérialisé par la suite de caractères *** | |**.

La clause de condition valide ou non l'envoi des messages contenus par le bloc. La clause de condition est exprimée dans un format libre :

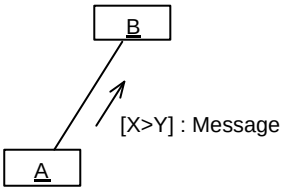


Figure 220 : Exemple de représentation d'un envoi de message conditionnel.

Résultat

Le résultat est constitué d'une liste de valeurs retournées par le message. Ces valeurs peuvent être utilisées comme paramètres des autres messages compris dans l'interaction. Ce champ n'existe pas en l'absence de valeurs retournées. Le format de ce champ est libre :

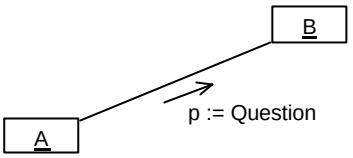


Figure 221 : Représentation de la récupération d'un résultat.

Nom

Il s'agit du nom du message. Souvent ce nom correspond à une opération définie dans la classe de l'objet destinataire du message.

Arguments

Il s'agit de la liste des paramètres du message. Les arguments et le nom du message identifient de manière unique l'action qui doit être déclenchée dans l'objet destinataire. Les arguments peuvent contenir des valeurs retournées par des messages envoyés précédemment, ainsi que des expressions de navigation construites à partir de l'objet source.

Les expressions suivantes donnent quelques exemples de la syntaxe d'envoi des messages :

```
4 : Afficher (x, y) -- message simple

3.3.1 : Afficher (x, y) -- message imbriqué

4.2 : âge := Soustraire (Aujourd'hui,
DateDeNaissance)
-- message imbriqué avec valeur retournée

[Age >= 18 ans] 6.2 : Voter () –- message
conditionnel

4.a, b.6 / c.1 : Allumer (Lampe) –-
synchronisation avec d'autres flots d'exécution

1 * : Laver () –- itération

3.a, 3.b / 4 *|||[i := 1..n] : Eteindre () –-
itération parallèle
```

Les arguments des messages sont représentés dans les diagrammes, soit en utilisant du pseudo-code, soit directement dans la syntaxe du langage de programmation. La notation propose également une représentation graphique des arguments sous la forme de flèches terminées par de petits cercles. Le diagramme suivant donne un exemple de représentation graphique des arguments d'un message.

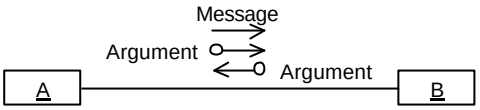


Figure 222 : Représentation graphique des arguments d'un message au moyen de flèches terminées par de petits cercles.

Introduction au métamodèle

Les collaborations

Une collaboration est un mécanisme composé d'éléments structurels et comportementaux. Les collaborations fournissent un mécanisme d'organisation, mais possèdent, contrairement aux paquetages, une identité et une portée sémantique. Un même élément peut intervenir dans plusieurs collaborations.

Une collaboration englobe deux genres de construction : un contexte composé d'une description de la structure statique des objets concernés et une interaction représentée par une séquence de messages échangés par les dits objets. Les deux aspects sont requis pour pleinement documenter le comportement, mais chaque aspect peut être visualisé indépendamment.

Les collaborations s'emploient, selon leur niveau de détail, pour décrire des spécifications et pour exprimer des réalisations. Le tableau suivant résume les éléments de modélisation qui peuvent être décrits par une collaboration.

Spécification	Type	Opération	Cas d'utilisation
Réalisation	Classe	Méthode	Réalisation de cas d'utilisation

Figure 223 : Tableau récapitulatif des éléments de modélisation qui peuvent être décrits par des collaborations.

Les collaborations existent également sous une forme générique (modèle), paramétrée par des classes, des relations, des attributs et des opérations. Une

collaboration générique est appelée *pattern*, ou schème[15] et micro-architecture en français. Les patterns possèdent toujours un nom, contrairement aux collaborations qui peuvent rester anonymes.

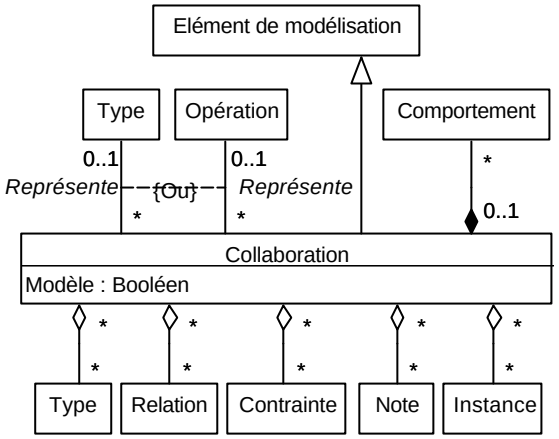


Figure 224 : Extrait du métamodèle. Représentation des collaborations.

Les interactions

Une interaction exprime le comportement qui résulte de la collaboration d'un groupe d'instances. Une interaction peut être visualisée selon le point de vue du temps (par les diagrammes de séquence) ou selon le point de vue de l'espace (par les diagrammes de collaboration). Les interactions comprennent principalement les éléments suivants :

- les *instances* qui sont la manifestation concrète d'un type,
- les *liens* qui relient les instances et servent de support pour les envois de message,
- les *messages* qui déclenchent les opérations,
- les *rôles* joués par les extrémités des liens.

[15] Ma préférence va à cette traduction, mais à l'heure actuelle tout le monde parle de pattern, j'en ferai donc autant !

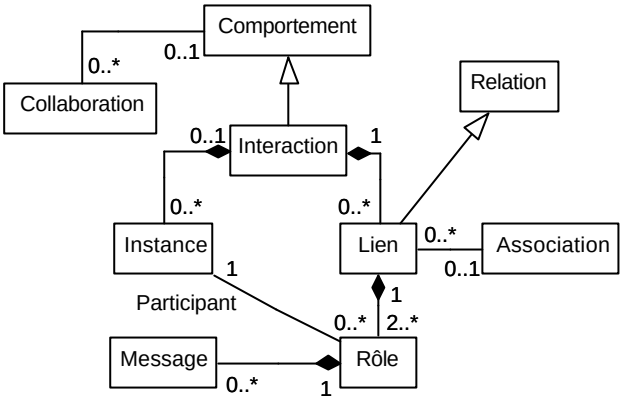


Figure 225 : Extrait du métamodèle. Représentation des interactions.

Les diagrammes de séquence

Les diagrammes de séquence montrent des interactions entre objets selon un point de vue temporel. Le contexte des objets n'est pas représenté de manière explicite comme dans les diagrammes de collaboration. La représentation se concentre sur l'expression des interactions.

Représentation des interactions

Un diagramme de séquence représente une interaction entre objets en insistant sur la chronologie des envois de messages. La notation est dérivée des *Object Message Sequence Chart*[16] du *Siemens Pattern Group*. Un objet est matérialisé par un rectangle et une barre verticale appelée ligne de vie des objets.

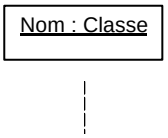


Figure 226 : Représentation graphique d'un objet.

Les objets communiquent en échangeant des messages représentés au moyen de flèches horizontales, orientées de l'émetteur du message vers le destinataire. L'ordre d'envoi des messages est donné par la position sur l'axe vertical. L'axe

[16] Wiley 1996, *Pattern-Oriented Software Architecture : A System of Patterns.* ISBN 0471958697.

vertical peut être gradué afin d'exprimer précisément les contraintes temporelles, dans le cas par exemple de la modélisation d'un système temps réel.

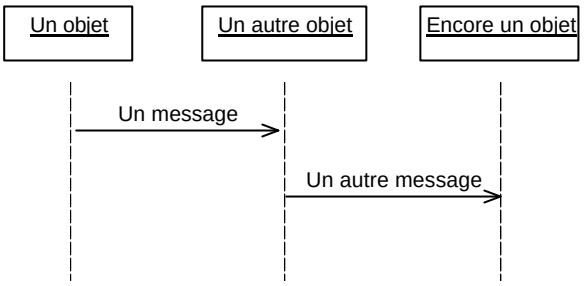


Figure 227 : Exemple de diagramme de séquence.

En modélisation objet, les diagrammes de séquence s'utilisent de deux manières bien différentes, selon la phase du cycle de vie et le niveau de détail désiré.

La première utilisation correspond à la documentation des cas d'utilisation ; elle se concentre sur la description de l'interaction, souvent dans des termes proches de l'utilisateur et sans entrer dans les détails de la synchronisation. L'indication portée sur les flèches correspond alors à des événements qui surviennent dans le domaine de l'application. A ce stade de la modélisation, les flèches ne correspondent pas encore à des envois de messages au sens des langages de programmation, et la distinction entre flots de contrôle et flots de données n'est généralement pas opérée. La figure suivante représente le début d'une communication téléphonique.

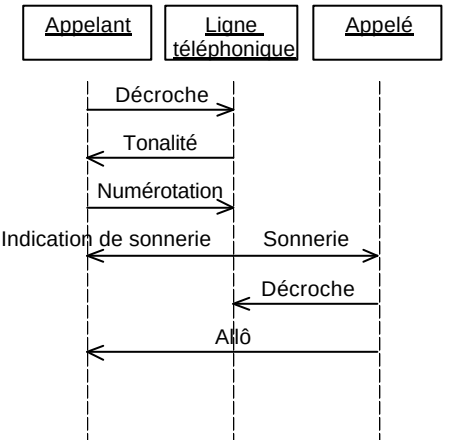


Figure 228 : Exemple d'utilisation d'un diagramme de séquence pour la représentation d'événements qui surviennent dans le domaine.

La deuxième utilisation correspond à un usage plus informatique et permet la représentation précise des interactions entre objets. Le concept de message unifie toutes les formes de communication entre objets, en particulier l'appel de procédure, l'événement discret, le signal entre flots d'exécution ou encore l'interruption matérielle.

Les diagrammes de séquence distinguent deux grandes catégories d'envois de message :

- les envois synchrones pour lesquels l'émetteur est bloqué et attend que l'appelé ait fini de traiter le message,
- les envois asynchrones pour lesquels l'émetteur n'est pas bloqué et peut continuer son exécution.

Un envoi synchrone se représente par une flèche qui part de l'émetteur du message et pointe vers son destinataire. Un envoi asynchrone se représente par une demi-flèche.

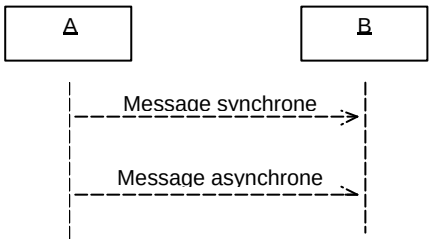


Figure 229 : Représentation graphique des envois de message synchrones et asynchrones.

La flèche qui symbolise un message peut être représentée en oblique pour matérialiser les délais de transmission non négligeables par rapport à la dynamique générale de l'application.

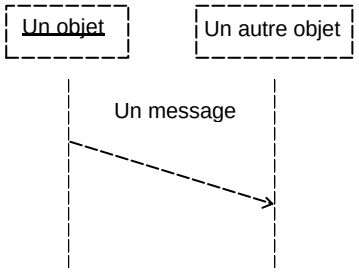


Figure 230 : Représentation d'un délai de propagation.

Un objet peut également s'envoyer un message. Cette situation se représente par une flèche qui boucle sur la ligne de vie de l'objet.

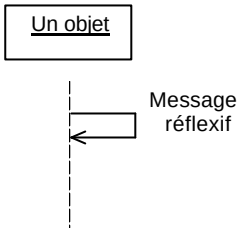


Figure 231 : Exemple d'envoi de message réflexif.

Cette construction ne correspond pas toujours à un vrai message ; elle peut indiquer un point d'entrée dans une activité de plus bas niveau, qui s'exerce au sein de l'objet. Les interactions internes (entre objets contenus par un objet composite) représentées par un message réflexif peuvent aussi être décrites dans un diagramme de séquence.

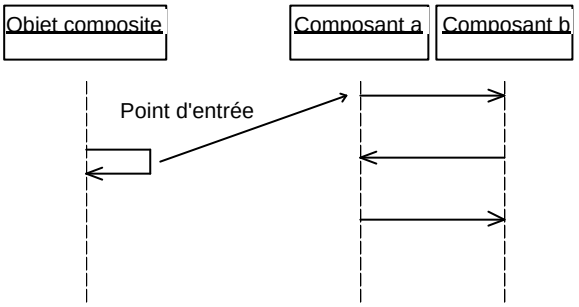


Figure 232 : Utilisation d'un message réflexif comme point d'entrée d'une interaction interne.

La création des objets se représente en faisant pointer le message de création sur le rectangle qui symbolise l'objet créé. La destruction est indiquée par la fin de la ligne de vie et par une lettre **X**, soit à la hauteur du message qui cause la destruction, soit après le dernier message envoyé par un objet qui se suicide.

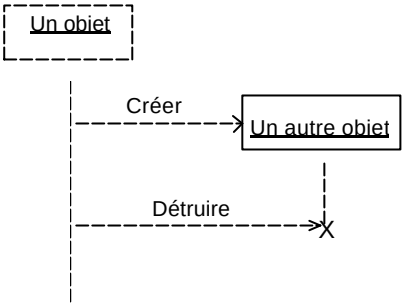


Figure 233 : Représentation de la création et de la destruction des objets.

Les diagrammes de séquence permettent également de représenter les périodes d'activité des objets. Une période d'activité correspond au temps pendant lequel un objet effectue une action, soit directement, soit par l'intermédiaire d'un autre objet qui lui sert de sous-traitant. Les périodes d'activité se représentent par des bandes rectangulaires placées sur les lignes de vies. Le début et la fin d'une bande correspondent respectivement au début et à la fin d'une période d'activité.

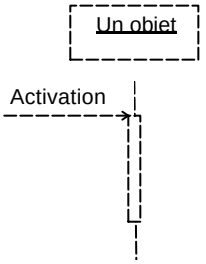


Figure 234 : Représentation de la période d'activité d'un objet au moyen d'une bande rectangulaire superposée à la ligne de vie de l'objet.

Le diagramme suivant montre le cas d'un objet **A** qui active un autre objet **B**. La période d'activité de l'objet **A** recouvre la période d'activité de l'objet **B**. Dans le cas d'un appel de procédure, le flot d'exécution est passé par l'objet **A** à l'objet **B**. L'objet **A** est alors bloqué jusqu'à ce que l'objet **B** lui redonne la main.

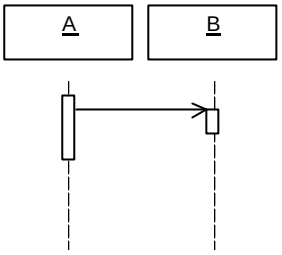


Figure 235 : Exemple d'objet qui active un autre objet.

Dans le cas d'un appel de procédure, et plus généralement dans le cas des envois synchrones, le retour en fin d'exécution de l'opération est implicite : il n'est pas nécessaire de le représenter dans les diagrammes. L'objet **A** reprend son exécution lorsque l'action déclenchée dans l'objet **B** est terminée.

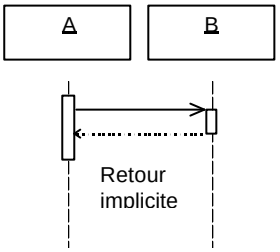


Figure 236 : Dans le cas des envois synchrones, le retour est implicite en fin d'activité et ne nécessite pas de représentation particulière.

En revanche, dans le cas des envois asynchrones, le retour doit être matérialisé lorsqu'il existe. Le diagramme suivant montre un objet **B** initialement activé par un objet **A**, qui retourne un message à l'objet **A** avant de cesser son exécution. Il faut noter que la fin de l'activation d'un objet ne correspond pas à la fin de sa vie. Un même objet peut être activé de nombreuses fois au cours de son existence.

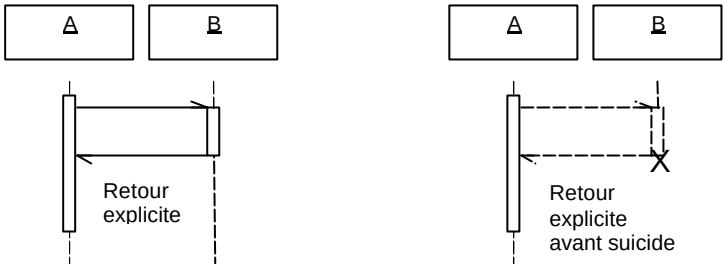


Figure 237 : Représentation explicite du retour dans le cas des envois asynchrones. La lettre ***X*** *symbolise une fin d'activité qui correspond également à une fin de vie.*

Le cas particulier des envois de messages récursifs se représente par un dédoublement de la bande rectangulaire. L'objet apparaît alors comme s'il était actif plusieurs fois.

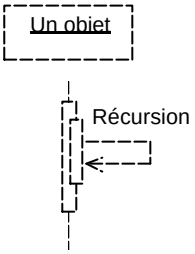


Figure 238 : Représentation de la récursion dans les diagrammes de séquence.

Structures de contrôle

Les formes des diagrammes de séquence reflètent indirectement les choix de structure. Les deux diagrammes suivants présentent respectivement un mode de contrôle centralisé et un mode décentralisé.

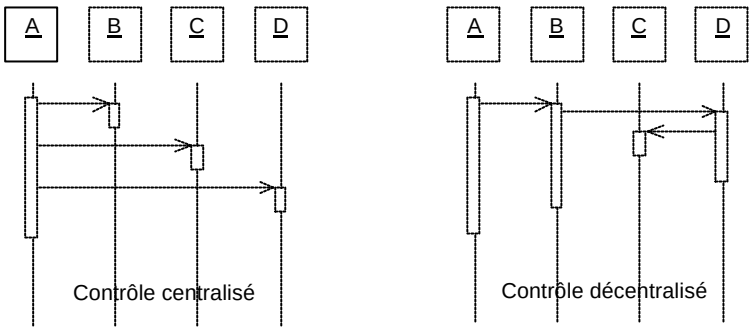


Figure 239 : La forme des diagrammes de séquence est le reflet du mode de contrôle de l'interaction.

Les diagrammes de séquence peuvent être complétés par des indications textuelles, exprimées sous la forme de texte libre ou de pseudo-code. L'instant d'émission d'un message, appelé transition, peut être nommé dans le diagramme à proximité du point de départ de la flèche qui symbolise le message. Ce nom sert alors de référence, par exemple pour construire des contraintes temporelles, comme dans le diagramme suivant. Lorsque la propagation d'un message prend un temps significatif par rapport à la dynamique du système, les instants d'émission et de réception des messages sont matérialisés par un couple **`(nom, nom prime)`**.

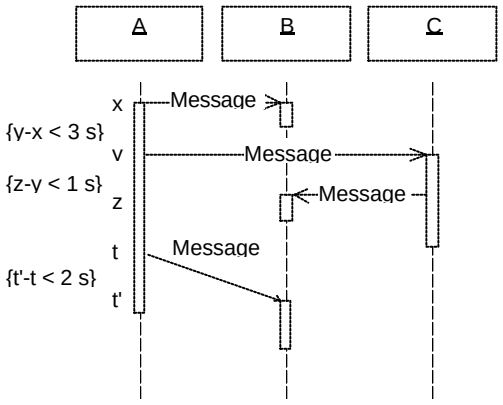


Figure 240 : Exemples de contraintes temporelles construites à partir de noms de transitions.

L'ajout de pseudo-code sur la partie gauche du diagramme permet la représentation des boucles et des branchements, de sorte que les diagrammes de séquence peuvent représenter la forme générale d'une interaction, au-delà de la seule prise en compte d'un scénario particulier.

Le diagramme suivant représente une boucle **`While`**. L'objet **A** envoie sans discontinuer un message à **B** tant que la condition **X** est vraie.

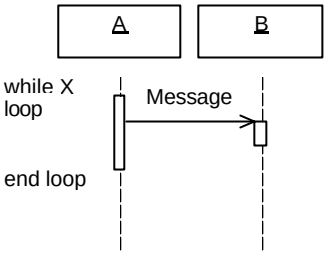


Figure 241 : Exemple de représentation d'une boucle ***`While`*** *au moyen de pseudo-code.*

La boucle **`while`** peut également être représentée au moyen d'une condition d'itération placée directement sur le message. L'itération est alors symbolisée par le caractère *****, placé devant la condition entre crochets.

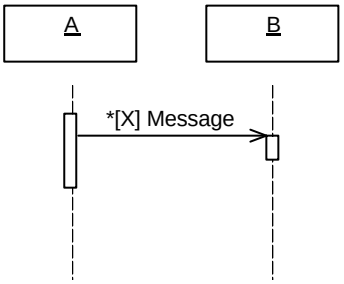


Figure 242 : Représentation alternative de la boucle ***`While`*** *au moyen d'une condition placée devant le message.*

Comme pour les boucles, les branchements conditionnels peuvent être matérialisés au moyen de pseudo-code placé sur la gauche du diagramme. Le diagramme suivant montre que l'objet **A** envoie un message à l'objet **B** ou à l'objet **C** selon la condition **X**.

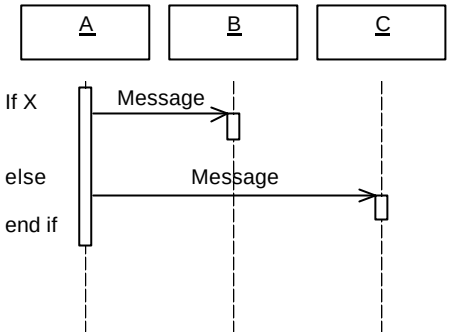


Figure 243 : Représentation des branchements conditionnels par l'ajout de pseudo-code.

Comme précédemment, des conditions placées devant les messages peuvent se substituer au pseudo-code. Les différentes branches sont alors matérialisées par plusieurs flèches qui prennent leur origine au même instant et se distinguent par les conditions placées devant les messages. A chaque branchement, les conditions doivent être mutuellement exclusives, et tous les cas doivent être couverts.

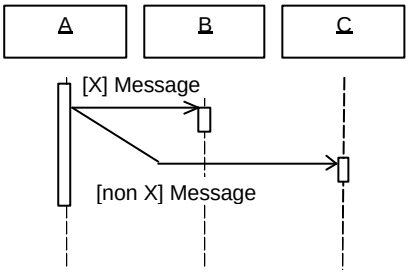


Figure 244 : Représentation graphique des branchements conditionnels.

Les alternatives, du côté du destinataire du message, sont représentées en dédoublant la ligne de vie de l'objet destinataire. La distinction entre les branches est indiquée par une condition placée cette fois-ci derrière le message, à proximité du point d'entrée sur la ligne de vie de l'objet destinataire.

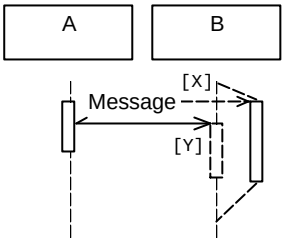


Figure 245 : Représentation graphique des branchements conditionnels.

Le pseudo-code permet également de mettre en correspondance l'interaction initiale, décrite par l'utilisateur lors de l'étude des cas d'utilisation, avec une interaction entre objets du domaine, construite par l'analyste.

Les diagrammes d'états-transitions

Les diagrammes d'états-transitions visualisent des automates d'états finis, du point de vue des états et des transitions.

Les automates

Le comportement des objets d'une classe peut être décrit de manière formelle en termes d'états et d'événements, au moyen d'un automate relié à la classe considérée.

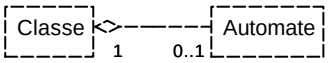


Figure 246 : Un automate peut être associé à chaque classe du modèle.

Les objets qui ne présentent pas un comportement réactif très marqué peuvent être considérés comme des objets qui restent toujours dans le même état : leurs classes ne possèdent alors pas d'automate.

Le formalisme retenu par UML pour représenter les automates s'inspire des *Statecharts*[17]. Il s'agit d'automates hiérarchiques, possédant les concepts d'orthogonalité, d'agrégation et de généralisation (ces notions sont définies plus loin dans le chapitre).

[17] Harel, D. 1987. *Statecharts : A Visual Formalism for Complex Systems*. Science of Computer Programming vol. 8.

Un automate est une abstraction des comportements possibles, à l'image des diagrammes de classes qui sont des abstractions de la structure statique. Chaque objet suit globalement le comportement décrit dans l'automate associé à sa classe et se trouve à un moment donné dans un état qui caractérise ses conditions dynamiques.

Les automates et les scénarios sont complémentaires. Les scénarios se représentent par une collaboration entre objets. La forme de l'interaction entre les objets qui collaborent au sein d'un scénario est déterminée par les états respectifs des différents objets. Les automates peuvent être utilisés pour décrire le comportement de groupes d'objets, en associant un automate à un composite, voire à un cas d'utilisation.

Un feu tricolore passe successivement de l'état vert, à l'état orange puis à l'état rouge, avant de repasser à l'état vert, et ainsi de suite durant toute sa période de fonctionnement.

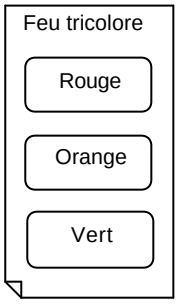


Figure 247 : De manière générale, tous les feux tricolores sont soit à l'état rouge, soit à l'état orange, soit à l'état vert.

Il est bien évident qu'il faut synchroniser l'état des feux qui se trouvent placés autour du même carrefour. Cette synchronisation, qui dépend de l'état du carrefour, peut aussi être décrite dans un automate associé à la classe des carrefours.

Les états

Chaque objet est à un moment donné dans un état particulier. Les états se représentent sous la forme de rectangles arrondis ; un chaque état possède un nom qui l'identifie.

Figure 248 : Les états se représentent au moyen de rectangles arrondis. Chaque état possède un nom qui doit être unique dans une portée lexicale donnée.

Les états se caractérisent par la notion de durée et de stabilité. Un objet est toujours dans un état donné pour un certain temps et un objet ne peut être dans un état inconnu ou non défini. Un état est toujours l'image de la conjonction instantanée des valeurs contenues par les attributs de l'objet, et de la présence ou non de liens, de l'objet considéré vers d'autres objets.

Le diagramme de classes suivant représente des personnes qui travaillent pour des sociétés.

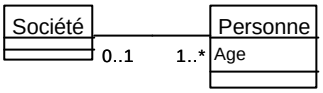


Figure 249 : Les personnes travaillent pour des sociétés.

Les personnes ne possèdent pas toutes un emploi et se trouvent, à un moment donné, dans un des états suivants : en activité, au chômage ou à la retraite. La figure suivante visualise les trois états d'une personne.

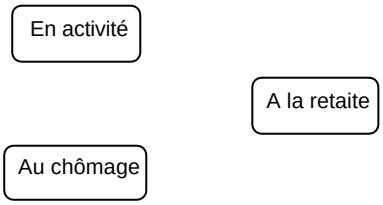


Figure 250 : Une personne peut être soit en activité, soit au chômage, soit à la retraite.

Pour connaître la situation d'une personne en particulier, il faut étudier la conjonction suivante :

- âge de la personne,
- présence d'un lien vers une société.

Dans le diagramme suivant, il n'y a pas de lien entre Toto, âgé de 30 ans, et une société : Toto est donc au chômage. Titi, quant à lui, possède un lien vers une société et est âgé de 40 ans : Titi est donc en activité. Ernest, enfin, ne possède pas de lien vers une société et est âgé de 75 ans : Ernest est donc à la retraite.

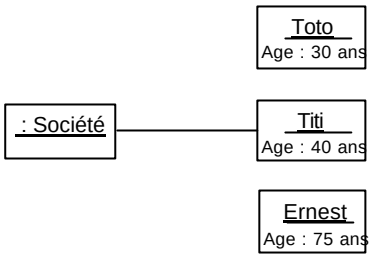


Figure 251 : L'état des trois personnes est différent. Toto est au chômage, Titi en activité et Ernest à la retraite.

Les automates retenus par UML sont déterministes. Un diagramme d'états-transitions ne doit pas laisser de place aux constructions ambiguës. Cela signifie en particulier qu'il faut toujours décrire l'état initial du système. Pour un niveau hiérarchique donné, il y a toujours un et un seul état initial. En revanche, il est possible d'avoir plusieurs états finaux qui correspondent chacun à une condition de fin différente. Il est également possible de n'avoir aucun état final, dans le cas par exemple d'un système qui ne s'arrête jamais.

L'état initial se représente par un gros point noir. Un état final se représente par un gros point noir encerclé.

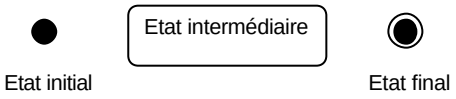


Figure 252 : L'état initial est représenté par un point noir. Un état final est représenté par un point noir encerclé.

Les transitions

Lorsque les conditions dynamiques évoluent, les objets changent d'état en suivant les règles décrites dans l'automate associé à leurs classes. Les diagrammes d'états-transitions sont des graphes dirigés. Les états sont reliés par des connexions unidirectionnelles, appelées transitions. Le passage d'un état à l'autre s'effectue lorsqu'une transition est déclenchée par un événement qui survient dans le domaine du problème. Le passage d'un état à un autre est instantané car le système doit toujours être dans un état connu.

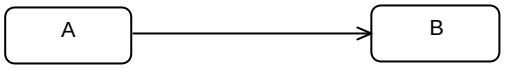


Figure 253 : Une transition permet de passer d'un état à un autre état ; elle se représente au moyen d'une flèche qui pointe de l'état de départ vers l'état d'arrivée.

Les transitions ne relient pas nécessairement des états distincts. L'exemple suivant décrit un fragment d'analyseur lexical. La reconnaissance des unités lexicales est effectuée dans un état de lecture. L'automate reste dans cet état tant que les caractères lus ne sont pas des séparateurs.

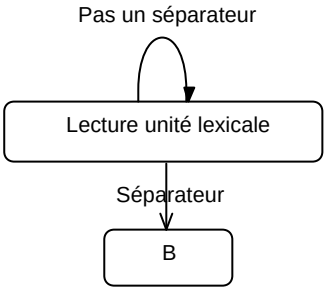


Figure 254 : Exemple de transition d'un état vers lui-même dans un fragment d'automate lexical.

Les événements

Un événement correspond à l'occurrence d'une situation donnée dans le domaine du problème. Contrairement aux états qui durent, un événement est par nature une information instantanée qui doit être traitée sans plus attendre. Un événement sert de déclencheur pour passer d'un état à un autre. Les transitions indiquent les chemins dans le graphe des états. Les événements déterminent quels chemins doivent être suivis. Les événements, les transitions et les états sont indissociables dans la description du comportement dynamique. Un objet, placé dans un état donné, attend l'occurrence d'un événement pour passer dans un autre état. De ce point de vue, les objets se comportent comme des éléments passifs, contrôlés par les événements en provenance du système.

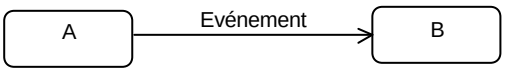


Figure 255 : Un événement déclenche la transition qui lui est associée.

La syntaxe générale d'un événement suit la forme suivante :

```
Nom_De_L_Evénement (Nom_De_Paramètre : Type, ...)
```

La spécification complète d'un événement comprend :

- le nom de l'événement,
- la liste des paramètres,
- l'objet expéditeur,
- l'objet destinataire,
- la description de la signification de l'événement.

Dans l'exemple suivant, chaque transition porte l'événement qui la déclenche.

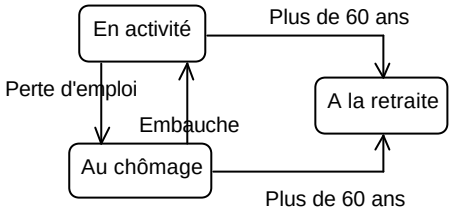


Figure 256 : Exemple d'automate.

La communication par événement est de type asynchrone, atomique, et unidirectionnelle. Un objet peut envoyer un événement à un autre objet qui doit toujours être à même de l'interpréter.

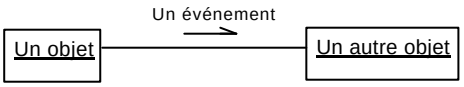


Figure 257 : Un objet peut envoyer un événement asynchrone à un autre objet.

Les besoins de communication par événements synchrones ou les échanges bidirectionnels peuvent se représenter au moyen de deux échanges asynchrones de direction opposée.

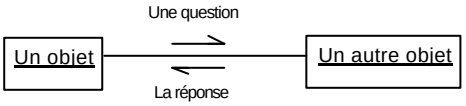


Figure 258 : Les besoins de communication par événements synchrones peuvent se réaliser au moyen de deux événements asynchrones, de direction opposée.

Dans ce cas, il appartient à l'objet émetteur de la requête de se mettre en attente de la réponse. Ceci implique que l'automate d'états finis qui le décrit possède une séquence du type suivant :

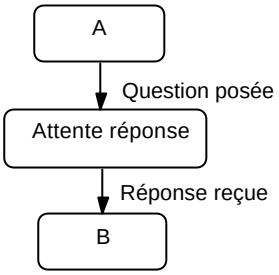


Figure 259 : Extrait de l'automate d'un objet qui communique de manière synchrone à partir d'événements asynchrones. L'émetteur de la requête doit se mettre en attente de la réponse.

Les gardes

Une garde est une condition booléenne qui valide ou non le déclenchement d'une transition lors de l'occurrence d'un événement.

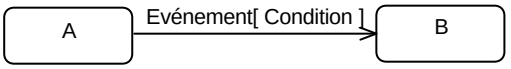


Figure 260 : Représentation des gardes.

Les gardes permettent de maintenir l'aspect déterministe d'un automate d'états finis, même lorsque plusieurs transitions peuvent être déclenchées par le même événement. Lorsque l'événement a lieu, les gardes – qui doivent être mutuellement exclusives – sont évaluées et une transition est validée puis déclenchée. Dans l'exemple suivant, lorsqu'il fait trop chaud, les gardes permettent selon la saison de déclencher un climatiseur ou d'ouvrir tout simplement les fenêtres.

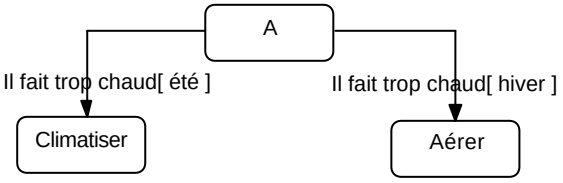


Figure 261 : L'événement ***`il fait trop chaud`*** *entraîne la climatisation ou l'ouverture des fenêtres selon la saison.*

Les opérations, les actions et les activités

Le lien entre les opérations définies dans la spécification de classe et les événements apparaissant dans les diagrammes d'états-transitions est effectué par l'intermédiaire des actions et des activités.

Chaque transition peut être décorée par le nom d'une action à exécuter lorsque la transaction est déclenchée par un événement. Pour respecter la sémantique générale de la transition, l'action est considérée comme instantanée et atomique.

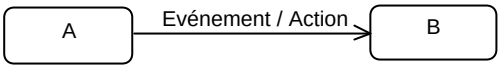


Figure 262 : Lorsqu'une transition est déclenchée, l'action qui lui est attachée est exécutée instantanément.

L'action correspond à une des opérations déclarées dans la classe de l'objet destinataire de l'événement. L'action a accès aux paramètres de l'événement, ainsi qu'aux attributs de l'objet. En réalité, toute opération prend un certain temps à s'exécuter ; la notion d'action instantanée doit s'interpréter comme une opération dont le temps d'exécution est négligeable devant la dynamique du système.

Les états peuvent également contenir des actions ; elles sont exécutées à l'entrée ou à la sortie de l'état ou lors de l'occurrence d'un événement pendant que l'objet est dans l'état.

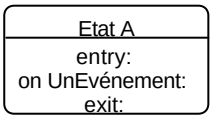


Figure 263 : Une action peut être exécutée à l'entrée ou à la sortie d'un état, ou en cas d'événement survenant dans l'état.

L'action d'entrée (symbolisée par le mot-clé **`entry:`**) est exécutée de manière instantanée et atomique dès l'entrée dans l'état. De même, l'action de sortie (symbolisée par **`exit:`**) est exécutée à la sortie de l'état. L'action sur événement interne (symbolisée par **`on:`**) est exécutée lors de l'occurrence d'un événement qui ne conduit pas à un autre état. Un événement interne n'entraîne pas l'exécution des actions d'entrée et de sortie, contrairement au déclenchement d'une transition réflexive.

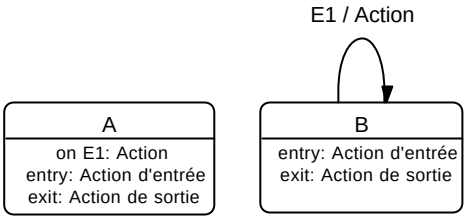


Figure 264 : Un événement interne n'entraîne pas l'exécution des actions de sortie et d'entrée, contrairement au déclenchement d'une transition réflexive.

Les actions correspondent à des opérations dont le temps d'exécution est négligeable. Une opération qui prend du temps correspond à un état plutôt qu'à une action. Le mot-clé **`do:`** indique une activité, c'est-à-dire une opération qui prend un temps non négligeable et qui est exécutée pendant que l'objet est dans un état donné.

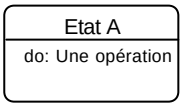


Figure 265 : Les opérations qui durent sont forcément exécutées au sein d'un état. Elles sont identifiées par le mot-clé ***`do:`*** *suivi du nom de l'opération.*

Contrairement aux actions, les activités peuvent être interrompues à tout moment, dès qu'une transition de sortie de l'état est déclenchée. Certaines activités sont cycliques et ne s'arrêtent que lorsqu'une transition de sortie est déclenchée. D'autres activités sont séquentielles et démarrent à l'entrée dans l'état (tout de suite après l'exécution des actions d'entrée).

Lorsqu'une activité séquentielle parvient à son terme, l'état peut être quitté si une des transitions est franchissable. Ce type de transition qui n'est pas déclenchée par un événement est appelée transition automatique.

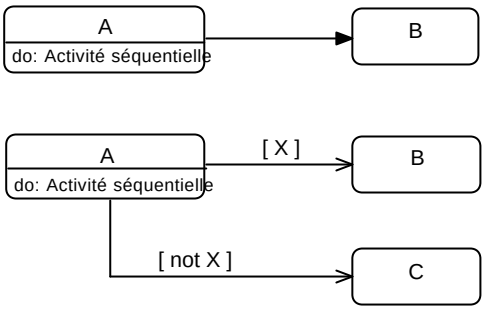


Figure 266 : Lorsque l'activité se termine, les transitions automatiques – sans événements, mais éventuellement protégées par des gardes – sont déclenchées.

Les états peuvent également contenir des variables exprimées sous la forme d'attributs. Les variables d'état appartiennent à la classe associée à l'automate, mais peuvent être représentées dans les diagrammes d'états-transitions lorsqu'elles sont manipulées par les actions ou les activités.

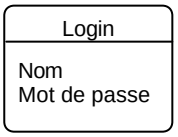


Figure 267 : Exemple de représentation de variables d'état.

Points d'exécution des opérations

En résumé, il existe six points pour spécifier les opérations qui doivent être exécutées. Ces points sont, dans l'ordre d'exécution :

- l'action associée à la transition d'entrée (**`Op1`**),
- l'action d'entrée de l'état (**`Op2`**),
- l'activité dans l'état (**`Op3`**),
- l'action de sortie de l'état (**`Op4`**),
- l'action associée aux événements internes (**`Op5`**),
- l'action associée à la transition de sortie de l'état (**`Op6`**).

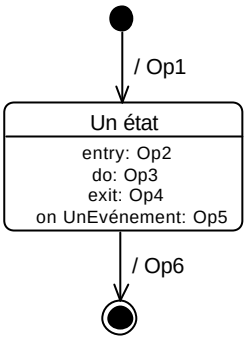


Figure 268 : Le formalisme des diagrammes d'états-transitions offre six points pour spécifier les opérations qui doivent être exécutées.

Généralisation d'états

Les diagrammes d'états-transitions peuvent devenir assez difficiles à lire lorsque, du fait de l'explosion combinatoire, le nombre de connexions entre états devient élevé : le diagramme se met alors à ressembler à un plat de spaghettis.

La solution pour maîtriser cette situation consiste à utiliser le principe de généralisation d'états. Les états plus généraux sont appelés super-états, les états plus spécifiques sont appelés sous-états. Cette approche d'abstraction procède de la même démarche que la généralisation et la spécialisation des classes ; elle facilite la représentation et permet d'occulter les détails.

Un état peut être décomposé en plusieurs sous-états disjoints ; les sous-états héritent des caractéristiques de leur super-état, en particulier des variables d'état et des transitions externes. La décomposition en sous-états est également appelée décomposition disjonctive (décomposition de type ou-exclusif) car l'objet doit être dans un seul et un seul sous-état à la fois.

Les deux diagrammes suivants illustrent la simplification apportée par la généralisation d'états.

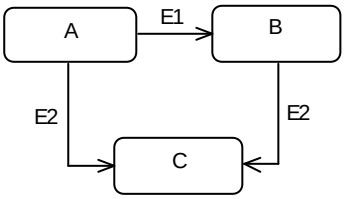


Figure 269 : Exemple d'automate dans lequel la transition ***E2*** *peut être factorisée.*

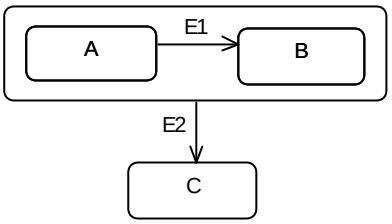


Figure 270 : Représentation hiérarchique de l'automate précédent.

Les transitions internes peuvent être héritées, sauf si la décomposition en sous-états a pour objet de définir un état particulier pour le traitement d'une transition interne. Les transitions d'entrée ne sont pas héritées par tous les états ; seul un état (le super-état ou un de ses sous-états) peut être cible de la transition. Dans l'exemple suivant, l'état **B** est divisé en deux sous-états **B1** et **B2**. La transition d'entrée dans l'état **B** doit être reportée sur un des sous-états, soit directement (comme dans le diagramme suivant), soit indirectement au moyen d'un état initial emboîté.

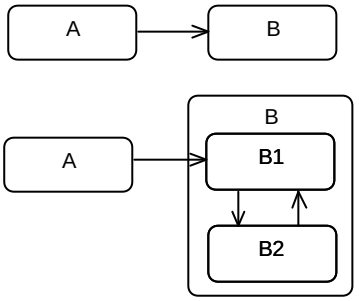


*Figure 271 : L'état **B** est divisé en deux sous-états **B1** et **B2**. La transition d'entrée dans l'état **B** doit être reportée directement sur un des sous états.*

Dans l'exemple précédent, l'état **A** est relié au sous-état **B1**. Cette situation manque d'abstraction et est comparable à un mécanisme écrit selon la spécification d'une super-classe, mais ayant besoin de connaître le détail des sous-classes. Il est toujours préférable de limiter les liens entre niveaux hiérarchiques d'un automate en définissant systématiquement un état initial pour chaque niveau. Le diagramme suivant limite les connaissances entre niveaux.

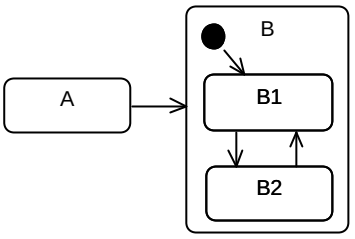


Figure 272 : Amélioration du niveau d'abstraction d'un automate par l'ajout d'un état initial dans un super-état.

La visualisation exhaustive des sous-états induit une forte charge d'information dans les diagrammes. Le détail des sous-états peut être masqué, pour donner par exemple une vision de plus haut niveau. Grâce à la notion de souche il est possible de montrer que les transitions entrantes dans un état composite concernent un sous-état particulier, sans pour autant entrer dans les détails de la représentation de ces sous-états.

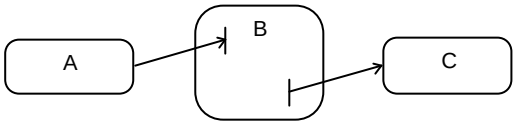


Figure 273 : Les souches réduisent la charge d'information, tout en matérialisant la présence des sous-états de ***B****.*

Agrégation d'états

L'agrégation d'états est la composition d'un état à partir de plusieurs autres états indépendants. La composition est de type conjonctive (composition de type *et*) ce qui implique que l'objet doit être simultanément dans tous les états composant l'agrégation d'états. La conjonction d'états représente une forme de parallélisme entre automates.

L'exemple suivant illustre différents aspects de la notion d'agrégation d'états. L'état **S** est un agrégat formé de deux états indépendants **T** et **U** ; **T** est composé des sous-états **X**, **Y** et **Z,** et **U** des sous-états **A** et **B**. Le domaine de **S** est le produit cartésien de **T** et **U**.

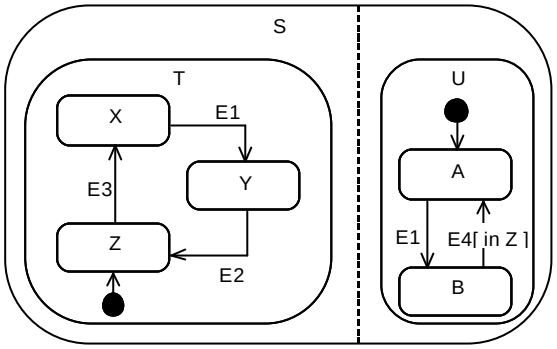


*Figure 274 : Exemple d'automate à agrégation d'états. L'état **S** appartient au produit cartésien des états **T** et **U**.*

Une transition entrante dans l'état **S** implique l'activation simultanée des automates **T** et **U**, c'est-à-dire que l'objet est placé dans l'état composite **(Z,A)**. Lorsque l'événement **E3** a lieu, les automates **T** et **U** peuvent évoluer indépendamment, ce qui amène l'objet de l'état composite **(Z,A)** vers l'état composite **(X,A)**. Les automates **T** et **U** peuvent évoluer simultanément, ce qui est le cas lorsque l'événement **E1** emmène l'objet de l'état composite **(X,A)** vers l'état composite **(Y,B)**.

L'ajout de conditions sur les transitions, comme la garde **[in Z]** placée sur la transition de **B** vers **A**, permet d'introduire des relations de dépendance entre composants de l'agrégat. Lorsque l'événement **E4** a lieu, la transition de **B** vers **A** est validée uniquement si l'objet est à ce moment-là également dans l'état **Z**.

L'agrégation d'états, tout comme la généralisation d'états, permet de simplifier la représentation des automates. La généralisation simplifie par factorisation, l'agrégation simplifie par segmentation de l'espace des états. Sans agrégation d'états, l'automate équivalent serait de la forme suivante :

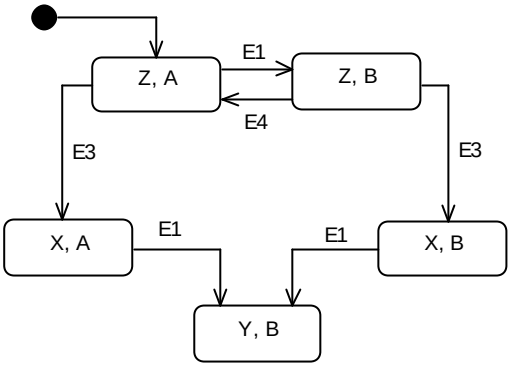


Figure 275 : Automate à plat équivalent à l'automate à agrégation précédent.

Le nombre d'états d'un tel automate à plat est au maximum égal au produit du nombre d'états de chaque automate composant. Dans le cas d'une agrégation de trois automates d'une centaine d'états chacun (ce qui est déjà beaucoup), l'automate à plat équivalent pourrait contenir jusqu'à un million d'états !

L'historique

Par défaut, un automate n'a pas de mémoire. La notation spéciale **H** offre un mécanisme pour mémoriser le dernier sous-état visité et pour le rejoindre lors d'une transition entrant dans le super-état qui l'englobe. Le concept d'historique s'applique au niveau dans lequel le symbole **H** est déclaré. Le symbole **H** peut être placé n'importe où dans l'état ; le coin bas gauche est la position par défaut.

Le diagramme suivant représente un état **C** qui mémorise le dernier sous-état actif. L'historique est initialisé lors du déclenchement de la transition issue de l'état initial.

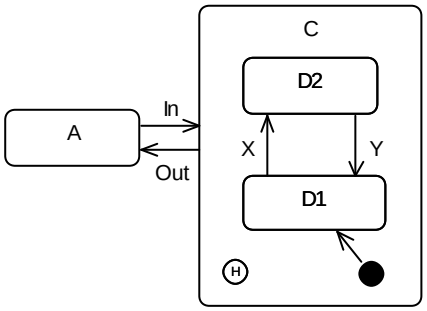


Figure 276 : Exemple d'automate hiérarchique. L'état ***C*** *mémorise le dernier sous-état actif.*

La mémorisation, quelle que soit la profondeur des sous-états emboîtés, est également possible ; elle est indiquée par le symbole **H*** . Les niveaux de mémorisation intermédiaires sont obtenus en plaçant un symbole **H** par niveau hiérarchique. Dans l'exemple suivant, l'état **A** mémorise le dernier sous-état actif, quelle que soit la profondeur d'emboîtement des sous-états.

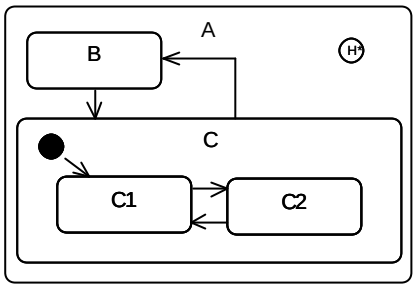


*Figure 277 : Le mécanisme d'historique **H*** permet de mémoriser le dernier sous-état actif, quelle que soit la profondeur d'emboîtement.*

L'exemple suivant montre l'utilisation de l'historique pour la réalisation d'un lave-vaisselle. Le cycle de lavage est découpé en trois grandes étapes : le rinçage, le lavage et le séchage. A tout moment, la porte peut être ouverte, pour rajouter par exemple une tasse ; dès que la porte est refermée, le cycle de lavage reprend au point où il a été interrompu.

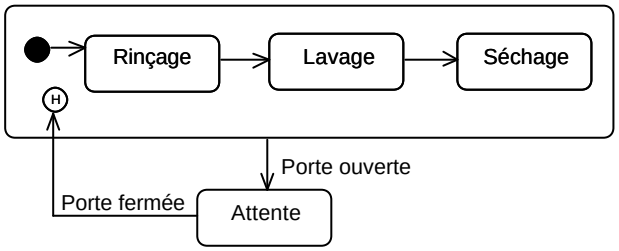


*Figure 278 : Un état englobant peut mémoriser le dernier sous-état actif. Le contrôle est transmis directement au sous-état mémorisé lorsqu'une transition qui arrive sur l'état spécial **H** est déclenchée.*

La communication entre objets

Les objets communiquent en échangeant des messages. A la réception d'un message, l'objet destinataire déclenche une opération pour traiter le message. Le message est un concept très général qui peut tout aussi bien représenter l'appel de procédure, l'interruption venant du matériel ou encore la liaison dynamique.

Le message est la manière de visualiser les échanges dans les diagrammes d'interaction entre objets, c'est-à-dire dans les diagrammes de séquence et de collaboration. Ces diagrammes d'interaction montrent des cas particuliers de comportement au sein d'un cas d'utilisation.

Les automates représentent des abstractions du comportement du point de vue d'un groupe d'objets (le plus souvent une classe). Le comportement décrit par les scénarios est une conséquence particulière de l'état de tous les objets qui collaborent au sein de ces scénarios.

Les envois de messages entre deux objets sont visualisés de manière abstraite dans le formalisme des diagrammes d'états-transitions par l'envoi d'un événement entre les automates d'états-finis des classes d'objets concernées. La visualisation dans les diagrammes d'états-transitions est plus abstraite car, à un événement envoyé entre deux automates, correspond de nombreux envoi de messages entre objets.

La syntaxe d'un envoi d'événement vers une classe est :

```
^Cible.Evénement (Arguments)
```

où la cible désigne la classe des objets destinataires de l'événement.

La syntaxe complète d'une transition est :

```
Evénement (Arguments)[Condition]
/Action
^Cible.Evénement (Arguments)
```

L'exemple suivant montre des fragments d'automates d'un téléviseur et de sa télécommande. Le téléviseur peut être allumé ou éteint par manipulation d'un interrupteur basculant. La télécommande possède un bouton-poussoir (touche on/off) qui, lorsqu'il est enfoncé, allume ou éteint le téléviseur. Le téléviseur peut être commandé directement ou par la télécommande.

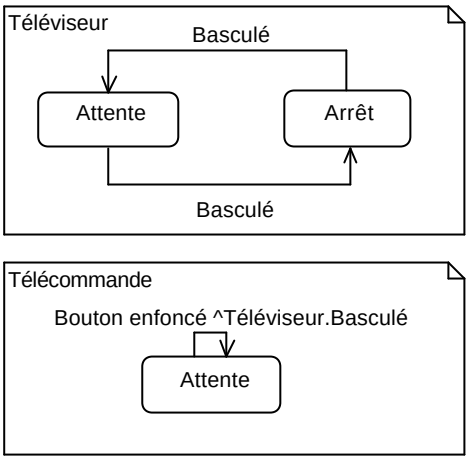


Figure 279 : Le mécanisme d'envoi d'événement entre automates représente de manière abstraite les échanges de messages entre les objets instances des classes associées à ces automates.

Par généralisation, l'envoi d'événement est possible vers n'importe quel ensemble d'objets (une classe est un cas particulier d'ensemble d'objets). Les formes les plus courantes sont l'envoi à tous les objets (la diffusion) et l'envoi à un objet particulier (le point à point). Dans le cas d'un automate qui décrit une classe composite, les actions peuvent faire référence directement aux opérations déclarées dans les différentes classes contenues par la classe composite.

Création et destruction des objets

La création d'un objet se représente par l'envoi d'un événement de création à la classe de l'objet. Les paramètres de cet événement permettent d'initialiser le nouvel objet, qui débute son existence à partir de l'état initial décrit dans l'automate de la classe. L'exemple suivant montre une transition de création qui immatricule un avion. En cas de crash, l'avion cesse d'exister.

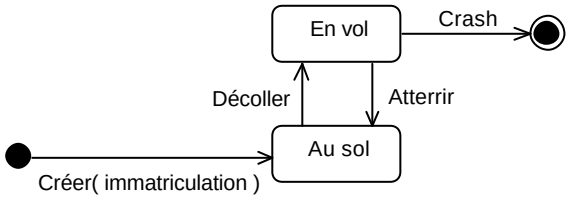


Figure 280 : La transition de création emmène l'objet depuis l'état initial vers son premier état de fonctionnement. L'arrivée dans l'état final implique la disparition de l'objet.

La destruction d'un objet est effective lorsque le flot de contrôle de l'automate atteint un état final non emboîté. L'arrivée dans un état final emboîté implique la remontée à l'état englobant, et non la fin de l'objet.

Les transitions temporisées

Les attentes sont par définition des activités qui durent un certain temps. Une attente est donc naturellement rattachée à un état plutôt qu'à une transition ; elle se représente au moyen d'une activité d'attente. L'activité d'attente est interrompue lorsque l'événement attendu a lieu. Cet événement déclenche en effet une transition permettant de sortir de l'état qui héberge l'activité d'attente. Le flot de contrôle est alors transmis à un autre état.

L'exemple suivant illustre une séquence d'attente dans un guichet automatique de banque. La trappe qui accueille les dépôts est ouverte ; le système informe l'utilisateur qu'il dispose de trois minutes pour effectuer son dépôt. Si le dépôt est effectué dans les trois minutes, l'activité d'attente est interrompue par le déclenchement de la transition vers l'état **B**. En revanche, si le dépôt n'intervient pas dans le délai imparti, la transition automatique vers l'état d'annulation est déclenchée à la fin de l'activité d'attente. Dans tous les cas, l'action de sortie de l'état d'attente referme la trappe.

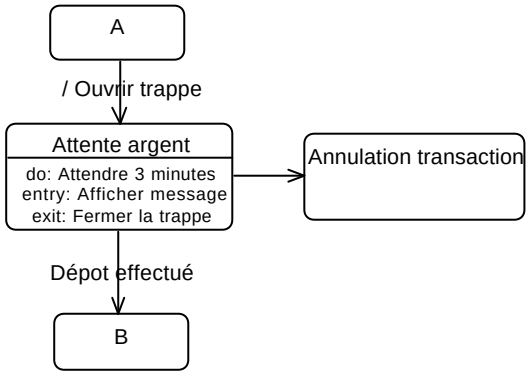


Figure 281 : Représentation d'une temporisation par combinaison d'une activité d'attente et d'une transition automatique.

Les temporisations peuvent se représenter au moyen d'une notation plus compacte, directement attachée à la transition déclenchée après le délai d'attente. L'événement déclencheur porte le nom générique de **`temporisation`** et le paramètre spécifie la durée de la temporisation.

La syntaxe d'un événement de temporisation est :

```
Temporisation(durée_de_temporisation)
```

Le diagramme précédent se transforme de la manière suivante :

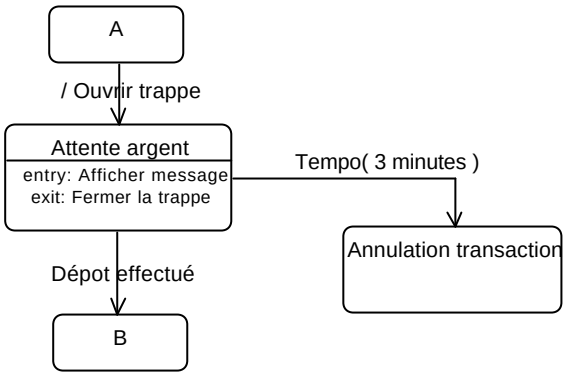


Figure 282 : Représentation d'une temporisation par une transition temporisée.

Les automates offrent un formalisme bien adapté pour la représentation des comportements complexes. En analyse, les diagrammes d'états-transitions capturent le comportement souhaité ; durant la réalisation, les automates peuvent s'écrire facilement, par exemple au moyen de tables contenant les états et les actions à exécuter lors des transitions.

Introduction au métamodèle

Un automate représente un comportement qui résulte d'opérations exécutées suite à une séquence de changements d'état. Un automate peut être visualisé selon le point de vue des états (par les diagrammes d'états-transitions) ou selon le point de vue des actions (par les diagrammes d'activités). Un automate spécifie le comportement d'une collaboration.

L'exécution d'une instance d'automate ne peut pas être interrompue. A tout moment, un automate peut réagir à un événement particulier qui l'entraîne d'un état stable vers un autre état stable. L'exécution de l'automate débute à partir du pseudo-état initial et se poursuit, au rythme des événements, jusqu'à ce qu'un pseudo-état final, non emboîté, soit atteint.

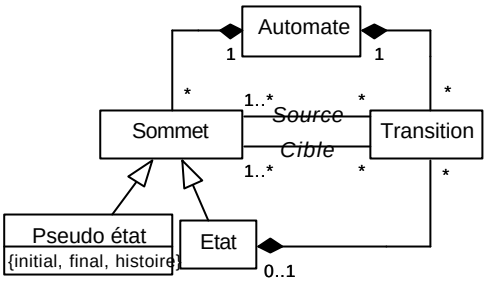


Figure 283 : Extrait du métamodèle. Un automate est un graphe composé d'états et de transitions.

Les événements déclenchent des transitions. Le déclenchement d'une transition entraîne l'automate de l'état source de la transition vers l'état de destination. Au passage, une ou plusieurs actions peuvent être déclenchées sur un ou plusieurs objets.

UML définit trois sortes d'événements :

- l'événement *signal* causé par un signal,
- l'événement *appel* causé par une opération,
- l'événement *temporel* causé par l'expiration d'une temporisation.

Le diagramme suivant représente les transitions, leurs actions et les différents événements qui les déclenchent.

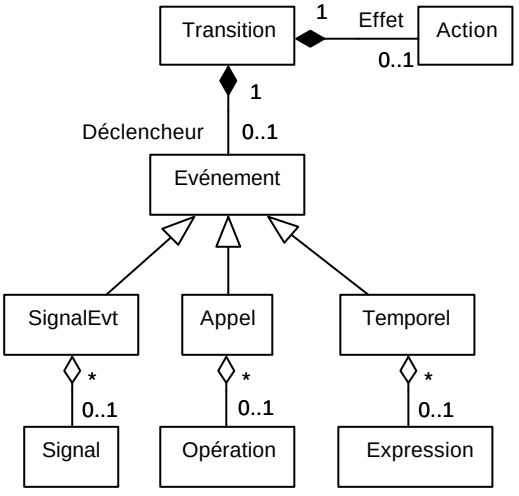


Figure 284 : Extrait du métamodèle. Représentation des différentes sortes d'événements.

Les diagrammes d'activités

Un diagramme d'activités est une variante des diagrammes d'états-transitions, organisé par rapport aux actions et principalement destiné à représenter le comportement interne d'une méthode (la réalisation d'une opération) ou d'un cas d'utilisation.

Représentation des activités

Un diagramme d'activités représente l'état de l'exécution d'un mécanisme, sous la forme d'un déroulement d'étapes regroupées séquentiellement dans des branches parallèles de flot de contrôle.

Un diagramme d'états-transitions peut également représenter ce déroulement d'étapes. Toutefois, étant donné la nature procédurale de la réalisation des opérations – dans laquelle la plupart des événements correspondent simplement à la fin de l'activité précédente – il n'est pas nécessaire de distinguer systématiquement les états, les activités et les événements. Il est alors intéressant de disposer d'une représentation simplifiée pour la visualisation directe des activités. Dans ce contexte, une activité apparaît comme un stéréotype d'état. Une activité est représentée par un rectangle arrondi, comme les états, mais plus étiré horizontalement.

La figure suivante affiche les représentations graphiques des activités et des états, et met en évidence la simplification apportée par la représentation directe des activités.

Figure 285 : Exemple de simplification graphique par représentation directe des activités.

Chaque activité représente une étape particulière dans l'exécution de la méthode englobante. Les activités sont reliées par des transitions automatiques, représentées par des flèches, comme les transitions dans les diagrammes d'états-transitions. Lorsqu'une activité se termine, la transition est déclenchée et l'activité suivante démarre. Les activités ne possèdent ni transitions internes, ni transitions déclenchées par des événements.

La figure suivante représente deux activités reliées par une transition automatique. Il n'est pas nécessaire de faire figurer un nom d'événement sur la transition.

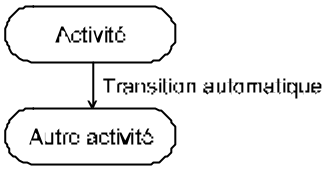


Figure 286 : Représentation d'une transition automatique.

Les transitions entre activités peuvent être gardées par des conditions booléennes, mutuellement exclusives. Les gardes se représentent à proximité des transitions dont elles valident le déclenchement.

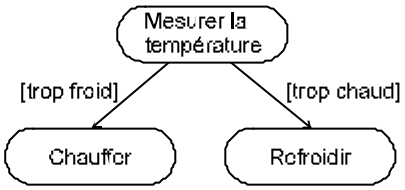


Figure 287 : Exemple de représentation des conditions.

UML définit un stéréotype optionnel pour la visualisation des conditions. Une condition est matérialisée par un losange d'où sortent plusieurs transitions. Le diagramme suivant est équivalent au diagramme précédent, si ce n'est que la condition est matérialisée explicitement.

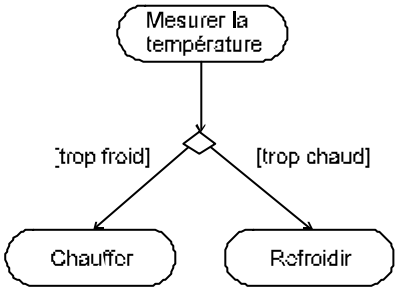


Figure 288 : Exemple d'activité stéréotypée pour représenter une décision.

Les diagrammes d'activités représentent les synchronisations entre flots de contrôles, au moyen de barres de synchronisation. Une barre de synchronisation

permet d'ouvrir et de fermer des branches parallèles au sein d'un flot d'exécution d'une méthode ou d'un cas d'utilisation. Les transitions au départ d'une barre de synchronisation sont déclenchées simultanément.

L'exemple suivant montre que pour refroidir, il faut simultanément arrêter le chauffage et ouvrir les fenêtres.

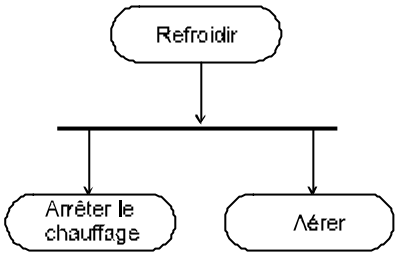


Figure 289 : Exemple de synchronisation de flots de contrôle parallèles à partir d'une barre de synchronisation.

Inversement, une barre de synchronisation ne peut être franchie que lorsque toutes les transitions en entrée sur la barre ont été déclenchées. La figure suivante reprend l'exemple précédent et montre que la mesure de température est effectuée une fois que le chauffage est arrêté et la pièce aérée.

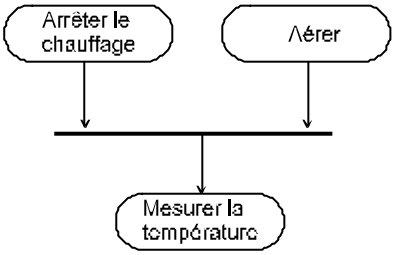


Figure 290 : Exemple de fusion de flots de contrôle parallèles regroupés sur une barre de synchronisation.

Les diagrammes d'activités peuvent être découpés en couloirs d'activités (comme une piscine est découpée en couloirs de natation) pour montrer les différentes responsabilités au sein d'un mécanisme ou d'une organisation. Chaque responsabilité est assurée par un ou plusieurs objets et chaque activité est allouée à un couloir donné. La position relative des couloirs n'est pas significative ; les transitions peuvent traverser librement les couloirs.

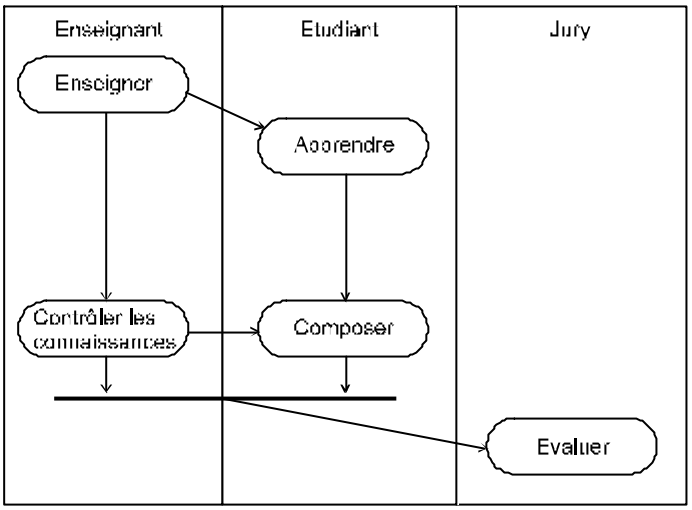


Figure 291 : Exemple de partition d'un diagramme d'activités en couloirs d'activités.

Il est possible de faire apparaître clairement les objets dans un diagramme d'activités, soit au sein des couloirs d'activités, soit indépendamment de ces couloirs. Les objets sont alors représentés par des barres verticales, comme dans les diagrammes de séquence ; les activités apparaissent objet par objet sur la ligne de vie de ces objets.

Souvent, différentes activités manipulent un même objet qui change alors d'état selon le degré d'avancement du mécanisme. Pour augmenter la lisibilité, cet objet peut figurer à plusieurs reprises dans les diagrammes ; son état est alors spécifié à chaque occurrence dans une expression entre crochets.

Les flots d'objets sont représentés par des flèches pointillées. Une flèche relie ainsi un objet à l'activité qui l'a créé. De même, une flèche relie un objet aux activités qui le mettent en jeu. Lorsqu'un objet produit par une activité est utilisé immédiatement par une autre activité, le flot d'objets représente également le flot de contrôle ; il est alors inutile de représenter explicitement ce flot de contrôle.

Le diagramme suivant illustre les différents cas de figure évoqués dans ce paragraphe. En particulier, l'exemple fait apparaître un objet **`Commande`** manipulé par différentes activités.

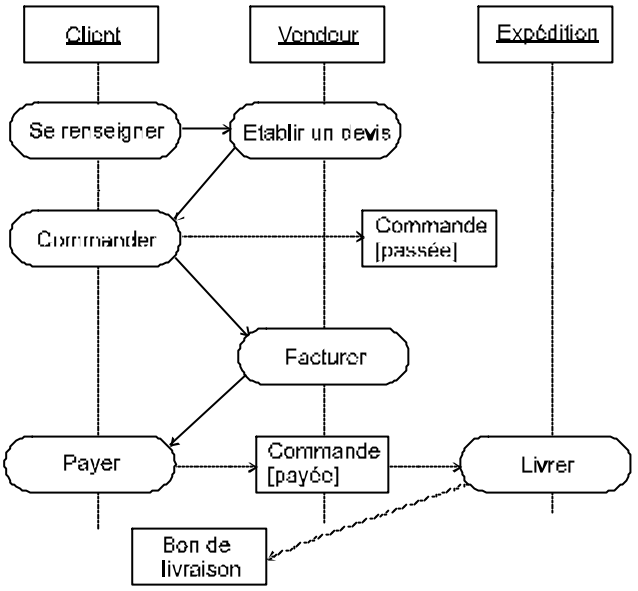


Figure 292 : Visualisation directe des objets responsables des différentes activités.

Les diagrammes d'activités peuvent également contenir des états et des événements représentés de la même manière que dans les diagrammes d'états-transitions. Le diagramme suivant donne un exemple d'emploi simultané des deux notations.

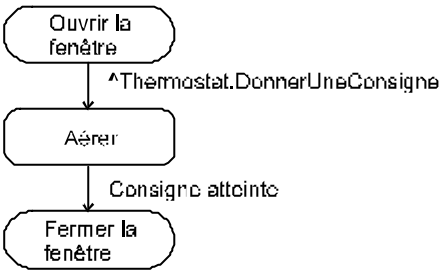


Figure 293 : Exemple de représentation mixte, incluant des états et des activités.

UML définit également des stéréotypes pour la représentation explicite des informations de transitions. L'envoi d'un signal se symbolise par un pentagone convexe relié par une flèche pointillée à l'objet destinataire du signal. L'attente d'un signal se symbolise par un pentagone concave relié par une flèche pointillée à l'objet émetteur du signal.

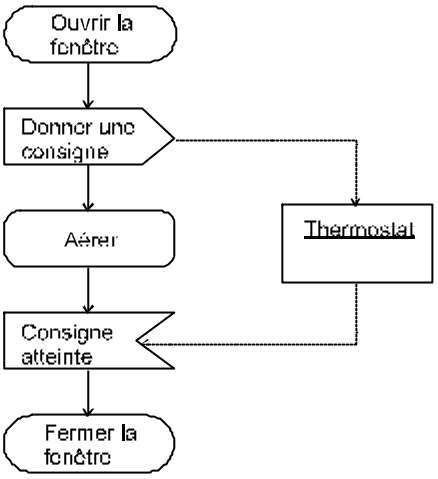


Figure 294 : Exemple de stéréotypes qui présentent graphiquement l'envoi et la réception de signaux.

Les diagrammes de composants

Les diagrammes de composants décrivent les éléments physiques et leurs relations dans l'environnement de réalisation. Les diagrammes de composants montrent les choix de réalisation.

Les modules

Les modules représentent toutes les sortes d'éléments physiques qui entrent dans la fabrication des applications informatiques. Les modules peuvent être de simples fichiers, des paquetages du langage Ada, ou encore des bibliothèques chargées dynamiquement.

Par défaut, chaque classe du modèle logique est réalisée par deux composants : la spécification et le corps. La spécification contient l'interface de la classe alors que le corps contient la réalisation de cette même classe. La spécification peut être générique dans le cas des classes paramétrables.

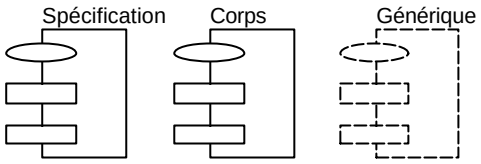


Figure 295 : Représentations graphiques des différentes sortes de modules.

La spécification et le corps d'une même classe peuvent être superposés dans les diagrammes afin de rendre la notation plus compacte. Chaque corps dépend alors implicitement de sa spécification.

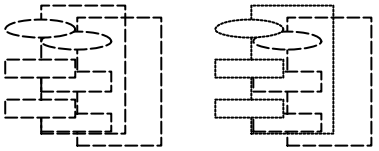


Figure 296 : Représentations compactes des spécifications et des corps.

En C++, une spécification correspond à un fichier avec un suffixe **`.h`** et un corps à un fichier avec un suffixe **`.cpp`**. En Ada, la notion de module existe directement dans le langage sous l'appellation de paquetage.

Les dépendances entre composants

Les relations de dépendance sont utilisées dans les diagrammes de composants pour indiquer qu'un composant fait référence aux services offerts par un autre composant. Ce type de dépendance est le reflet des choix de réalisation. Une relation de dépendance est représentée par une flèche pointillée qui pointe de l'utilisateur vers le fournisseur.

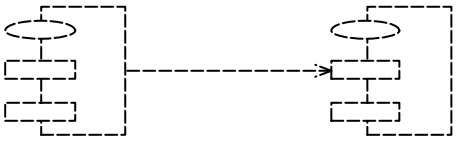


Figure 297 : La relation de dépendance permet de relier les différents composants.

La relation de dépendance peut être spécialisée par un stéréotype afin de préciser la nature des choix de réalisation qui entraînent la relation de dépendance. L'exemple suivant montre la construction d'un composant **`X`** à partir d'une liste

générique. La procédure **`Itérer`** est un ami de **`X`**, ami étant pris au sens C++ de *friend*.

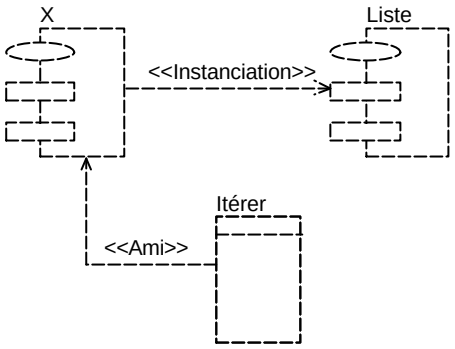


Figure 298 : Emploi de stéréotypes pour préciser la nature des choix de réalisation.

Dans un diagramme de composants, les relations de dépendance représentent généralement les dépendances de compilation. L'ordre de compilation est donné par le graphe des relations de dépendance.

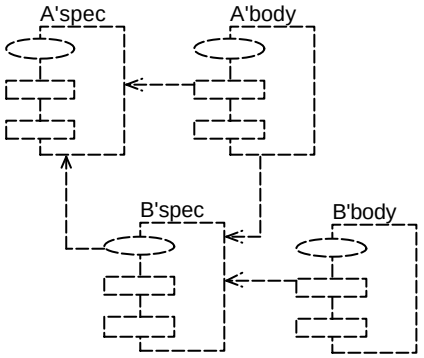


Figure 299 : Représentation des dépendances de compilation dans un diagramme de composants.

Les processus et les tâches

Les tâches correspondent à des composants qui possèdent leur propre flot (*thread* en anglais) de contrôle. Les tâches peuvent être contenues par d'autres composants comme les unités de compilation du langage Ada. Comme pour tous les éléments de modélisation, l'ajout de stéréotypes permet de préciser la sémantique d'un composant dynamique. Les stéréotypes `<<Processus>>` et

`<<Flot>>` sont prédéfinis par UML. Plusieurs flots peuvent partager le même espace d'adressage au sein d'un processus.

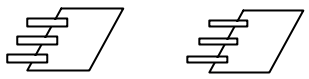


Figure 300 : Représentations graphiques des spécifications et corps de tâches.

Les programmes principaux

Les points d'entrée dans les applications sont identifiés par l'icône suivante :

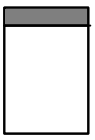


Figure 301 : Représentation graphique des programmes principaux.

En C++, le programme principal correspond à une fonction libre appelée *main*, stockée dans un fichier avec un suffixe **`.cpp`**. En Ada, toute procédure de bibliothèque peut être désignée comme programme principal.

Le nom du programme principal est souvent utilisé par l'éditeur de lien pour nommer le programme exécutable correspondant à l'application. Ceci permet, entre autres, de relier le modèle des composants avec le modèle des processus.

Les sous-programmes

Les sous-programmes regroupent les procédures et les fonctions qui n'appartiennent pas à des classes. Ces composants peuvent contenir des déclarations de types de base nécessaires pour la compilation des sous-programmes. En revanche, ils ne contiennent jamais de classes.

Il existe deux représentations graphiques : une pour la spécification des sous-programmes et une pour leur réalisation.

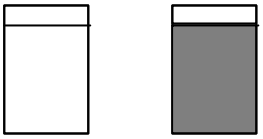


Figure 302 : Représentations graphiques des spécifications et réalisations des sous-programmes.

En Ada, il est possible de définir le corps d'un sous-programme seul : la spécification est alors implicite et ne figure donc pas dans le diagramme.

Les sous-systèmes

Pour faciliter la réalisation des applications, les différents composants peuvent être regroupés dans des paquetages selon un critère logique. Ils sont souvent stéréotypés en sous-systèmes pour ajouter les notions de bibliothèques de compilation et de gestion de configuration à la sémantique de partition déjà véhiculée par les paquetages. Les sous-systèmes jouent pour les composants le même rôle que les catégories pour les classes.

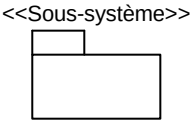


Figure 303 : Représentation graphique des sous-systèmes à partir d'un paquetage et d'un stéréotype.

Les sous-systèmes organisent la vue de réalisation d'un système ; chaque sous-système peut contenir des composants et d'autres sous-systèmes. Par convention, tout composant du modèle est placé soit au niveau racine, soit dans un sous-système. Les sous-systèmes doivent être vus comme de grandes briques pour la construction des systèmes.

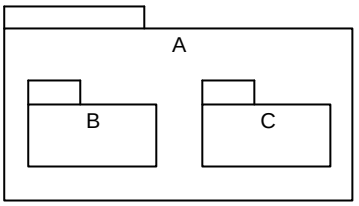


Figure 304 : Les sous-systèmes peuvent être emboîtés les uns dans les autres.

La décomposition en sous-systèmes n'est pas une décomposition fonctionnelle. Les fonctions du système sont exprimées du point de vue de l'utilisateur dans la vue des cas d'utilisation. Les cas d'utilisation se traduisent en interactions entre objets dont les classes sont elles-mêmes encapsulées par des catégories. Les objets qui réalisent les interactions sont distribués dans les différentes catégories ; le code correspondant est stocké dans des modules et des sous-systèmes.

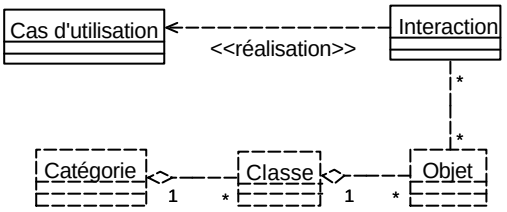


Figure 305 : Les objets interagissent pour réaliser les comportements décrits fonctionnellement dans les cas d'utilisation.

Le sous-système est, dans la vue physique, l'équivalent de la catégorie dans la vue logique. La figure suivante montre la correspondance entre les deux vues.

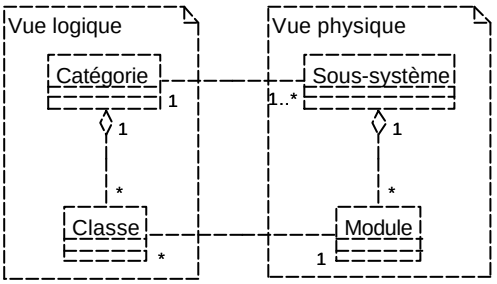


Figure 306 : Extrait du métamodèle montrant la correspondance entre la vue logique et la vue physique.

Un sous-système est un moyen de gérer la complexité par l'encapsulation des détails. Les sous-systèmes possèdent une partie publique et une partie privée. Sauf indication contraire explicite, tout module placé dans un sous-système est visible de l'extérieur.

Les sous-systèmes peuvent dépendre d'autres sous-systèmes et de composants. De même, un composant peut dépendre d'un sous-système.

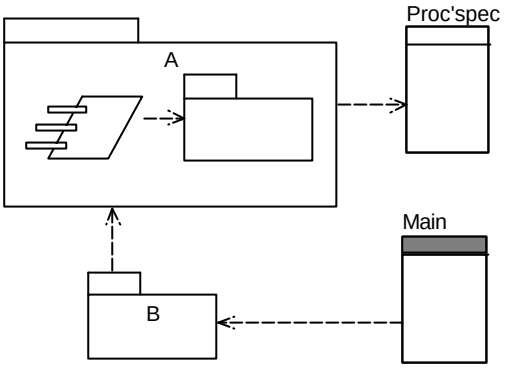


Figure 307 : Relations de dépendance entre différents types de composants et sous-systèmes.

Intégration avec les environnements de développement

Les sous-systèmes n'existent pas en tant que tels dans tous les environnements de développement de logiciel. Il appartient alors à l'utilisateur de mettre en place une structure à base de répertoires et de fichiers pour les réaliser physiquement.

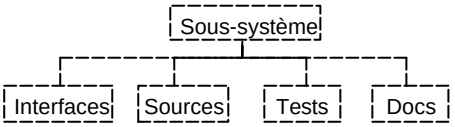


Figure 308 : Réalisation d'un sous-système à partir d'une structure de répertoires et de fichiers.

Dans cette optique, il est particulièrement judicieux de se servir de la notion de sous-système comme d'un point d'intégration avec les outils d'analyse et de conception de logiciels, les systèmes de compilation et les systèmes de gestion de versions et de configurations.

Chaque sous-système est matérialisé par un répertoire qui contient des fichiers ; ces fichiers correspondent aux différents composants contenus par le sous-système. Le sous-système contient en plus les différents fichiers nécessaires à la compilation, à la documentation et au test des composants. L'intégration avec les

systèmes de compilation permet d'associer la notion de bibliothèque de programmes à la notion de sous-système. L'intégration avec un gestionnaire de configurations conduit à la construction de systèmes par combinaison de bibliothèques de compilation. L'intégration peut encore être enrichie par un gestionnaire de versions, idéalement à deux niveaux de granularité. Il devient alors possible de construire un système par combinaison de versions de sous-systèmes (exprimées sous la forme de bibliothèques de programmes), elles-mêmes composées de versions de composants.

Les diagrammes de déploiement

Les diagrammes de déploiement montrent la disposition physique des différents matériels (les nœuds) qui entrent dans la composition d'un système et la répartition des programmes exécutables sur ces matériels.

Représentation des nœuds

Chaque ressource matérielle est représentée par un cube évoquant la présence physique de l'équipement dans le système. Tout système est décrit par un petit nombre de diagrammes de déploiement ; souvent un seul diagramme suffit.

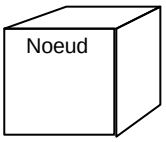


Figure 309 : Représentation graphique des nœuds.

La nature de l'équipement peut être précisée au moyen d'un stéréotype. L'exemple suivant propose trois stéréotypes pour distinguer les dispositifs, les processeurs et les mémoires. Au besoin, l'utilisateur a la possibilité de définir d'autres stéréotypes.

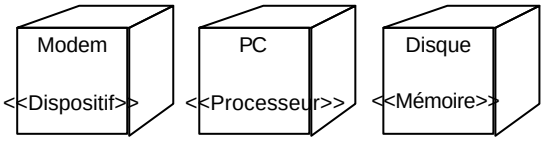


Figure 310 : Exemples de stéréotypes de nœud.

La distinction entre un dispositif et un processeur dépend fortement du point de vue. Un terminal X sera vu comme un dispositif par l'utilisateur du terminal alors

qu'il correspondra à un processeur pour le développeur du serveur X qui s'exécute sur le processeur embarqué dans le terminal X.

Les différents nœuds qui apparaissent dans le diagramme de déploiement sont connectés entre eux par des lignes qui symbolisent un support de communication *a priori* bidirectionnel. La nature de ce support peut être précisée au moyen d'un stéréotype.

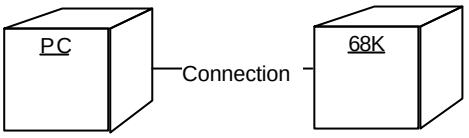


Figure 311 : Représentation graphique des connexions entre nœuds.

Les diagrammes de déploiement peuvent montrer des classes de nœuds ou des instances de nœuds. Comme dans les autres types de diagrammes, la distinction graphique entre les classes et les objets est réalisée en soulignant le nom de l'objet. L'exemple suivant montre le diagramme de déploiement d'un système de gestion des accès d'un bâtiment.

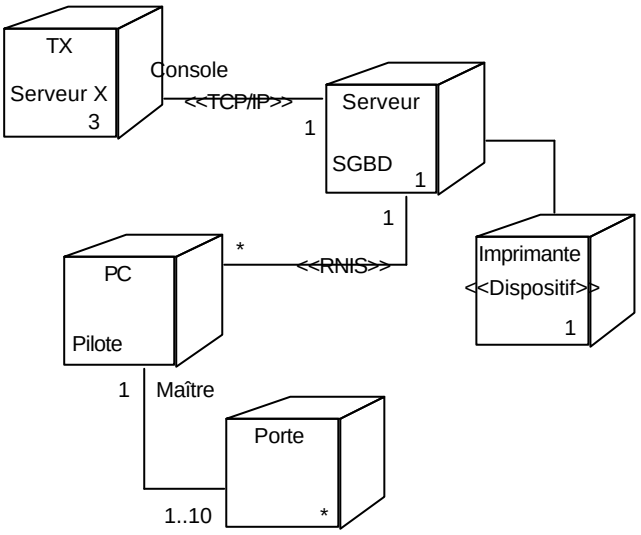


Figure 312 : Exemple de classes dans un diagramme de déploiement.

La présence d'informations de multiplicité et de rôle confirme que le diagramme précédent représente des classes de nœuds. Le diagramme montre que le système est constitué d'un serveur autour duquel gravitent des PC pilotant l'ouverture et la fermeture de portes. Le nombre de PC n'est pas précisé. En revanche, il apparaît

que chaque PC peut être le maître de 10 portes au plus. Trois terminaux X jouent le rôle de console pour accéder au système ; une imprimante est reliée au serveur.

Le diagramme décrit également la nature des liens de communication entre les différents nœuds. Le serveur et les PC sont reliés par une liaison RNIS ; les terminaux X et le serveur communiquent via TCP/IP. La nature de la connexion des autres nœuds n'a pas été précisée.

Les nœuds qui correspondent à des processeurs du point de vue de l'application – le stéréotype de processeurs n'apparaît pas obligatoirement dans les diagrammes de déploiement – portent également le nom des processus qu'ils hébergent. Le serveur exécute un système de gestion de bases de données ; les PC hébergent le logiciel de pilotage dédié à la commande et au contrôle des portes. Le nom de ces processus permet de faire le lien entre le diagramme de déploiement et le diagramme des composants. Chaque processus nommé dans le diagramme de déploiement exécute un programme principal du même nom, décrit dans le diagramme des composants.

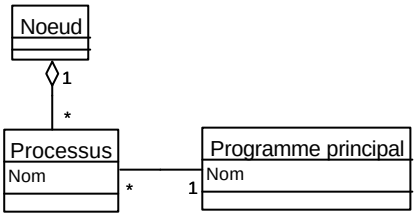


Figure 313 : Le nom des processus et des programmes principaux permet de faire le lien entre les diagrammes de déploiement et de composants.

Les diagrammes de déploiement peuvent également montrer des instances de nœuds (reconnaissables au nom souligné). Le diagramme suivant nous renseigne ainsi sur la situation précise au niveau du site d'implantation du système. Il apparaît que les portes 6, 7, 8 et 9 sont pilotées par le PC 4.

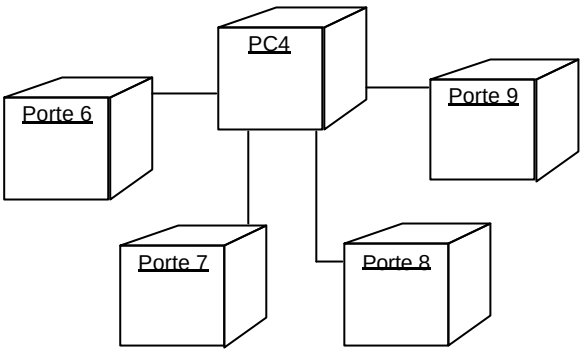


Figure 314 : Exemple d'objets dans un diagramme de déploiement. Les noms des objets sont soulignés afin de distinguer les objets des classes.

4

Encadrement des projets objet

La présence d'un processus de développement formalisé, bien défini et bien géré, est un facteur de réussite d'un projet. Un processus est stable si ses performances futures peuvent être prédites statistiquement[18].

UML étant avant tout un langage de modélisation elle ne définit pas un processus de développement particulier. Néanmoins, elle peut servir de notation pour différentes approches méthodologiques basées sur les objets.

L'objectif de ce chapitre est de présenter les grandes lignes de l'encadrement des projets objet au travers d'un cadre méthodologique générique proche de celui décrit par Objectory 4.0[19].

Il appartiendra à chaque projet ou organisation de raffiner ce cadre en fonction de ses besoins.

---

[18] Deming W. E. 1982, *Quality, Productivity, and Competitive Position*. Massachusetts Institute of Technology Center for Advanced Engineering Study, Cambridge, MA.

[19] Rational Software Corporation 1996, *Rational Objectory Process – Introduction*.

Caractérisation du logiciel

Ce chapitre a pour objet de caractériser le logiciel, de mettre en évidence l'origine des difficultés liées à son développement et d'esquisser les mesures à prendre pour les maîtriser.

La crise du logiciel

La formule est née il y a trente ans[20] pour décrire la limite des technologies disponibles et l'incapacité à maîtriser le logiciel qui en résulte. Bien sûr, au cours de ces trente dernières années, de nombreux progrès ont été réalisés, mais le logiciel est toujours en crise. Il est probable, comme le suggère Grady Booch, qu'il ne s'agit pas d'une crise mais plutôt d'un état permanent.

Les méthodes objet ont pris le relais des techniques structurées traditionnelles ; les logiciels objet sont plus extensibles, plus proches des utilisateurs, plus faciles à maintenir, mais ils restent toujours difficiles à réaliser.

Il faut bien le dire, une des raisons de la crise du logiciel est la fuite en avant vers la complexité des applications. La complexité intrinsèque de chaque nouvelle génération de système est plus grande de sorte que les solutions techniques qui étaient adaptées pour la génération précédente arrivent à leurs limites pour la génération suivante. La crise du logiciel ne connaît pas de fin.

Il existe plusieurs types de logiciels qui peuvent chacun se caractériser par les contraintes qu'ils placent sur le processus de développement.

Les catégories de logiciels

Les logiciels amateurs constituent la première catégorie. Le terme amateur ne doit pas être pris dans un sens péjoratif : il désigne tout logiciel qui n'a pas d'impact économique significatif. Ce type de logiciel est généralement développé par des individus au sein de groupes qui partagent des centres d'intérêt communs, comme par exemple les radioamateurs ou les astronomes amateurs. Les logiciels développés par les étudiants tombent également dans cette catégorie.

La deuxième catégorie regroupe les logiciels jetables, dits aussi logiciels consommables, comme les traitements de texte ou les tableurs. Ce type de logiciel ne coûte pas très cher à l'achat (quelques milliers de francs tout au plus) et son remplacement par un logiciel équivalent ne risque pas de mettre en péril l'équilibre financier de l'entreprise qui en a fait l'acquisition. Le point de vue du fabricant de

---

[20] Buxton J. N. & Randell B. (eds) 27-31 Octobre 1969, *Software Engineering Techniques*. Report on a Conference Sponsored by the NATO Science Committee, Rome, Italy.

ce type de logiciel est quelque peu différent, et cela d'autant plus que la diffusion de ce type de logiciel est large. Une fois le logiciel en exploitation chez des milliers de clients, éventuellement du monde entier, il n'est guère aisé de proposer des corrections de défauts, d'autant que les revenus dérivés des contrats de maintenance sont faibles. Il faut donc maintenir un équilibre difficile entre qualité du produit, coût de développement et diffusion. L'objectif de qualité à moindre coût est difficile à tenir, mais impératif pour les produits à marge faible et grande distribution. Internet et le principe du *shareware* simplifient énormément la diffusion de ce type de logiciel, mais ne résorbent pas l'impératif de qualité.

La troisième catégorie comprend les logiciels essentiels au fonctionnement d'une entreprise et qui ne peuvent être échangés facilement : ils ont été construits pour une tâche donnée et sont le fruit d'un investissement non négligeable. Ce type de logiciel exige un comportement fiable, sûr et prévisible : leur défaillance peut entraîner la chute de l'entreprise qui les utilise. Ils sont présents à la fois dans les secteurs industriels et tertiaires, par exemple pour piloter un autocommutateur ou pour gérer une salle des marchés. La grande complexité de ces logiciels – plus précisément de ces systèmes logiciels – provient d'une part de la complexité intrinsèque liée au domaine des applications et d'autre part de la complexité des environnements d'exécution, souvent hétérogènes et distribués. Leur maintenance évolutive devient de plus en plus difficile car chaque modification rend le logiciel plus confus. Les informaticiens subissent alors le poids du passé, souvent jusqu'à un point de rupture qu'il est essentiel d'anticiper. La triste réalité est que l'entropie des logiciels est à l'image de l'entropie de l'univers : elle ne cesse de croître. De ce point de vue, un processus de développement doit limiter le désordre logiciel.

La dernière catégorie de logiciels englobe les systèmes vitaux, c'est-à-dire ceux dont dépend la vie d'êtres humains. Ces systèmes se rencontrent dans les domaines du transport de l'armement et de la médecine. Les contraintes d'exploitation de ces logiciels n'ont absolument rien à voir avec les catégories précédentes. Un dysfonctionnement ne se chiffre plus en argent, mais en vies humaines.

La complexité des logiciels

Les informaticiens, de manière générale, ont un grave défaut. Ils ne savent pas faire simple. D'un autre côté, faire simple est extrêmement difficile.

Faire simple, c'est par exemple apprendre à encapsuler la complexité ; c'est aussi lutter contre le plaisir que procure le maniement des arcanes d'un langage de programmation. Faire simple, c'est également apprendre à faire semblant que c'est simple, c'est créer l'illusion de la simplicité.

L'informaticien doit apprendre à imiter la nature qui, dans ses constructions les plus élémentaires comme les plus complexes, cherche toujours l'équilibre stable, la consommation d'énergie minimale, loin du caprice ou du superflu.

Trop souvent, l'élaboration des logiciels reste une activité parcellaire et dispersée. Si l'informaticien construisait des meubles, il commencerait par planter des glands, débiterait les arbres en planches, creuserait le sol à la recherche de minerai de fer, fabriquerait une forge pour faire ses clous et finirait par assembler le tout pour obtenir un meuble. Ce mode de fabrication engendre une large dispersion des efforts et requiert la maîtrise d'une trop grande palette de compétences. Durant les siècles passés, nos ancêtres ont appris à spécialiser les activités humaines et c'est ainsi que certains spécialistes savent comment faire pousser les arbres alors que d'autres connaissent les secrets de fabrication des clous. L'avantage immédiat de cette approche est que les autres personnes peuvent se concentrer sur leur propre domaine de connaissance, par exemple la fabrication des meubles. Un mode de développement qui repose sur les épaules de quelques programmeurs héroïques, de gourous et autres magiciens du logiciel, ne constitue pas une pratique industrielle pérenne et reproductible.

La formalisation du processus de développement procède de cette constatation. Les activités mises en œuvre lors de l'élaboration du logiciel doivent être décrites, expliquées, motivées : la connaissance et le savoir-faire informatique peuvent alors se propager autrement que de bouche de druide à oreille de druide[21]. Un processus décrit l'enchaînement des activités de l'équipe de développement ; il dirige les tâches de chaque individu ainsi que celles du groupe. Le processus définit des points de mesures et des critères pour contrôler et mesurer les produits et les activités du projet.

Malheureusement, la formalisation du processus de développement ne s'applique que partiellement au domaine informatique. Contrairement à l'ébénisterie, l'informatique est un domaine de connaissance très jeune et pauvre en savoir-faire transmissibles. De plus, le niveau de formalisation du savoir-faire informatique est très faible. Avant de vouloir propager l'expérience, il faut d'abord savoir la décrire, la représenter, la modéliser.

Cela revient également à dire que l'informatique est un métier qui s'apprend comme les autres métiers : il n'est pas possible de s'improviser informaticien. Une entreprise qui recherche un spécialiste de l'informatique doit donc rechercher un ingénieur informaticien plutôt qu'un spécialiste de la chimie ou du textile.

Origines de la complexité des logiciels

Il ne suffit pas de jeter la pierre aux informaticiens : de plus en plus, la problématique qui leur est soumise est complexe de par la nature même du

[21] Gosciny, Uderzo1961, Astérix le Gaulois, Dargaud.

domaine des applications. Avec la meilleure volonté du monde, cette complexité ne peut être réduite par l'informaticien. En revanche, elle peut être canalisée, confinée dans des espaces clos faiblement couplés. Par une segmentation judicieuse de l'espace des états, la caractérisation du comportement des systèmes discrets – que sont les programmes – est facilitée.

De la même manière, la complexité liée à l'environnement informatique, c'est-à-dire aux méthodes, aux langages, aux systèmes d'exploitation, etc, peut et doit être minimisée le plus possible : il faut éviter les solutions moins onéreuses mais plus complexes.

En amont de ces problèmes de complexité, il faut insister sur l'infinie flexibilité du logiciel qui est à la fois son plus grand avantage et son plus grand défaut. Il est extrêmement facile d'écrire une ligne de programme ; il est plus difficile de la mettre au point. Il est tout aussi facile de modifier une ligne dans un programme ; il est infiniment plus difficile de garantir que le programme continuera de fonctionner correctement.

Conséquence de la complexité

Du fait de la complexité liée à l'élaboration des logiciels, il y a fort peu de chances qu'un logiciel soit exempt de défauts. Au contraire, le risque de déficience grave est élevé.

Les défauts logiciels, les bogues, sont comme les couacs en musique[22]. Les meilleurs musiciens ne sont pas à l'abri d'un couac ; ce qu'ils font est difficile et assurer une qualité constante, sur une longue durée, l'est aussi. L'informaticien est confronté au même problème, quelles que soient ses compétences ; il n'est pas à l'abri d'un couac. Les logiciels sont écrits par des humains, les humains sont faillibles, les logiciels aussi.

La mise au point des logiciels est lente et chaotique, et le coût de maintenance est disproportionné. Les techniques de test et de validation permettent de dégager des chemins d'exécution sécurisés : suivis à la lettre, ils permettent de garantir un niveau de confiance raisonnable dans le logiciel.

Tous les défauts n'ont pas les mêmes conséquences. Les défauts d'analyse se traduisent par des logiciels qui ne satisfont pas leurs utilisateurs. Les défauts de conception génèrent des logiciels lourds à manier, trop lents ou peu efficaces. Les défauts de réalisation induisent des comportements non prévus. Tout défaut est indésirable ; ceux introduits dans les activités en amont restent cependant les plus difficiles à corriger.

---

[22] Adda J.-L., communication privée.

La crise de logiciel est le fruit de notre incapacité à maîtriser les défauts du logiciel. Le résultat de cette crise est le coût astronomique des applications complexes.

La portée de l'approche objet

Le développement d'une application peut être divisé en plusieurs grandes parties : elles s'enchaînent de manière séquentielle au sein d'un cycle de vie en cascade ou elles se distribuent sur les différentes itérations d'un cycle de vie itératif.

De manière générale, quelle que soit la manière de procéder, linéairement ou itérativement, selon une approche structurée ou objet, le développement d'une application répond à quatre questions :

Application = Quoi + Dans quel domaine + Comment + Avec quelles compétences

Ces questions correspondent à différents points de vue et concernent différents intervenants. Elles peuvent être étudiées selon des techniques variées, mais doivent dans tous les cas être considérés pour développer une application :

- *Quoi faire ?* La réponse est exprimée par l'utilisateur qui décrit ce qu'il attend du système, comment il entend interagir avec lui et quels sont les différents intervenants. Il s'agit d'une description fonctionnelle qui ne rentre pas dans les détails de la réalisation : le *quoi faire* est purement descriptif.
- *Dans quel domaine ?* La réponse doit décrire le domaine (l'environnement) dans lequel l'application va exister et préciser quels sont les éléments pertinents dans ce domaine pour l'application. L'étude du domaine est le fruit d'une analyse, totalement déconnectée de toute considération de réalisation. Le domaine est analysé par un analyste et doit pouvoir être compris par un utilisateur.
- *Comment ?* Il faut le déterminer lors de la conception. Le *comment* est le fruit de l'expérience et du savoir-faire du concepteur. La conception est l'art de rendre possible les désirs de l'utilisateur – exprimés dans le *quoi faire* – en considérant le domaine de l'application et en tenant compte des contraintes de réalisation.
- *Avec quelles compétences ?* Il faut déterminer tout ce qui est nécessaire à la fabrication de l'application. Ce point repose sur des compétences techniques pour le développement des classes, des objets et des mécanismes, sur des compétences d'animation pour l'encadrement des équipes et sur des compétences d'organisation pour assurer la logistique générale.

L'approche objet répond à ces questions, selon trois regards différents : l'analyse objet, la conception objet et la programmation objet.

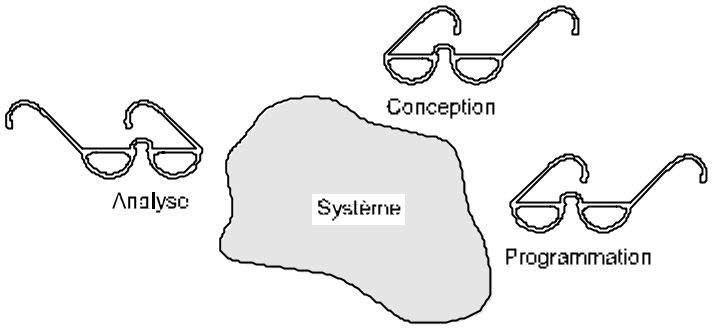


Figure 315 : L'approche objet porte trois regards différents sur les systèmes.

L'analyse objet

L'analyse remonte de la conséquence vers le principe : elle essaie de comprendre, d'expliquer et de représenter la nature profonde du système qu'elle observe. L'analyse ne se préoccupe pas de solutions mais de questions ; elle identifie le *quoi faire* et l'environnement d'un système, sans décrire le *comment* qui est le propre de la conception.

L'analyse commence par la détermination du *quoi faire*, c'est-à-dire des besoins de l'utilisateur. Bien souvent, l'utilisateur n'est pas capable d'exprimer clairement ses attentes, de sorte que le cahier des charges n'est qu'une représentation approximative de ses besoins réels. La présence d'un imposant **`cahier des charges`** n'est pas toujours de bon augure. Sa qualité dépend très fortement de la technique employée pour son élaboration. Trop souvent, les cahiers des charges sont touffus, confus, contiennent des points contradictoires et ne reflètent pas les vrais besoins des utilisateurs. L'expérience montre que la technique des cas d'utilisation (*use case*s) se prête bien à la détermination des besoins de l'utilisateur.

L'analyse se poursuit par la modélisation du domaine de l'application, c'est-à-dire par l'identification des objets et des classes qui appartiennent fondamentalement à l'environnement de l'application, et par la représentation des interactions entre ces objets. L'analyse rend compte de la structure statique au moyen des relations entre les classes et des interactions entre les objets au moyen des messages. Il n'existe pas d'ordre préférentiel dans l'élaboration des différents types de diagrammes : ils sont souvent construits simultanément. Parmi les nombreux diagrammes définis par UML, ceux qui montrent des éléments de la vue logique peuvent être utilisés en analyse, en particulier :

- les cas d'utilisation,

- les spécifications de classes et d'objets,
- les diagrammes de collaboration,
- les diagrammes de séquence,
- les diagrammes de classes,
- les diagrammes d'états-transitions.

Cette liste n'est pas limitative et chaque projet est libre d'utiliser d'autres techniques pour parfaire l'analyse de son problème. En particulier, les approches cognitives et systémiques apportent souvent un éclairage intéressant et complémentaire de la modélisation objet.

L'assise sur les éléments du monde réel facilite le départ de la démarche car elle concentre la pensée sur des éléments concrets. Elle doit impérativement être poursuivie par une démarche d'abstraction qui vise à faire oublier les contingences du monde réel et à déterminer les concepts fondateurs qui sont à leurs origines. Si la réflexion se focalise trop sur l'existant, les risques de reproduire une solution au lieu d'identifier le problème posé sont grands (voir plus loin le paragraphe sur la place de l'informatique ou le choc des cultures).

L'analyse est l'antithèse de la conformité. L'analyse est souvent surprenante car elle demande de changer de point de vue, d'oublier ce qui est connu, ce qui s'impose de prime abord, afin de trouver l'essentiel, la nature cachée des choses. Le propre de l'analyse est de trouver une description à laquelle personne n'avait pensé jusqu'alors, mais qui une fois déterminée s'impose au point d'être évidente.

Pour prendre une image, analyser c'est regarder un point, un cercle, une ellipse, une droite, deux droites sécantes, une hyperbole et une parabole et reconnaître que toutes ces formes peuvent être obtenues par l'intersection d'un plan et d'un cône. Dans ce cas, le principe générateur peut être exprimé par une équation de la forme $ax^2 + by^2 + 2cx + 2dy + e = 0$ Cette représentation est économe car elle décrit l'essentiel. L'analyse va à l'essentiel et recherche un modèle générique plus abstrait.

La conséquence immédiate d'une analyse bien menée est toujours une grande économie dans la réalisation qui s'ensuit, en vertu de la dichotomie (essence, manifestation) sous-tendu par l'approche objet. L'identification du principe générateur permet de réaliser en une seule fois ce qui se manifeste généralement sous plusieurs formes, puis de l'instancier pour obtenir les différentes manifestations désirées, voire souvent d'autres manifestations non encore présentes dans le domaine.

Parallèlement à l'analyse de l'environnement, se déroule parfois une analyse de l'existant et des contraintes de réalisation. L'objet de cette analyse est de comprendre parfaitement les caractéristiques et les contraintes de l'environnement de réalisation afin de pouvoir prendre des décisions motivées et

réfléchies lors de la conception. Cette analyse est le seul moyen de prendre en compte de manière réaliste les choix de réalisation qui peuvent être imposés à un projet, avant même la conception. Cette situation est malheureusement trop courante en informatique. Les informaticiens doivent résoudre un problème à partir d'un fragment de solution !

Au passage à la conception, il faut prendre garde à ne pas perdre la vue d'analyse. La perte de cette vue est la conséquence de l'enrichissement des modèles durant la conception. Comme il a été dit plus haut, la conception apporte des informations complémentaires qui viennent enrichir les descriptions effectuées en analyse. La tentation est forte de partir des diagrammes obtenus en analyse et de rajouter l'information de conception dans ces diagrammes. L'analyse se fond alors progressivement dans la conception et les diagrammes ne montrent plus les objets du domaine mais le *comment*. La situation est souvent autrement complexe car les modèles subissent des transformations qui sont plus que de simples enrichissements ; un filtrage ne suffit pas pour retrouver la vue d'analyse.

Il n'existe pas de solution simple au problème de la transformation des modèles, en particulier de la transformation bidirectionnelle ; lorsque les éclaircissements de la conception peuvent entraîner des modifications de l'analyse par effet rétroactif. UML définit le concept de trace entre éléments de modélisation, y compris d'un modèle à l'autre, de sorte qu'il est possible d'enregistrer l'histoire de ces éléments.

La conception objet

Comme l'analyse et la conception, la conception et la réalisation se chevauchent en partie. Il est rare de savoir comment tout faire : pour cette raison il faut essayer des techniques alternatives, comparer les résultats puis choisir en faisant des compromis. La conception et la réalisation se complètent. La conception a besoin d'expérimenter, la réalisation a besoin de principes directeurs.

Afin de maintenir le plus longtemps possible les bénéfices de l'abstraction, la conception est généralement subdivisée en deux étapes : une étape logique indépendante de l'environnement de réalisation et une étape physique qui se préoccupe de l'ordonnancement des ressources et des particularités des langages de programmation ou de l'environnement d'exécution.

La conception est souvent considérée, à tort, comme un simple enrichissement des résultats obtenus durant l'analyse. Il s'agit là d'une vision fortement réductrice qui ignore tout simplement que la conception est le temps de la mise en œuvre du savoir-faire. Ce savoir-faire peut être interne au projet ou acquis à l'extérieur sous la forme d'outils, de composants réutilisables ou plus largement de cadres de développement. L'émergence des *patterns* (les *patterns* et les *frameworks* sont décrits plus loin) marque une avancée dans la formalisation du

savoir-faire objet, indépendamment des langages de programmation et de leurs subtils arcanes.

La conception débute par la détermination de l'architecture du logiciel, c'est-à-dire par l'élaboration des structures statiques et dynamiques qui serviront de charpente pour l'ensemble du développement. L'architecture définit la forme générale de l'application ; de sa qualité dépendent le développement et l'évolution du logiciel. L'architecture est la conséquence de décisions stratégiques arrêtées lors de la conception et de décisions tactiques prises en cours de réalisation.

Se donner une stratégie informatique, c'est considérer le logiciel comme un élément déterminant du développement de l'entreprise ; c'est aussi chercher à s'assurer une avance technologique par des choix d'architecture qui dépassent le cadre des besoins immédiats, liés à une application particulière.

Se donner une stratégie informatique, c'est par exemple choisir :

- *l'internationalisation*, autrement dit, concevoir le logiciel de telle manière que le dialogue avec l'utilisateur puisse être mené dans toutes les langues ;
- *la réutilisation* qui n'est pas le fruit du hasard, mais le résultat d'une adhésion forte de toute une organisation, depuis la définition de la ligne de produits jusqu'au développement de ces produits ;
- *l'unicité de langage* pour l'ensemble des développements (Ada dans le cas du ministère de la défense américain) ;
- *l'indépendance par rapport aux dispositifs d'affichage*, autrement dit, le découplage entre l'information à afficher et la manière de l'afficher ;
- *la portabilité* des sources, voire des exécutables, qui réduit notablement les coûts de développement et surtout de maintenance dans le cas d'un développement multicibles.
- *la généralisation des mécanismes de communication* entre objets qui permet de rendre transparente la localisation des objets et ainsi de faciliter la distribution des applications et de leurs composants.

La conception est le domaine du compromis. Il n'existe pas de solution toute blanche ou toute noire : la vérité est toujours quelque part dans le gris. Les décisions tactiques couvrent toutes les activités qui guideront la recherche de cette vérité dans l'effort de développement au jour le jour. Elles contiennent par exemple les éléments suivants :

- *la généralisation d'abstractions et de mécanismes pour les tâches ancillaires*, autrement dit, tous les composants largement réutilisables dans les programmes, comme les dates, les chaînes de caractères, les structures de données de base, les algorithmes de tri, etc. ;

- *le traitement des erreurs* qui peut être effectué par des exceptions ou par des statuts exprimés sous la forme de valeurs entières. L'exception ne peut être ignorée, mais son traitement obscurcit le code normal. Le statut d'erreur est simple à mettre en œuvre mais il peut être ignoré : le traitement de l'erreur n'est alors pas garanti ;
- *la gestion de la mémoire dynamique* qui peut être effectuée par le programmeur ou automatisée par l'environnement d'exécution ;
- *la communication entre processus* qui peut reposer sur des liaisons point-à-point, multi-points ou par diffusion. Elle peut être assurée par des mécanismes ad hoc ou inscrite dans un schéma général d'intégration d'application comme Corba ou ActiveX ;
- *le choix des modes de communication* qui inclut la forme et la nature des messages et les différents modes de synchronisation : synchrone, semi-synchrone, dérobante, minutée, asynchrone, etc ;
- *la présentation des messages utilisateur* afin de rendre toutes les interactions avec l'utilisateur le plus homogènes possible ;
- *les langages de programmation* qui sont compilés ou interprétés et présentent des points forts et des limites. Les langages compilés permettent d'écrire des programmes plus fiables, les langages interprétés apportent plus de souplesse lors du développement. Dans certains cas, il peut être intéressant de mélanger différents langages au sein de la même application ;
- *le choix des structures de données et des algorithmes* qui sont encapsulés dans les objets afin de découpler la spécification de la réalisation des classes ;
- *l'optimisation du réseau de relations* qui reflète les besoins de communication de l'utilisateur. Le concepteur peut être amené à modifier ce réseau pour diverses raisons : optimiser la vitesse des accès, prendre en compte des contraintes particulières de réalisation, comme l'absence de lien bidirectionnel dans les langages de programmation de troisième génération ;
- *la mutation de certaines relations de généralisation* parce que l'héritage multiple n'est pas disponible dans le langage de programmation utilisé ou encore parce que la forme de généralisation recherchée doit être dynamique alors que l'héritage est statique ;
- *la mise en œuvre de composants réutilisables* dans le cadre de deux réalités ; la programmation avec la réutilisation et la programmation pour la réutilisation. La première forme de réutilisation consiste à incorporer un composant existant et ainsi à accélérer un développement en cours. La deuxième forme s'apparente plus à un investissement : le coût est immédiat et les bénéfices ne sont perçus que plus tard.

L'effort développé en conception est plus ou moins important selon la nature de l'application et la richesse de l'environnement de réalisation. Il est de plus en plus

possible d'effectuer un transfert de complexité des applications vers les environnements, grâce à une factorisation des éléments et des mécanismes communs. C'est le cas dans les familles d'applications bien ciblées dans un domaine donné, comme les logiciels clients qui interagissent avec des serveurs de bases de données. Inversement, plus une application est technique, plus elle interagit avec des matériels particuliers et plus l'effort de conception est important.

Les outils RAD – développement rapide d'applications – proposent une voie bien tracée. Tout comme avec les *frameworks*, il faut suivre les règles du jeu fixées à l'avance : au bénéfice de la simplicité mais au détriment de l'originalité. Toutes les applications finissent par se ressembler, ce qui est bien et mal à la fois. Le travers des outils RAD est d'encourager à faire vite et salement – *quick and dirty* – plutôt que vite et bien.

Curieusement, le RAD est un mélange de génie logiciel dans sa conception et d'horreur informatique dans son utilisation. Ceci est la conséquence d'une vision à court terme des utilisateurs pour qui seul l'aspect rapidité compte bien souvent. Comme le RAD n'encourage pas particulièrement la modularité et les abstractions en couches, très rapidement – c'est le cas de le dire – le code d'interface utilisateur et le code de l'application se retrouvent intimement mélangés. Le logiciel construit de cette manière est difficile à maintenir et quasiment impossible à réutiliser pour une autre application.

Le propre de la conception est de rechercher l'équilibre entre vitesse de réalisation et possibilité de réutilisation, ouverture et solution clé en main, vision à court terme (la tactique) et vision à long terme (la stratégie).

La programmation objet

La programmation objet est le dernier maillon de la chaîne objet, après l'analyse et la conception. La programmation dans un langage objet est la manière la plus commode de traduire une conception objet en un programme exécutable par une machine. Il est également possible d'utiliser un langage non objet, mais ceci impose une charge de travail supplémentaire au programmeur qui doit simuler les constructions objet.

Parallèlement aux vrais langages objet, comme Smalltalk ou Java, il est possible de trouver des langages ou des environnements à teinture objet, par exemple Visual Basic de Microsoft ou encore Powerbuilder de Powersoft ; d'une certaine manière ils permettent de faire de l'objet en douceur, sans s'en rendre compte[23].

Les langages de programmation objet sont en majorité des langages algorithmiques et structurés. Ils offrent au moins les concepts de classes,

---

[23] Coydon B. 1995, *Visual Basic, Développement d'applications professionnelles sous Windows*. Eyrolles, Paris.

d'objets, d'héritage, de liaison dynamique et d'encapsulation. Bertrand Meyer insiste tout particulièrement sur la gestion automatique[24] de la mémoire. Eiffel, Smalltalk et Java sont pourvus d'un système de ramasse-miettes qui affranchit le programmeur du difficile, voire insurmontable, problème de libération de la mémoire dynamique. La principale qualité des langages objet est de réduire le décalage entre la manière de penser du programmeur et le langage fruste compris par l'ordinateur. Il existe toutefois de grandes variations syntaxiques entre les différents langages objet selon qu'ils sont une extension conservatrice comme C++ ou une innovation totale comme Smalltalk.

Le choix d'un langage de programmation objet est souvent dicté par la culture informatique de l'entreprise. Jusqu'à présent, l'informatique technique penchait plutôt pour C++ alors que le monde de la gestion manifestait un intérêt croissant pour Smalltalk ou les solutions semi-objet évoquées précédemment. L'année 96 a été marquée par le déferlement du langage Java développé par Sun : il suscite un intérêt très large de la part de l'ensemble de la communauté informatique.

Il existe de nombreux langages de programmation objet dont les principaux sont décrits ici dans leurs grandes lignes :

- Ada est un langage de programmation de haut niveau conçu pour réduire les coûts de développement exorbitants auxquels le ministère de la défense américain devait faire face dans les années 70. Ada a été normalisé une première fois en 1983 par l'ANSI. Après une décennie d'utilisation, le langage Ada a été amélioré par l'ajout de constructions objet et sa deuxième mouture, standardisée en 1995, en fait un langage objet à part entière. Ada 83 est le langage du génie logiciel ; Ada 95 est le langage du génie logiciel objet.
- C++ est un langage objet hybride conçu par Bjarne Stroustrup dans les années 80 : c'est une extension du langage C. C++ corrige le C en rajoutant le typage fort, les classes, la surcharge et la liaison dynamique. Sa syntaxe reste touffue comme celle du C et rend C++ difficile à mettre en œuvre.
- Eiffel est un langage objet volontairement simple, à la syntaxe très propre. Il a été conçu par Bertrand Meyer dans les années 80. Eiffel se distingue des autres langages objet par la place qu'il accorde à la notion de programmation par contrat ; c'est ainsi le seul langage qui prenne en compte le principe de substitution dans sa syntaxe.
- Java est le dernier-né des langages objet. Il s'inspire de C++ pour la syntaxe et de Smalltalk pour la machine virtuelle, tout en restant comme Eiffel volontairement simple. Initialement conçu pour la programmation des

[24] Meyer B. 1988, Object-Oriented Software Construction, Prentice Hall, New York, NY.

assistants électroniques (*personal digital assistant*), le langage Java a été propulsé sur le devant de la scène par sa forte complémentarité avec Internet.

- Smalltalk est le langage objet par excellence : en Smalltalk, tout est objet, y compris les classes elles-mêmes. Smalltalk a été conçu dans les années 70 au PARC (le centre de recherche de Xerox à Palo Alto). Depuis 1980, Smalltalk est accompagné d'un environnement de développement sophistiqué et d'une grande collection de classes prédéfinies. Du fait de sa souplesse d'emploi et de sa simplicité, Smalltalk connaît actuellement un regain d'intérêt pour la création de systèmes d'information.

Le tableau suivant résume les caractéristiques des principaux langages objet. Le cas de Smalltalk est particulier car les caractéristiques du langage peuvent être étendues en intervenant au niveau des métaclasses.

	Ada 95	**C++**	**Eiffel**	**Java**	**Smalltalk**
Paquetage	Oui	Non	Non	Oui	Non
Héritage	Simple	Multiple	Multiple	Simple	Simple
Généricité	Oui	Oui	Oui	Non	Non
Typage	Fort	Fort	Fort	Fort	Non typé
Liaison	Dynamique	Dynamique	Dynamique	Dynamique	Dynamique
Parallélisme	Oui	Non	Non	Oui	Non
Ramasse-miettes	Non	Non	Oui	Oui	Oui
Assertions	Non	Non	Oui	Non	Non
Persistance	Non	Non	Non	Non	Non

Figure 316 : Tableau récapitulatif des grandes caractéristiques des principaux langages objet.

La transition vers l'objet

La transition d'une organisation vers la technologie objet demande un temps non négligeable.

Pour des développeurs confirmés, il faut compter en moyenne :

- 1 mois pour maîtriser les constructions d'un langage objet,
- 6 à 9 mois pour acquérir un mode de pensée objet,
- 1 mois pour maîtriser une bibliothèque de classes,
- 6 à 9 mois pour dominer un framework.

Il faut savoir néanmoins qu'une petite minorité de réfractaires ne parviendront jamais à sauter le pas (entre 10 et 15 % des développeurs). Aucun critère simple ne permet a priori de les identifier.

Pour faciliter la transition, il est fortement conseillé :

- de faire appel à des compétences externes,
- de lancer un projet pilote,
- de mettre en place un plan de formation sous une forme qui combine théorie et pratique,
- d'avoir recours à des spécialistes du transfert de technologie qui accompagneront le projet,
- de former tout le monde, les développeurs mais aussi les groupes de test, les équipes d'assurance qualité et l'encadrement.

La transition vers l'objet est coûteuse. Il est sage de prévoir un surcoût de 10 à 20 % pour le premier projet objet du fait de l'apprentissage des différentes techniques par l'équipe de développement et de l'expérimentation de la gestion du cycle itératif par l'encadrement. Les bénéfices de la transition ne seront récoltés qu'à partir du deuxième ou troisième projet, qui devrait demander de 20 à 30 % de ressources en moins.

Vers une méthode de développement

Un processus de développement de logiciel a pour objectif la formalisation des activités liées à l'élaboration des systèmes informatiques. La formalisation d'un processus de développement vise à doter les entreprises d'un ensemble de mécanismes qui, lorsqu'ils sont appliqués systématiquement, permettent d'obtenir de manière répétitive et fiable des systèmes logiciels de qualité constante. Par nature, la description du processus reste générale car il n'est pas possible de définir autoritairement un standard unique, adapté à toutes les personnes, tous les types d'applications et toutes les cultures. Il convient plutôt de parler de cadre configurable, éventuellement raffiné de manière consensuelle par la pratique et la mise en œuvre de produits largement adoptés par la communauté des utilisateurs.

Une méthode de développement comprend :

- des éléments de modélisation qui sont les briques conceptuelles de base,
- une notation dont l'objectif est d'assurer le rendu visuel des éléments de modélisation,
- un processus qui décrit les étapes à suivre lors du développement du système,

- du savoir-faire, plus ou moins formalisé.

Watts S. Humphrey a proposé cinq niveaux de maturité[25] des processus de développement de logiciel :

- *initial* : le développement n'est pas formalisé, l'équipe réagit au jour le jour et choisit des solutions au cas par cas de sorte que le succès dépend fortement des personnes impliquées dans le projet,
- *reproductible* : l'organisation est capable d'établir des plans raisonnables en termes de budget et de vérifier l'état d'avancement du projet par rapport à ces plans,
- *défini* : le processus de développement est bien défini, connu et compris par tous les intervenants du projet,
- *encadré* : les performances du processus de développement sont mesurables objectivement,
- *optimisant* : les données de contrôle des performances du processus permettent l'amélioration du processus.

A partir de ce modèle, le SEI (*Software Engineering Institute*) de l'université de Carnegie Mellon à développé une procédure d'évaluation des processus de développement connue sous l'abréviation CMM (*Capability Maturity Model*).

Le diagramme suivant représente les cinq niveaux de maturité des processus.

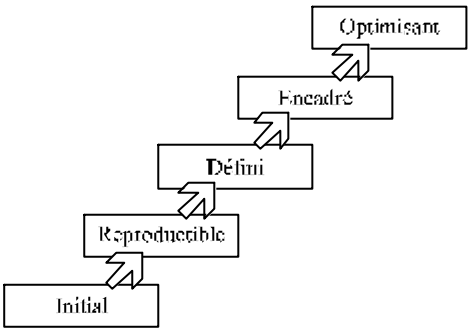


Figure 317 : Représentation des cinq niveaux de maturité des processus de développement selon Watts S. Humphrey.

La segmentation de la modélisation permet de gérer la complexité en réduisant la portée de l'étude à une partie, un sous-ensemble ou un point de vue. De cette manière, lorsque le tout est trop complexe pour être compris d'un seul coup, la compréhension globale peut se dégager par la perception parallèle de plusieurs

[25] Humphrey W. S. 1989, Managing the Software Process, Addison-Wesley.

vues disjointes, mais concourantes. Le choix des points de vue, c'est-à-dire de ce qui est modélisé, influence fortement la manière d'approcher le problème, et donc la forme des solutions qui retenues. Il n'existe pas de modèle universel et les niveaux de détail, de précision ou de fidélité peuvent varier. Les meilleurs modèles sont en prise directe sur la réalité.

UML est un langage pour la représentation des modèles objet, mais le processus d'élaboration de ces modèles n'est pas décrit par UML. Les auteurs d'UML insistent néanmoins sur les caractéristiques suivantes, qui leur semblent essentielles pour un processus de développement :

- *piloté par les cas d'utilisation* : toutes les activités, de la spécification de ce qui doit être fait jusqu'à la maintenance, sont guidées par les cas d'utilisation.
- *centré sur l'architecture*. Dès le départ, le projet place la détermination des traits de l'architecture en ligne de mire. L'architecture est conçue pour satisfaire les besoins exprimés dans les cas d'utilisation, mais aussi pour prendre en compte les évolutions et les contraintes de réalisation. L'architecture doit être simple et se comprendre de manière intuitive ; elle est conçue avec et pour la réutilisation.
- *itératif et incrémental* : l'ensemble du travail est divisé en petites itérations, définies à partir du degré d'importance des cas d'utilisation et de l'étude des risques. Le développement procède par des itérations qui conduisent à des livraisons incrémentales du système. Les itérations peuvent être conduites en parallèle.

Le diagramme suivant représente la forme générale du processus de développement préconisé par les auteurs d'UML.

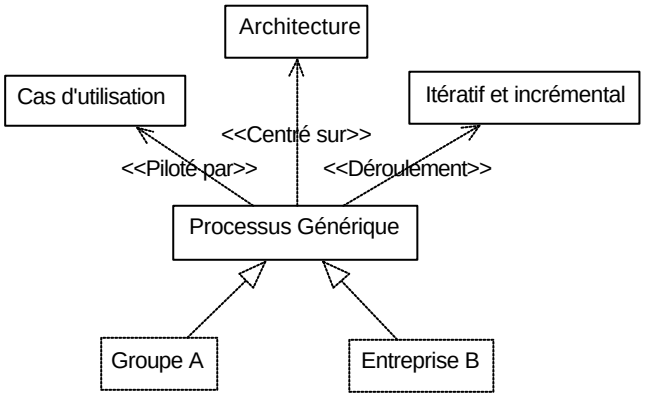


Figure 318 : Le processus de développement préconisé par les auteurs d'UML est piloté par les cas d'utilisation, centré sur l'architecture et déroulé de manière itérative et incrémentale.

Les cas d'utilisation

Les cas d'utilisation (*use cases*) viennent en complément de l'approche objet et forment la base du processus de développement. Dans ce contexte, les cas d'utilisation doivent être vus comme un fil rouge ; ce fil balise les différentes activités du développement de l'expression des besoins d'un utilisateur vers la réalisation informatique qui satisfait ces besoins.

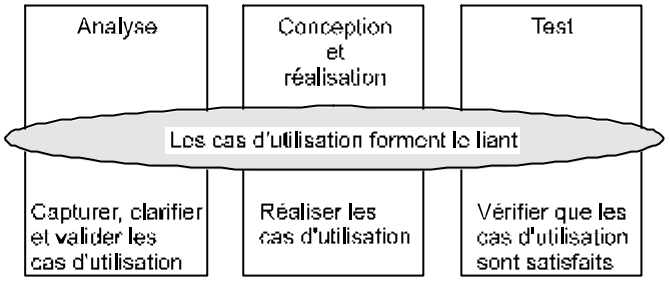


Figure 319 : Les cas d'utilisation balisent les différentes activités et étapes du processus.

Les besoins sont d'abord segmentés par rapport aux différents types d'utilisateurs (les acteurs) et décrits par des cas d'utilisation exprimés dans le langage des acteurs. A ce stade, les cas d'utilisation aident l'utilisateur à articuler ses besoins : il décrit les services qu'il attend du système sous la forme d'interactions entre l'acteur et le système. L'analyse des besoins est principalement exprimée par un modèle des cas d'utilisation.

Le processus continue par une démarche d'analyse objet. Les objets qui appartiennent fondamentalement au domaine de l'application sont identifiés, puis décrits de manière générale par leurs classes. Chaque cas d'utilisation est ensuite réalisé par une collaboration entre les objets trouvés précédemment. La décomposition n'est pas fonctionnelle : la collaboration qui réalise un cas d'utilisation n'est qu'un cas particulier des collaborations potentielles générées par la structure des classes.

Le passage de l'analyse à la conception est marqué par l'introduction de considérations de réalisation dans les modèles. L'analyse décrit le *quoi* d'un système alors que la conception s'intéresse au *comment*. Les cas d'utilisation continuent de baliser la conception puisqu'ils spécifient ce que les mécanismes doivent faire.

Les cas d'utilisation décrivent les tests fonctionnels.

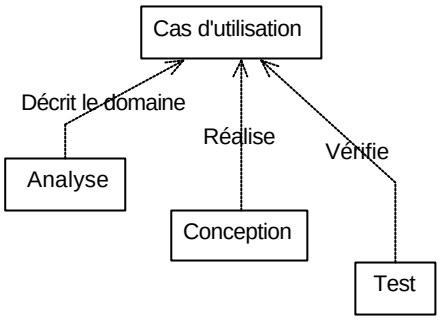


Figure 320 : Les cas d'utilisation coordonnent et balisent les différents modèles.

Pour autant, les cas d'utilisation ne donnent pas la forme du système. Dans une approche objet, la forme (l'architecture) du logiciel ne reflète pas la forme des cas d'utilisation.

Architecture logicielle

Le développement d'un logiciel peut être vu comme la traversée d'un océan en bateau. Le jour de départ est connu, le jour d'arrivée l'est moins, et en cours de route, il faudra affronter des tempêtes et réparer des avaries. La conception de l'architecture d'un logiciel a pour objet de donner à un effort de développement les moyens de parvenir à bon port.

L'architecture logicielle s'intéresse à la forme globale d'une application : cette forme émerge de la conception au-delà des structures de données ou des algorithmes[26]. L'architecture logicielle se préoccupe d'intégrité, d'uniformité, de simplicité et d'esthétisme. Elle est presque à contre-courant d'une discipline (le développement de logiciel) que rien ne prédispose aux vertus évoquées plus haut, du fait principalement de l'immatérialité du logiciel et de l'absence de contraintes physiques.

La mise en place d'une architecture adaptée est la clé de voûte du succès d'un développement. Il importe de la stabiliser le plus tôt possible. Il n'existe pas d'architecture universelle, adaptée à tous les types d'applications. En revanche, il existe des architectures réutilisables, dans les limites d'un domaine particulier.

---

[26] Garland D., Shaw, M. 1993, *An Introduction to Software Architecture*. Advances in Software Engineering and Knowledge Engineering, Vol. 1, Singapore, World Scientific Publishing Co., pp. 1-39

Perry et Wolf[27] définissent l'architecture des logiciels par la formule suivante :

Architecture logicielle = Eléments + Formes + Motivations

Dans le contexte de la démarche objet, les éléments sont les objets et les classes, les formes sont des regroupements de classes et d'objets (les *patterns*), et les motivations expliquent pourquoi tel ou tel regroupement est plus adapté qu'un autre dans un contexte donné.

L'architecture offre une vision globale de l'application ; elle décrit les choix stratégiques qui déterminent les qualités du logiciel, comme la fiabilité, l'adaptabilité ou la garantie des performances, tout en ménageant des espaces pour les décisions tactiques prises en cours de développement[28].

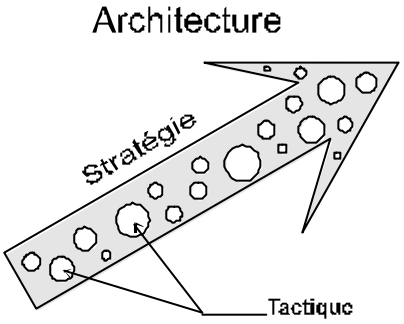


Figure 321 : L'architecture reflète les choix stratégiques généraux et ménage des espaces pour les décisions tactiques ponctuelles.

Les bonnes architectures se caractérisent par :

- la simplicité,
- l'élégance,
- l'intelligibilité,
- des niveaux d'abstraction bien définis,

---

[27] Perry D. E.and Wolf A. L. Oct. 1992, *Foundations for the Study of Software Architecture*. ACM Software Eng. Notes, pp. 40-52.

[28] Pour prendre une image gastronomique, le développement consiste à transformer un morceau de gruyère en un morceau de comté, par remplissage des trous !

- une séparation claire entre l'interface et l'implémentation de chaque niveau.

La vision de l'architecte

Il n'existe pas une seule manière de considérer un système. De multiples points de vue sont requis. Par ailleurs, pour satisfaire les multiples intervenants chaque type de diagramme ne donne qu'une image partielle d'un système.

Tous les projets objet couronnés de succès se caractérisent par la présence d'une forte vision de l'architecture. Cette vision peut être décrite à partir de plusieurs vues complémentaires, inspirées de celles décrites par Philippe Kruchten[29], dans son modèle dit des *4 + 1 vues*.

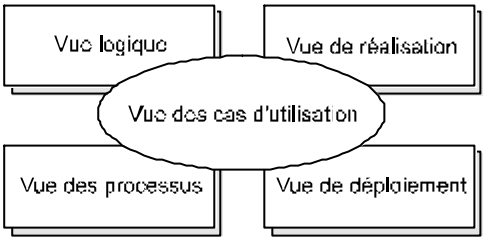


Figure 322 : Représentation du modèle d'architecture de Philippe Kruchten.

Ce modèle d'architecture est bien adapté à la représentation des contraintes variées que l'architecte doit prendre en considération.

La vue logique

La vue logique décrit les aspects statiques et dynamiques d'un système en termes de classes et d'objets et se concentre sur l'abstraction, l'encapsulation et l'uniformité. Le système est décomposé dans un jeu d'abstractions-clés, originellement issues du domaine du problème. Au-delà de la satisfaction des besoins fonctionnels de l'utilisateur, la vue logique permet d'identifier et de généraliser les éléments et les mécanismes qui constituent les différentes parties du système.

La vue logique met en jeu les éléments de modélisation suivants :

- les objets,
- les classes,
- les collaborations,
- les interactions,

---

[29] Kruchten P. Nov. 95, *The 4+1 View Model of Architecture*. IEEE Software.

- les catégories (paquetages stéréotypés).

La vue de réalisation

La vue de réalisation se préoccupe de l'organisation des modules dans l'environnement de développement. Elle montre l'allocation des classes dans les modules, et l'allocation des modules dans les sous-systèmes. Les sous-systèmes sont eux-mêmes organisés en niveaux hiérarchiques aux interfaces bien définies.

La vue de réalisation traite des éléments de modélisation suivants :

- les modules,
- les sous-programmes,
- les tâches (en tant qu'unités de programme, comme en Ada),
- les sous-systèmes (paquetages stéréotypés).

La vue des processus

La vue des processus représente la décomposition en flots d'exécution (processus, threads, tâches...), la synchronisation entre flots et l'allocation des objets et des classes au sein des différents flots. La vue des processus se préoccupe également de la disponibilité du système, de la fiabilité des applications et des performances.

La vue des processus prend toute son importance dans les environnements multitâches.

La vue des processus manipule les éléments de modélisation suivants :

- les tâches, les threads, les processus,
- les interactions.

La vue de déploiement

La vue de déploiement décrit les différentes ressources matérielles et l'implantation du logiciel dans ces ressources.

La vue de déploiement traite les points suivants :

- les temps de réponse et les performances du système,
- la bande passante des chemins de communication et les capacités,
- les contraintes géographiques (disposition physique des processeurs),
- les besoins en puissance de calcul distribuée,
- les surcharges et l'équilibrage des charges dans un système distribué,
- la tolérance aux fautes et aux pannes.

La vue de déploiement prend toute son importance lorsque le système est distribué.

La vue de déploiement se concentre sur les éléments de modélisation suivants :

- les nœuds,
- les modules,
- les programmes principaux.

La vue des cas d'utilisation

Les cas d'utilisation forment la colle qui unifie les quatre vues précédentes. Les cas d'utilisation motivent et justifient les choix d'architecture. Ils permettent d'identifier les interfaces critiques, ils forcent les concepteurs à se concentrer sur les problèmes concrets, ils démontrent et valident les autres vues d'architecture.

La vue des cas d'utilisation rend compte des éléments de modélisation suivants :

- les acteurs,
- les cas d'utilisation,
- les classes,
- les collaborations.

Organisation des modèles

Les cinq vues décrites plus haut, associées au concept de paquetage, permettent la structuration hiérarchique d'un modèle. Un modèle décrit par UML prend la forme de l'arborescence représentée dans le diagramme de classes suivant :

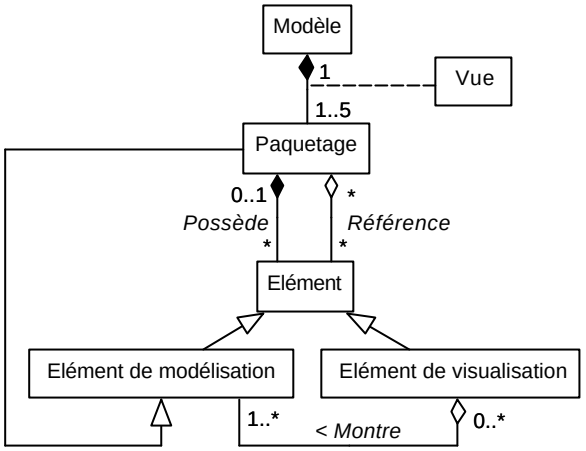


Figure 323 : Extrait du métamodèle. Organisation hiérarchique des modèles selon plusieurs vues.

Les paquetages sont à la fois des éléments de modélisation et des éléments de structuration des modèles. L'arborescence des paquetages organise les modèles, comme les répertoires organisent les systèmes de fichiers.

Le diagramme d'objets suivant donne un exemple d'organisation d'un modèle, structuré selon les cinq vues d'architecture présentées plus haut :

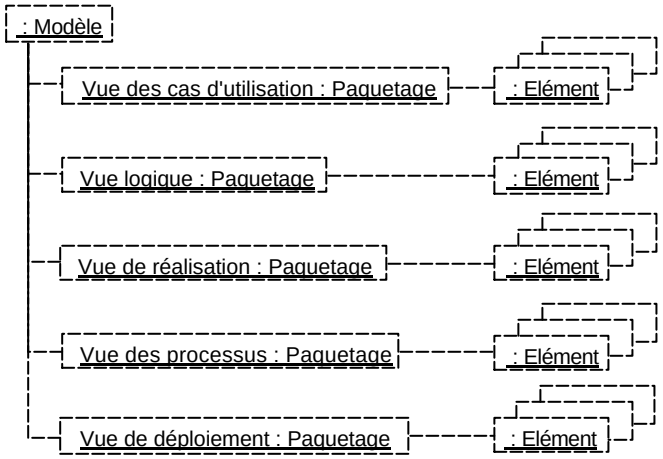


Figure 324 : Exemple de modèle structuré selon cinq vues d'architecture.

Documentation de l'architecture

L'architecture est décrite dans un document d'architecture qui comprend :

- une description textuelle des traits de l'architecture (les vues) et les besoins-clés du système,
- les compromis retenus et les alternatives explorées,
- une représentation de haut niveau de la vue logique (entre 5 et 50 classes-clés),
- des scénarios spécifiques à l'architecture,
- la description des mécanismes-clés.

Articulation des différents diagrammes

Les éléments de modélisation sont accessibles à l'utilisateur par l'intermédiaire de projections graphiques (les éléments de visualisation) représentées dans les neufs types de diagrammes suivants :

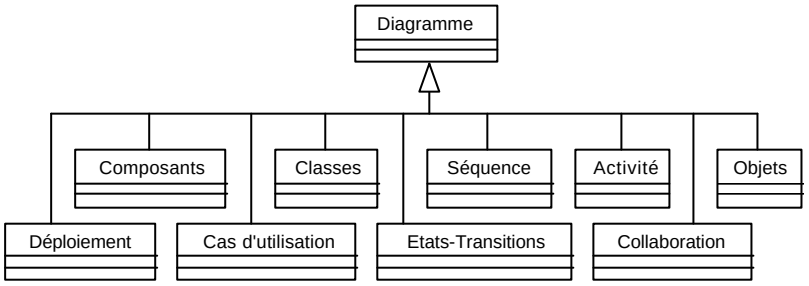


Figure 325 : Représentation des neuf types de diagrammes définis par UML.

Ces diagrammes forment la base des vues d'architecture décrites plus haut. Il importe de bien conceptualiser les liens entre les différents éléments de modélisation et leurs représentations associées afin de documenter un système avec clarté et efficacité.

Un modèle est une construction sémantique, organisée sous la forme d'un réseau. Ce réseau peut se parcourir en suivant les règles générales décrites dans le métamodèle d'UML.

Les différents diagrammes peuvent être vus comme des chemins graphiques parcourant les modèles. Les paragraphes suivants proposent un exemple de cheminement, depuis l'énoncé des besoins jusqu'au déploiement dans le matériel.

Expression des besoins

L'expression des besoins d'une catégorie d'utilisateur est décrite sous une forme fonctionnelle dans les cas d'utilisation. Les cas d'utilisation sont exprimés en langage naturel, dans les termes de l'utilisateur.

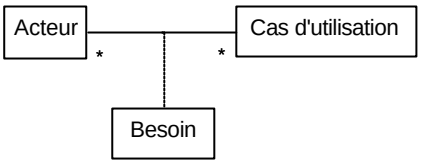


Figure 326 : Les besoins des utilisateurs sont exprimés, acteur par acteur, sous la forme de cas d'utilisation.

Transition vers l'objet

La transition vers l'objet est opérée par l'analyste. Il réalise le comportement décrit dans le cas d'utilisation au moyen d'une collaboration entre des objets, initialement issus du domaine de l'application. Le contexte d'une collaboration est représenté dans un diagramme d'objets.

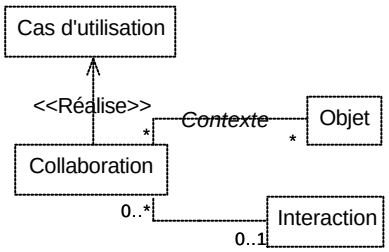


Figure 327 : La transition vers l'objet est opérée en réalisant un cas d'utilisation par une collaboration entre objets du domaine.

Expression du comportement

Le comportement d'une collaboration ou d'une classe peut se représenter sous la forme d'une interaction ou d'un automate. Les interactions sont visualisées du point de vue temporel, par des diagrammes de séquence et du point de vue spatial, par des diagrammes de collaboration. Les automates sont visualisés du point de vue des états et des transitions, par des diagrammes d'états-transitions et du point de vue des activités, par des diagrammes d'activités.

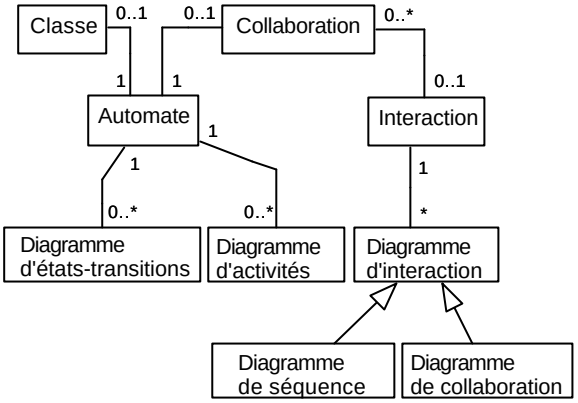


Figure 328 : UML propose 5 types de diagrammes pour la représentation du comportement.

Représentation de la structure

Chaque objet est instance d'une classe. Les classes du domaine décrivent de manière générale les objets métiers. Les objets sont représentés dans les diagrammes d'objets, dans les diagrammes de collaboration et dans les diagrammes de séquence. Les classes sont représentées dans les diagrammes de classes.

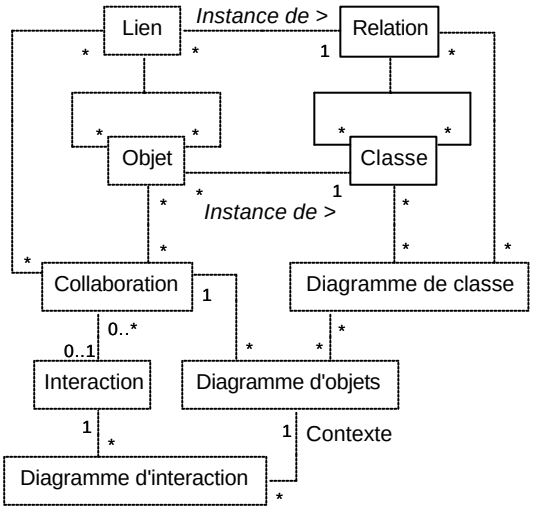


Figure 329 : Chaque objet est instance d'une classe.

Réalisation des objets et des classes

Le code qui réalise les classes, les objets et les interactions entre ces objets est stocké dans des composants qui constituent les briques de la structure physique des systèmes. Les composants regroupent les modules, les sous-programmes, les tâches (au sens des unités de programmes Ada) et les programmes principaux.

Les modules contiennent le code qui correspond aux classes et aux objets, les programmes principaux contiennent le code qui correspond aux points d'entrée dans la réalisation des interactions. Un module peut contenir plusieurs classes et plusieurs objets, mais par défaut, il n'y a qu'une seule classe par module ; le nom de la classe est utilisé pour construire le nom du module dans l'espace de développement.

Un diagramme de composants montre les dépendances de compilation entre les différents composants d'une application.

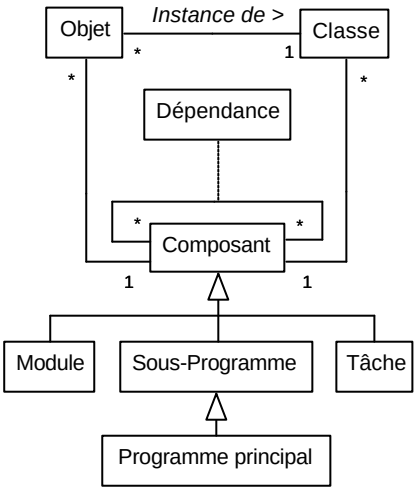


Figure 330 : Le code qui correspond aux objets et aux classes est stocké dans des modules.

Déploiement du code exécutable

Un programme principal est une sorte de composant qui contient un code exécutable correspondant à une ou plusieurs interactions. Très souvent, un programme principal ne correspond qu'à une seule interaction. Un programme principal est déployé sur des nœuds, au sein de processus qui l'exécutent.

Les diagrammes de déploiement montrent les différentes sortes de processeurs et de dispositifs qui composent l'environnement d'exécution du système ; ils montrent aussi la configuration de composants, de processus et d'objets qui

s'exécutent dans cet environnement. Si nécessaire, la migration des objets et des processus est représentée au moyen d'une relation de dépendance stéréotypée.

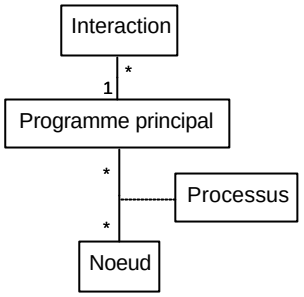


Figure 331 : Un programme principal est une sorte de composant qui contient un code exécutable correspondant à une ou plusieurs interactions.

Organisation des modèles et des vues

Les paquetages organisent les vues et les modèles en encapsulant les autres éléments de modélisation dans une décomposition hiérarchique, analogue à la structuration d'un système de fichiers par des répertoires :

- dans la vue logique, les paquetages contiennent des objets, des classes, d'autres paquetages, et les diagrammes qui visualisent ces éléments,
- dans la vue de réalisation, les paquetages contiennent des composants et d'autres paquetages et les diagrammes correspondants.

Les paquetages peuvent être spécialisés sous la forme de catégories et de sous-systèmes, pour distinguer respectivement les paquetages de la vue logique et les paquetages de la vue de réalisation.

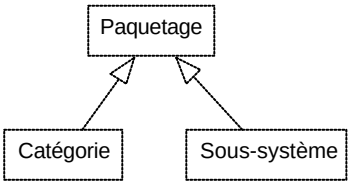


Figure 332 : Exemple de spécialisation des paquetages pour distinguer les concepts d'organisation de la vue logique, de ceux de la vue de réalisation.

En règle générale, il y a correspondance directe entre une catégorie et un sous-système. Les hiérarchies de catégories et de sous-systèmes peuvent différer, entre autres pour :

- regrouper des objets qui ont une forte interaction,
- réaliser des fonctionnalités de bas niveau, non représentées en analyse,
- répartir le travail dans l'équipe de développement.

Granularité des éléments de modélisation

Les éléments de modélisation d'UML suivent une dichotomie telle que pour un type de concept donné, il existe toujours un élément de modélisation à granularité fine et un élément de modélisation à granularité plus élevée.

Le diagramme suivant illustre les correspondances entre les éléments à granularité forte.

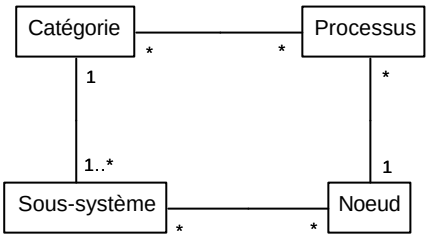


Figure 333 : Représentation des correspondances entre les éléments à granularité forte.

Les éléments à plus faible granularité sont également en correspondance.

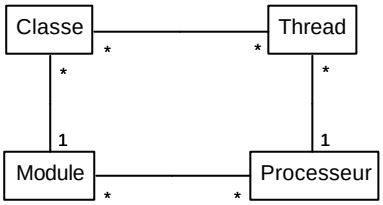


Figure 334 : Représentation des correspondances entre les éléments à granularité fine.

Vue par vue, les éléments à forte granularité sont des agrégats d'éléments à granularité plus fine. UML permet de construire des modèles empreints de cette décomposition hiérarchique, et favorise ainsi la navigation entre les différents points de vue et entre les niveaux de détails de ces points de vue.

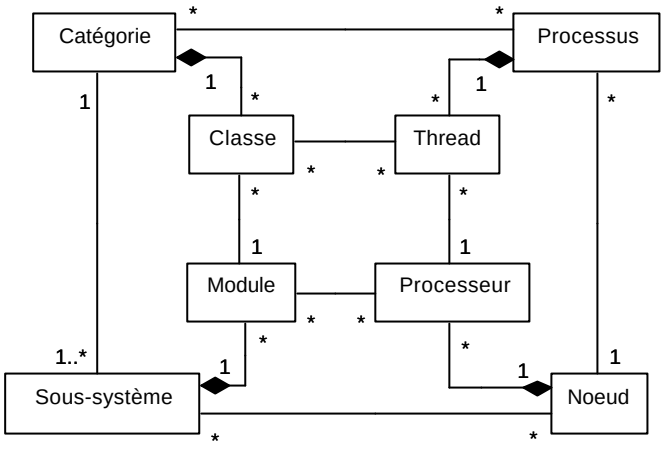


Figure 335 : Correspondance entre les éléments de modélisation au sein d'une représentation hiérarchique matérialisée par les agrégations.

Cette dichotomie entre éléments peut être représentée sous la forme d'un empilement pyramidal des éléments de modélisation d'UML ; il montre que tous les éléments sont visibles simultanément en cas de besoin, mais que la vision peut également se limiter à un niveau d'abstraction donné.

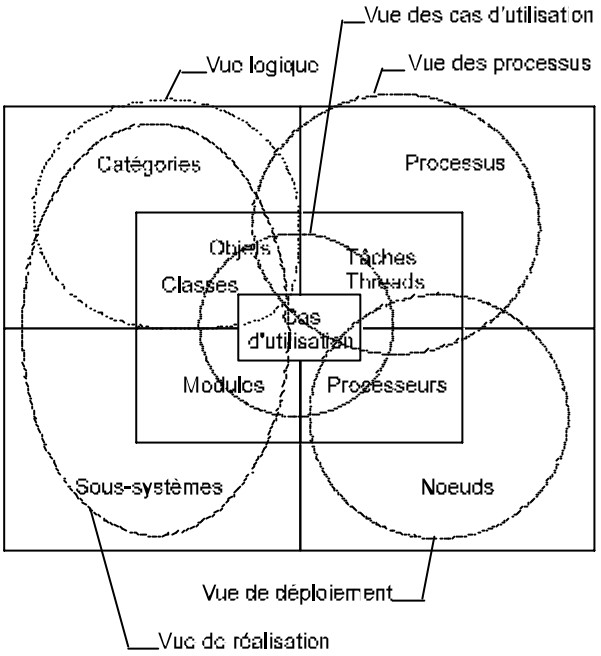


Figure 336 : Représentation pyramidale des éléments de modélisation d'UML.

La figure précédente fournit une vue du dessus de cet empilement. Les éléments de même niveau hiérarchique dans la pyramide sont en correspondance vue par vue. La base de la pyramide regroupe les éléments à forte granularité, l'étage intermédiaire contient les éléments à fine granularité et le sommet est constitué des cas d'utilisation. Ces derniers se réalisent par des collaborations construites à partir des éléments des niveaux inférieurs.

Grâce à ces correspondances entre éléments de modélisation, UML facilite la navigation entre les points de vue généraux et les points de vue détaillés, comme le décrit Grady Booch, pour qui la forme générale influence les formes détaillées et réciproquement. Il qualifie sa démarche de *Round Trip Gestalt*[30].

Récapitulatif des éléments et des vues

Le tableau suivant résume l'utilisation courante des éléments dans les différents diagrammes dans le but de représenter les différents points de vue. Cette représentation n'est pas limitative ; l'utilisateur est libre de représenter les éléments qu'il trouve pertinents dans les diagrammes qui lui conviennent.

---

[30] Booch G. 1994, *Object-Oriented Analysis and Design, with Applications.* Benjamin Cummings.

	Vue des cas d'utilisation	**Vue logique**	**Vue de réalisation**	**Vue des processus**	**Vue de déploiement**
Diagramme de cas d'utilisation	Acteurs Cas d'utilisation				
Diagramme de classes		Classes Relations			
Diagramme d'objets	Objets Liens	Classes Objets Liens			
Diagramme de séquence	Acteurs Objets Messages	Acteurs Objets Messages		Objets Messages	
Diagramme de collaboration	Acteurs Objets Liens Message	Acteurs Objets Liens Messages		Objets Liens Messages	
Diagramme d'états-transitions	Etats Transitions	Etats Transitions		Etats Transitions	
Diagramme d'activité	Activités Transitions	Activités Transitions		Activités Transitions	
Diagramme de composants			Composants	Composants	Composants
Diagramme de déploiement					Nœuds Liens

Figure 337 : Tableau récapitulatif de l'utilisation des éléments dans les différents types de diagrammes dans le but de représenter les différents points de vue.

Micro-architecture

Le terme anglais *pattern*, en français forme, micro-architecture, ou schème, désigne des combinaisons récurrentes d'objets et de classes. Elle se retrouvent fréquemment dans les conceptions objet et peuvent être qualifiées de savoir-faire objet. Le principal intérêt des patterns est de permettre la manipulation de

concepts d'architecture plus élaborés que les objets et classes. Ils augmentent la puissance d'expression des langages de modélisation. Cette démarche évite également de trop s'attarder sur des objets liés aux particularités du domaine de l'application, et ainsi de rechercher des conceptions de plus haut niveau, éventuellement moins performantes, mais plus flexibles et plus extensibles. Dans une certaine mesure, un pattern est pour les objets, l'équivalent du sous-programme pour les instructions, ou encore des circuits intégrés pour les transistors. Le pattern est l'unité de réutilisation de conception objet.

Le concept de pattern[31] a été décrit par Christopher Alexander[32]. Cet architecte désirait formaliser les grands traits de l'architecture afin de faciliter la construction des bâtiments. Cette idée de formalisation des éléments d'architecture, ou encore des formes d'architecture (d'où le terme de pattern), peut tout à fait être transposée dans le domaine du logiciel, afin de définir des formes logicielles réutilisables d'applications en applications.

UML représente les patterns par les collaborations. Une collaboration regroupe un contexte d'objets et une interaction entre ces objets. Une collaboration convient donc pour la représentation des patterns de structure, comme des patterns de comportement. Les collaborations sont représentées par des ellipses pointillées, connectées aux objets qui y participent.

Le diagramme suivant représente un pattern de chaîne de responsabilité.

---

[31] Alexander C., Ishikawa S., Silverstein M., Jacobson M., Fiskdahl-King I., Angel S. 1977, *A Pattern Language*. Oxford University Press, New York.

[32] « *Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.* » Christopher Alexander

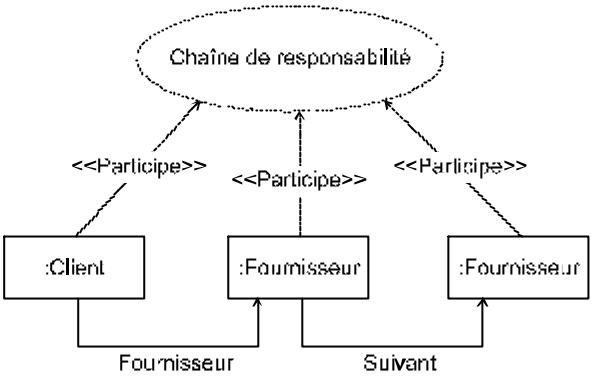


Figure 338 : Représentation d'un pattern ***chaîne de responsabilité*** *par une collaboration.*

Il existe également des patterns d'analyse, construits à partir d'objets métiers, et dédiés spécifiquement à des logiciels déployés dans un domaine donné.

Description des patterns

La description[33] précise des patterns est essentielle pour permettre leur mise en œuvre. Au minimum, un pattern est décrit par :

- un nom pour augmenter le niveau d'abstraction du langage de modélisation,
- un problème avec son contexte et des pré- et post-conditions,
- une solution abstraite indépendante des langages de programmation,
- des conséquences en termes de flexibilité, d'extensibilité et de portabilité.

Idiomes

Les patterns sont des constructions caractéristiques totalement indépendantes des langages de réalisation. Les patterns formalisent des choix d'architecture qui peuvent être mis en œuvre de manière uniforme, quel que soit l'environnement de réalisation. Parallèlement à ces formes indépendantes, il existe des constructions récurrentes spécifiques à un langage de programmation donné. Ces formes particulières, appelées idiomes, regroupent des constructions syntaxiques existantes. Les idiomes permettent de former de nouvelles constructions et ne sont généralement pas transposables tels quels d'un langage à l'autre.

---

[33] Gamma E., Helm R., Johnson R., Vlissides J. 1995, *Elements of Reusable Object-Oriented Software*. Addison-Wesley.

Les idiomes appartiennent au florilège de trucs et astuces qui accompagnent un style de programmation, considéré comme élégant dans un langage donné. Selon la puissance d'expression des langages, une technique donnée peut être un idiome dans un langage et une construction syntaxique dans un autre.

La copie d'une chaîne de caractères en C++ peut se réaliser de la manière suivante :

```
While (*destination++ = *source++) ;
```

En Ada, les noms peuvent être préfixés :

```
Standard.Mailbox.Open ;
```

En C++, beaucoup de techniques de programmation – comme la gestion de la mémoire, les entrées-sorties ou l'initialisation – relèvent des idiomes. Ces idiomes sont décrits dans des ouvrages[34] qui se présentent comme des manuels de programmation avancée. A l'usage, C++ se révèle être un langage assez difficile à utiliser, du fait de la nécessité de maîtriser un grand nombre d'idiomes pour programmer efficacement.

Macro-architecture

La représentation des grands systèmes fait appel à de nombreuses classes. Du fait de la complexité des modèles – dont le contenu dépasse rapidement les limites[35] de compréhension de l'être humain – il est impossible de distinguer les traits de l'architecture dans les entrelacs de classes et de relations. L'architecture générale d'une application ne peut pas se représenter à partir d'éléments à fine granularité ; il faut introduire des concepts structurants plus larges pour faire émerger la macrostructure et pour faciliter la compréhension des modèles.

Les patterns décrivent de petites formes récurrentes, utilisables au jour le jour pour résoudre des problèmes ponctuels. Ces patterns n'expriment pas la forme générale d'une application. C'est pourquoi des représentations plus larges regroupent les patterns au sein de *frameworks*, véritables infrastructures réutilisables, dédiées à la réalisation d'applications ciblées. Ces frameworks

---

[34] Meyers S. 1991, *Effective C++, 50 Specific Ways to Improve Your Programs and Designs*. Addison Wesley.

[35] Miller G. March 1956, *The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information*. The Psychological Review V. 63 (2).

peuvent être représentés au moyen des trois concepts structurants proposés par UML : les cas d'utilisation, les paquetages et les collaborations. Ces concepts expriment respectivement, le comportement vu de l'extérieur, la macrostructure statique et les patterns.

La macro-architecture se représente au moyen de catégories (paquetages stéréotypés pour insister sur leur fonction architecturale). Les catégories encapsulent les classes et les objets couplés logiquement au sein d'éléments plus larges. Ils permettent ainsi de construire des niveaux hiérarchiques. Ils segmentent une application en couches d'abstraction croissante. Chaque niveau possède des interfaces bien définies. Un niveau de rang donné ne dépend que des niveaux de rang inférieur. La forme générale d'une application se représente par des diagrammes de catégories, en fait des diagrammes de classes qui ne contiennent que des catégories. Les différentes catégories sont reliées entre elles par des relations de dépendances stéréotypées pour représenter les imports entre catégories.

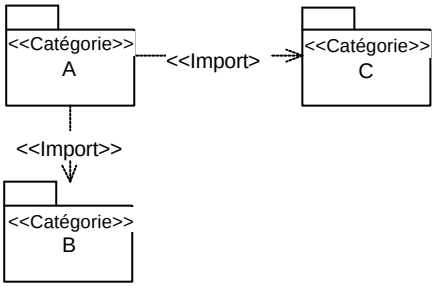


Figure 339 : Exemple de représentation des catégories et des imports entre catégories.

Le diagramme de catégories suivant représente un exemple de macrostructure. Le stéréotype **<<Framework>>** a été défini afin de désigner des ensembles de classes stabilisées, réutilisables d'applications en applications. L'application est composée de quatre niveaux d'abstraction. Le niveau le plus bas encapsule les spécificités du système d'exploitation. Le deuxième niveau offre des services de communication et de persistance. Le troisième niveau comprend un framework de contrôle-commande et deux catégories d'objets métiers : les machines-outils et les gammes d'usinage. Le dernier niveau contient un framework d'objets graphiques réutilisables et une catégorie IHM qui contient des objets miroirs des objets métiers. Les trois catégories de plus bas niveau pourraient très certainement être transformées en un framework.

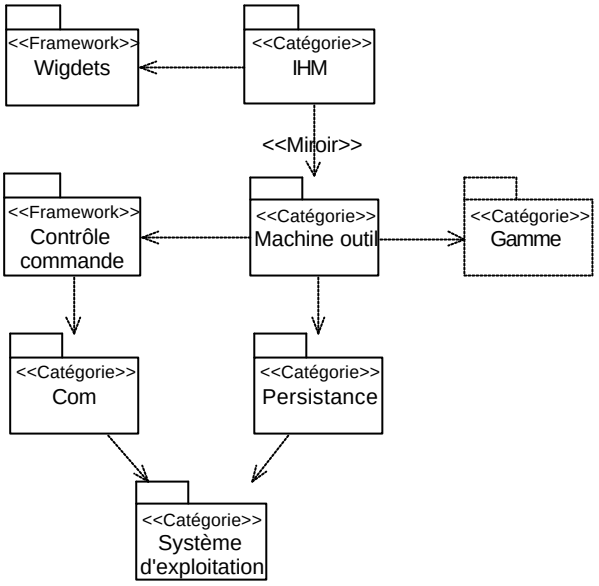


Figure 340 : Exemple de diagramme de catégories pour la représentation de la structure générale d'une application.

La place du logiciel ou le choc des cultures

Selon la part de matériel spécifique construit par l'entreprise, une distinction peut être opérée entre l'architecte du système et l'architecte du logiciel. L'architecte du système est responsable pour l'ensemble du système, le matériel, le logiciel et l'utilisation. L'architecte du logiciel ne se consacre qu'à la partie logicielle.

Cette distinction est logique, mais peut se révéler malheureuse lorsque l'architecte principal ne connaît pas assez le logiciel. Un bon architecte doit connaître à la fois le domaine de l'application et les différentes techniques employées dans la réalisation, dont le logiciel.

Dans les quinze dernières années, de nombreuses entreprises se sont trouvées confrontées à une profonde mutation de leur activité du fait de l'introduction de l'informatique dans leurs produits. Souvent, le choix de l'architecte dépend plus de sa connaissance du domaine que de sa connaissance de l'informatique. Cette situation est particulièrement sensible dans les entreprises qui ont bâti une culture électromécanique, puis muté vers l'électronique ; elles perçoivent souvent l'informatique comme un accessoire de plus et non comme le lien intégrateur qu'elle devrait être.

Les diagrammes suivants décrivent la situation. A l'origine, la part de matériel était prédominante ; le métier de l'entreprise était essentiellement exprimé au travers de ce matériel.

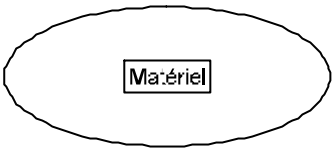


Figure 341 : A l'origine, le savoir-faire de l'entreprise est exprimé par du matériel.

Petit à petit, des éléments logiciels ont été rajoutés pour contrôler, commander ou réguler. Ces premiers programmes ont d'abord été développés en assembleur ; ils restent très proches du matériel.

Cette topologie d'application est la conséquence d'un manque de vision globale. Elle se traduit par une absence totale d'intégration entre les composants logiciels et par une duplication de l'effort, allant parfois jusqu'à des redondances matérielles. Dans le domaine de l'automobile, les systèmes distincts d'anti-patinage et d'anti-blocage des roues illustrent bien ce phénomène.

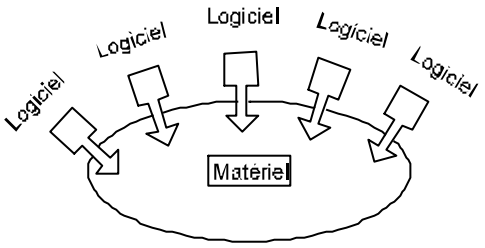


Figure 342 : Des éléments logiciels sont rajoutés dans le système, sans concertation.

Le choc culturel a lieu lorsque la tendance s'inverse au profit du logiciel, parfois par nécessité économique. Il est licite de parler d'architecture logicielle à partir du moment où une démarche raisonnée, unificatrice et intégrante regroupe le logiciel et le matériel au sein d'un cadre coopérant.

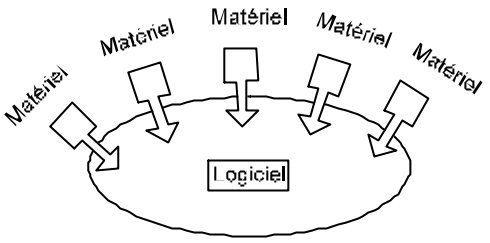


Figure 343 : Le choc culturel correspond à l'inversion du cadre intégrant.

Lorsque la mutation est opérée totalement, le savoir-faire de l'entreprise est complètement transféré dans le logiciel. Le matériel spécifique n'existe plus, ou alors est intégré dans l'architecture générale du logiciel. Cette mutation a déjà été vécue par des entreprises de CAO ou d'imagerie médicale.

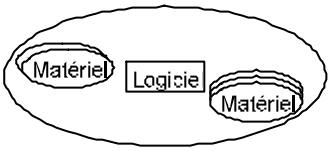


Figure 344 : Après mutation, le savoir-faire de l'entreprise est exprimé dans du logiciel.

L'informatique est un catalyseur. L'informatique s'est lovée dans les matériels, tant et tant que l'informatique a cessé d'être secondaire pour devenir principale. Ceci ne se fait pas toujours sans douleur du fait du profond choc culturel entraîné par cette mutation des métiers. Les personnes de l'entreprise qui détenaient traditionnellement le savoir du métier se sentent dépossédées ; elles réagissent, inconsciemment, en freinant des deux pieds et en essayant de confiner l'informatique dans des tâches accessoires. Ce problème est difficile à gérer car souvent les personnes qui freinent le plus fort ont suffisamment de pouvoir de décision pour imposer leur vision.

Cette attitude conservatrice engendre un déséquilibre dans l'architecture au profit du matériel : le logiciel est alors sous-employé. Un architecte qui ne possède pas de vision logicielle conçoit des systèmes dans lesquels le logiciel est mal intégré avec le matériel, au lieu de concevoir des systèmes dans lesquels le matériel est bien intégré dans le logiciel. Au-delà des habitudes, le problème est lié à une mauvaise analyse. Les solutions proposées ne sont pas assez innovantes et se contentent souvent d'informatiser l'existant.

Cette situation n'est pas réservée à l'informatique technique et se manifeste également dans le monde des systèmes d'information des entreprises. Les processus en place dans l'entreprise sont l'équivalent des éléments matériels spécifiques décrits précédemment ; le savoir-faire de l'entreprise est traduit dans

ces processus. L'approche conservatrice attend – et admet – de l'informatique l'automatisation de certaines tâches, mais seulement au sein de processus déjà en place.

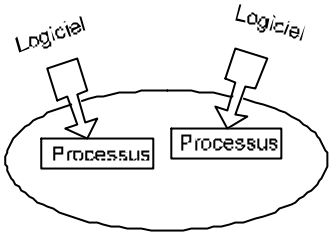


Figure 345 : Informatisation limitée dans le cadre d'un processus en place dans l'entreprise.

La rupture est consommée lorsque l'informatisation dépasse le cadre d'un processus donné et recherche l'intégration entre plusieurs processus, par exemple pour échanger de l'information. Dans les faits, cela se traduit généralement par la détection d'incohérences, de pertes de temps, de mouvements inutiles. Le choc culturel résultant est souvent plus intense que dans le monde technique. Les conséquences d'une remise à plat des processus d'une entreprise touchent plus directement les personnes et leur emploi.

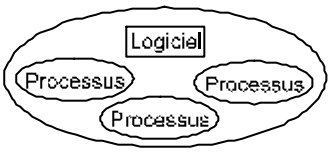


Figure 346 : Intégration des processus d'une entreprise dans un cadre informatique global.

Cycle de vie itératif et incrémental

Le cycle de vie itératif repose sur une idée très simple : lorsqu'un système est trop complexe pour être compris, conçu ou réalisé du premier coup, voire les trois à la fois, il vaut mieux le réaliser en plusieurs fois par évolutions. Dans la nature, les systèmes complexes qui fonctionnent sont toujours des évolutions de systèmes plus simples[36].

La mise en œuvre de cette idée n'est pas aussi simple que cela dans le monde de l'informatique : le logiciel ne se prête pas spontanément à l'évolution. Au

[36] Gall, J. 1986, *Systematics : How Systems Really Work and How They Fail*. 2nd ed. Ann Arbor, MI : The General Systematics Press.

contraire, le logiciel se révèle souvent extrêmement fragile en face d'un changement ou d'une modification. Ceci est directement lié à la forme interne des programmes et au couplage qui existe entre les différents constituants. L'effet d'une modification locale peut se propager dans l'ensemble de l'application et, à la limite, le basculement d'un seul bit suffit pour complètement faire s'écrouler une application qui fonctionnait correctement auparavant.

Avant de parler de développement par évolution, il faut s'attacher à rendre les logiciels plus stables et moins enclins à l'effondrement face à l'évolution. Le génie logiciel nous apprend que pour rendre un programme plus robuste, il faut segmenter l'espace des états possibles, réduire le couplage au moyen de niveaux d'abstraction et séparer les spécifications des réalisations.

L'approche objet est bâtie autour de concepts comme l'encapsulation et la modularité qui encouragent un style de programmation défensif : par nature, les programmes objet présentent une résistance plus grande face aux changements et aux surprises. L'approche objet, de ce point de vue, favorise le développement de programmes selon une démarche itérative.

Il faut être bien conscient que l'approche objet n'est pas nécessaire pour mettre en œuvre un développement itératif ; elle n'entraîne et ne requiert aucun développement itératif, mais simplement les facilite.

En résumé, l'approche objet nous fournit un cadre confortable pour développer par évolution, de manière itérative.

Cycle de vie linéaire

L'expression *cycle de vie linéaire* désigne une démarche de développement de logiciels, basée sur une succession d'étapes, depuis le cahier des charges jusqu'à la réalisation.

Le modèle en tunnel

Le modèle de développement en tunnel[37] est en fait une manière imagée de désigner l'absence de modèle de développement. Dans les projets qui suivent la démarche en tunnel, il est globalement impossible de savoir ce qui se passe. Le développement est en cours, les gens travaillent – souvent beaucoup – mais aucune information fiable n'est disponible, ni sur l'état d'avancement du logiciel, ni sur la qualité des éléments déjà développés. Ce genre de modèle de développement n'est adapté que pour les petits efforts, avec un nombre très limité de participants.

---

[37] Pierre D., communication privée.

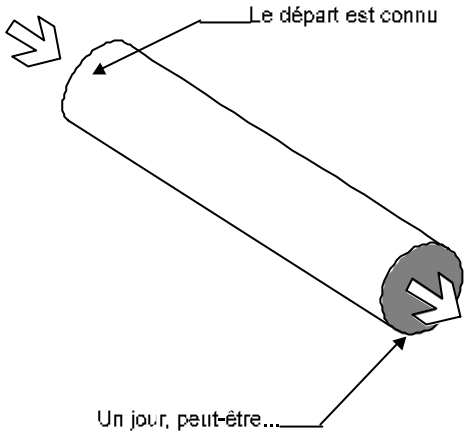


Figure 347 : Modèle de développement en tunnel, le processus de développement n'est pas visible.

Le modèle en cascade

Le cycle de vie en cascade[38], décrit par Royce dès 1970, a été largement employé depuis, pour la description générale des activités liées au développement de logiciels.

Le cycle de vie en cascade présente le développement de logiciel comme une suite de phases qui s'enchaînent dans un déroulement linéaire, depuis l'analyse des besoins jusqu'à la livraison du produit au client. Chaque phase correspond à une activité.

Le développement en cascade est rythmé par la génération de documents qui servent de support tangible pour les revues de validation du passage d'une phase à une autre.

Par rapport au modèle en tunnel, le modèle en cascade présente l'énorme avantage de fournir des points de mesure concrets, sous la forme de documents ; il augmente ainsi la visibilité sur l'état d'avancement du système en cours de développement. Le diagramme suivants illustre le gain de visibilité apporté par le modèle en cascade.

[38] Royce, W. W. Août 1970, *Managing the development of large software systems*. Proceedings of IEEE WESCON.

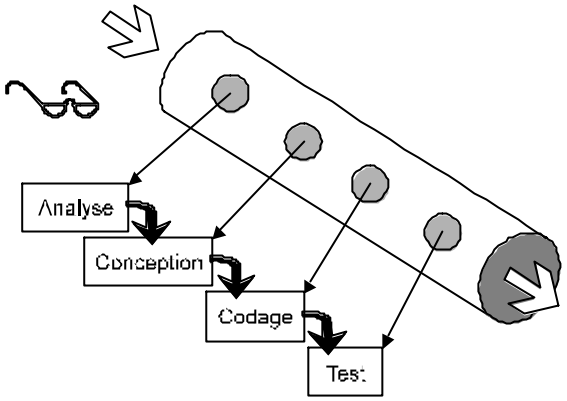


Figure 348 : Le modèle de développement en cascade ouvre des points de visibilité sur le processus de développement.

Le cycle en cascade est souvent représenté sous la forme d'une lettre V pour faire apparaître que le développement des tests est effectué de manière synchrone avec le développement du logiciel. Cette approche permet de tester ce qui devait être fait et non ce qui a été fait. Le diagramme suivant montre un exemple de modèle en V, dans lequel les tests fonctionnels sont spécifiés lors de l'analyse, les tests d'intégration lors de la conception et les tests unitaires pendant la phase de codage.

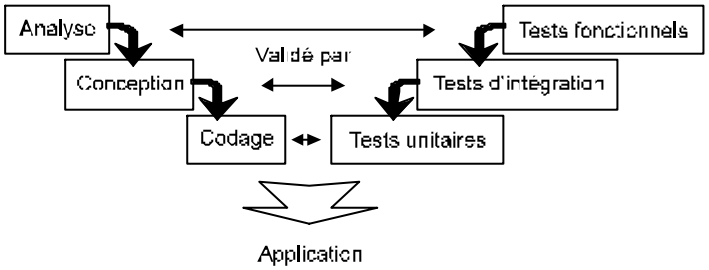


Figure 349 : Exemple de modèle de développement en V. Les tests sont développés parallèlement au logiciel.

Limites du modèle en cascade

Le modèle en cascade repose sur une séquence de phases bien cernées. Le projet produit des documents évalués lors de revues censées valider le passage d'une phase à une autre. Toutefois, la preuve effective du bon (ou mauvais) fonctionnement du système n'est réalisée que tardivement dans le cycle – lors de la phase d'intégration – lorsqu'il est possible d'évaluer concrètement le logiciel.

Avant cette phase d'intégration, seuls des documents ont été produits. Or, ce n'est pas parce qu'un document passe avec succès une étape de validation sur papier que le logiciel qu'il décrit donnera forcément des résultats convaincants.

En fait, la démarche en cascade ne donne des résultats satisfaisants que lorsqu'il est effectivement possible d'enchaîner les phases sans trop de problèmes. Il faut que l'ensemble des besoins soit parfaitement connu et le problème complètement compris par les analystes ; il faut aussi que la solution soit facile à déterminer par les concepteurs et le codage réduit idéalement à la génération automatique de code, à partir des documents de conception.

Hélas, les projets ne se présentent pas tous sous ces conditions idéales. Force est de constater que la part d'inconnu, et donc de risques, peut être grande dans certains développements, notamment du fait :

- de la méconnaissance des besoins par le client,
- de l'incompréhension des besoins par le fournisseur,
- de l'instabilité des besoins, qui découle des deux points précédents,
- des choix technologiques,
- des mouvements de personnel...

Pour toutes ces raisons, des retours d'information entre les phases sont nécessaires pour incorporer des corrections en amont, en fonction des découvertes réalisées en aval. Ces retours entre phases perturbent la vision linéaire donnée par le cycle de vie en cascade, comme le montre le diagramme suivant :

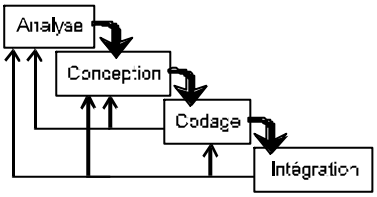


Figure 350 : La part d'inconnu qui caractérise les systèmes complexes se traduit par des retours d'information entre phases pour incorporer des corrections.

En fait, les retours entre phases ne sont que le reflet de la réalité. Tant que ces retours restent marginaux et limités entre phases adjacentes, le modèle en cascade conserve toute sa pertinence. Dans le cas contraire, le modèle en cascade n'est plus vraiment adapté. Il devient alors de plus en plus artificiel de considérer le développement comme un enchaînement linéaire.

Lorsqu'un modèle ne représente plus une réalité, il faut en changer et non essayer de faire coller la réalité au modèle[39].

Caractéristiques du cycle de vie itératif

Le cycle de vie itératif est basé sur l'évolution de prototypes exécutables, mesurables, et donc sur l'évaluation d'éléments concrets. Il s'oppose ainsi au cycle de vie en cascade qui repose sur l'élaboration de documents. Les livraisons forcent l'équipe à donner des résultats concrets régulièrement, ce qui permet d'éviter le syndrome des 90 % finis, avec encore 90 % à faire. Le déroulement régulier des itérations facilite la prise en compte des problèmes ; les changements sont incorporés dans les itérations futures, plutôt que de déranger et d'interrompre l'effort en cours.

Au cours du développement, certains prototypes sont montrés aux utilisateurs et aux clients. La démonstration des prototypes présente de nombreux avantages :

- l'utilisateur est placé devant des situations d'utilisation concrètes qui lui permettent de mieux structurer ses désirs et de les communiquer à l'équipe de développement ;
- l'utilisateur devient partenaire du projet ; il prend sa part de responsabilité dans le nouveau système et, de fait, l'accepte plus facilement ;
- l'équipe de développement est plus fortement motivée du fait de la proximité de l'objectif ;
- l'intégration des différents composants du logiciel est réalisée de manière progressive, durant la construction, sans effet *big bang* à l'approche de la date de livraison ;
- les progrès se mesurent par des programmes démontrables, plutôt que par des documents ou des estimations comme dans le cycle de vie en cascade. L'encadrement dispose ainsi d'éléments objectifs et peut évaluer les progrès et l'état d'avancement avec plus de fiabilité.

En revanche, le cycle de vie itératif demande plus d'attention et d'implication de l'ensemble des acteurs du projet. Il doit être présenté et compris par tous : les clients, les utilisateurs, les développeurs, l'encadrement, l'assurance qualité, les testeurs, les documentalistes. Tous doivent organiser leur travail en conséquence.

Une mini-cascade

Dans le cycle de vie itératif, chaque itération reproduit le cycle de vie en cascade, à une plus petite échelle. Les objectifs d'une itération sont établis en fonction de

---

[39] Kuhn T. S. 1970, *The Structure of Scientific Revolutions*, 2nd edition. The University of Chicago Press, Chicago.

l'évaluation des itérations précédentes. Les activités s'enchaînent ensuite dans une mini-cascade dont la portée est limitée par les objectifs de l'itération.

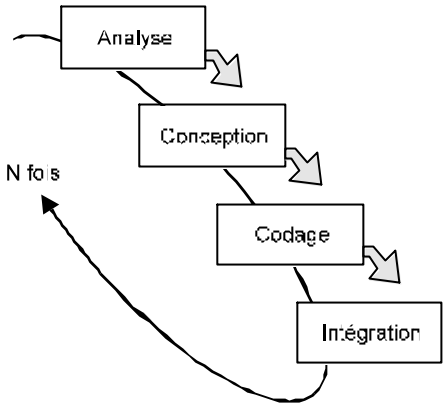


Figure 351 : Le cycle de vie itératif repasse plusieurs fois par les phases du cycle de vie en cascade.

Les phases traditionnelles sont couvertes graduellement, itération après itération. La nature des activités menées au sein des phases ne varie pas fondamentalement selon qu'il s'agit d'un développement itératif ou d'un développement en cascade. La différence se situe plus dans la distribution de ces activités par rapport aux phases.

Chaque itération comprend les activités suivantes :

- *la planification* de l'itération est élaborée en fonction des résultats de l'étude de risques ;
- *l'analyse* étudie les cas d'utilisation et les scénarios à réaliser dans l'itération ; elle met à jour le modèle d'analyse pour prendre en compte les nouvelles classes et associations découvertes pendant l'analyse de l'itération ;
- *la conception* se concentre sur les choix tactiques nécessaires pour réaliser les mécanismes alloués à l'itération. Au besoin, des retouches sont apportées à l'architecture et le modèle de conception est mis à jour. Les procédures de test sont définies parallèlement à la conception des nouveaux mécanismes ;
- *le codage et les tests* dont la charpente du code est générée automatiquement depuis le modèle de conception. Le détail des opérations est ensuite réalisé manuellement. L'intégration du nouveau code avec le code existant, issu des itérations précédentes, est réalisée graduellement pendant la construction. Les procédures de tests unitaires et d'intégration sont appliquées au prototype.

- *l'évaluation de la livraison exécutable* : l'examen des résultats des tests sert de base pour l'évaluation du prototype. Les résultats sont évalués par rapport aux critères définis avant de lancer l'itération.
- *la préparation de la livraison* : le prototype est gelé et l'ensemble des éléments qui sont entrés dans son développement est placé dans des bibliothèques contrôlées. Les différents documents de description et d'installation du prototype sont finalisés.

Dans le diagramme suivant, les activités internes se chevauchent pour montrer qu'au sein d'une itération, elles n'ont pas besoin de se terminer brutalement et que la transition entre deux activités peut être progressive. Le cycle itératif se distingue ici du cycle en cascade qui effectue un amalgame entre phases et activités.

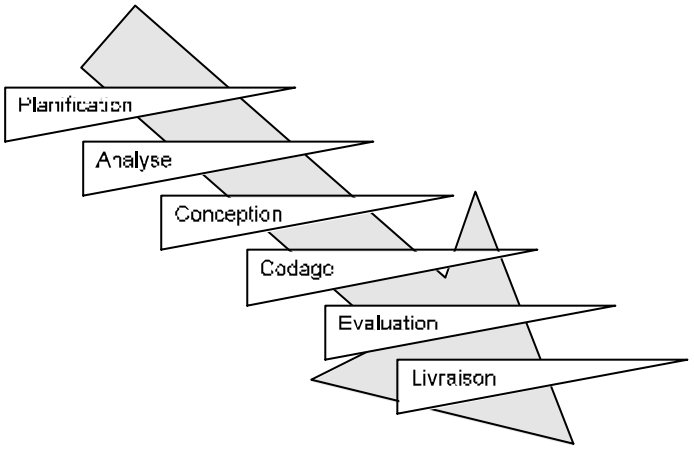


Figure 352 : Exemple de transition progressive entre les activités d'une itération.

Idées fausses sur le cycle de vie itératif

Le monde est plein de préjugés et d'idées fausses. L'informatique n'échappe pas à cette tendance et, en particulier, le cycle de vie itératif draine dans son sillage bon nombre de mythes, parmi lesquels :

- *Le cycle de vie itératif encourage la bidouille.*

Le cycle de vie itératif encourage surtout la créativité en ménageant des espaces de liberté, sans règles trop strictes.

- *Le cycle de vie itératif engendre des problèmes.*

Le cycle de vie itératif n'engendre pas les problèmes : il révèle précocement des problèmes qui auraient été détectés tardivement avec le cycle en cascade. Le cycle en cascade semble prévisible : les revues d'analyse et de conception se

succèdent dans les délais prévus. Tout va bien jusqu'à la phase d'intégration où mille turpitudes sont commises pour livrer quelque chose au client. Le cycle de vie itératif commence mal et se termine bien, contrairement au cycle de vie en cascade qui commence bien et se termine mal.

- *Le cycle de vie itératif et incrémental demande de recommencer n fois jusqu'à ce que le résultat soit bon.*

Le cycle de vie itératif ne demande pas de recommencer éternellement le même programme, mais d'enrichir un programme existant par l'ajout de nouvelles fonctionnalités. La confusion vient du fait que certaines parties doivent être retouchées, précisément parce qu'elles comportaient des défauts détectés par une itération. Plus tôt les problèmes sont détectés dans le cycle de développement, moins ils sont coûteux à corriger.

- *Le cycle de vie itératif est une excuse pour ne pas planifier et gérer un projet.*

Au contraire, le cycle de vie itératif demande plus de planification et une attention soutenue. La planification n'est pas faite au début ; elle est répartie sur l'ensemble du développement. Il est facile de faire des plans, il est plus difficile de les corriger ou d'en changer.

- *Le cycle de vie itératif ne concerne que les développeurs.*

Il est vrai que les développeurs se sentent à l'aise avec le cycle itératif, mais ce n'est pas parce que les uns sont heureux que les autres – en particulier l'encadrement – doivent être malheureux. En effet, le cycle de vie itératif leur fournit plus de points de mesure que le cycle en cascade et ces mesures sont plus fiables : elles sont connectées sur la réalité de l'évaluation d'un prototype, plutôt que sur la virtualité de la lecture d'une documentation.

- *Le cycle de vie itératif encourage à rajouter toujours de nouveaux besoins, sans fin.*

Voilà une idée fausse dont il faut absolument se défaire. L'analyse itérative ne consiste pas à rajouter toujours et encore de nouveaux besoins ; elle permet de se concentrer d'abord sur les besoins majeurs et d'incorporer plus tard les besoins secondaires. Le cycle de vie itératif évite la paralysie par l'analyse.

Stabilité des besoins et cycles de vie

Le cycle de vie itératif est souvent décrié par des informaticiens qui ont souffert de projets dans lesquels l'expression des besoins des utilisateurs n'était pas stable. Ces informaticiens sont d'autant plus virulents que le cycle de vie itératif leur apparaît comme une perpétuelle invitation à la surenchère. Ils l'opposent par là au cycle en cascade, dans lequel les phases ont théoriquement une limite bien définie.

Cette crainte à l'égard du cycle de vie itératif n'est pas fondée. Quelle que soit la forme de cycle de vie, l'explosion des besoins est toujours la conséquence d'un

mauvais départ. Il n'y a pas tellement de nouveaux besoins, il y a surtout des besoins sous-estimés ou mal identifiés.

Le diagramme suivant représente le phénomène d'explosion des besoins. D'une part, les besoins ne sont pas stables, et d'autre part, les utilisateurs en rajoutent toujours de nouveaux.

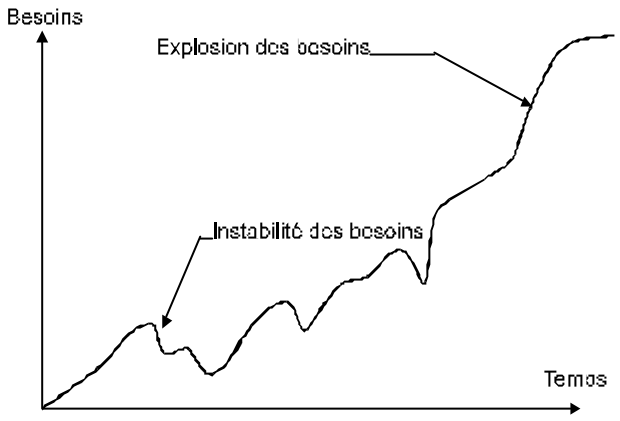


Figure 353 : Représentation du phénomène d'explosion des besoins.

Pour régler le problème de l'explosion des besoins, il faut impérativement aider (il faudrait plutôt dire forcer) les utilisateurs à articuler leurs besoins dans un cadre stable et structurant. Les cas d'utilisation apportent une très bonne technique pour ce faire ; leur élaboration oblige les utilisateurs à imaginer comment ils comptent utiliser le futur système. Ceci reste difficile en l'absence d'éléments concrets à critiquer, comme c'est le cas avec le cycle en cascade qui, dans ses premières phases, ne produit que des documents. Le cycle de vie itératif vient au secours des utilisateurs en leur fournissant des prototypes à évaluer. Les informaticiens peuvent de leur côté expérimenter leurs choix de réalisation en vraie grandeur.

Dès le début du développement, un cycle itératif produit des prototypes documentés qui peuvent être évalués objectivement. Il s'oppose au cycle en cascade qui ne fournit que des documents. Dans le premier cas, le logiciel est mis en avant ; dans le second cas, ce sont les sous-produits. A qualité d'encadrement égale, un cycle itératif produit des résultats plus stables qu'un cycle linéaire.

Le cycle en cascade ne secoue pas assez les utilisateurs ; les documents qu'il produit ne procurent qu'une illusion de progrès ; rien ne dit que leur contenu satisfait réellement les besoins. Les revues de conception se passent bien, le projet suit le plan de développement jusqu'à la phase d'intégration. Là, plus rien ne marche : de nombreuses réparations sont alors effectuées dans la précipitation,

sans recul suffisant, et cela se traduit par des retards et un produit fragile et difficile à maintenir.

Le diagramme suivant représente la courbe de connaissance des besoins dans une approche itérative. Tous les besoins ne sont pas nécessairement connus dès le début ; la connaissance des besoins peut progresser avec les itérations.

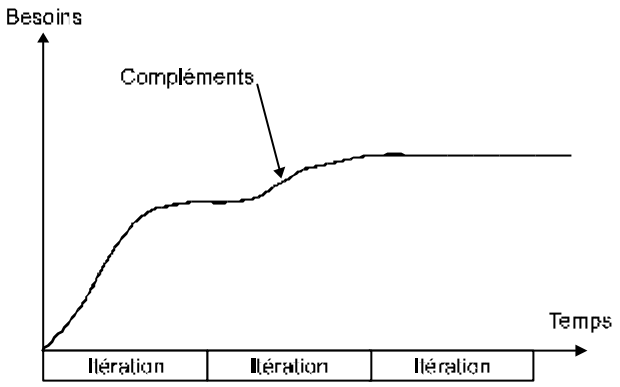


Figure 354 : Exemple de progression dans la connaissance des besoins dans le cadre d'une démarche itérative.

L'adoption de l'approche itérative ne fait pas disparaître tous les problèmes par magie. L'approche itérative est seulement un outil pour mieux gérer les problèmes.

There is no silver bullet[40]

La documentation

Les prototypes sont les principaux produits tangibles issus du cycle de vie itératif, de sorte qu'il n'est pas nécessaire de fournir des documents pour servir de support aux revues de validation de fin de phase, comme dans le cycle en cascade.

Il est néanmoins possible de construire une documentation conventionnelle lorsque celle-ci est exigée par contrat. Cette documentation est constituée de documents d'analyse et de conception par exemple dans le format normalisé DoD 2167a.

---

[40] Brooks, F. P. April 1987, *No silver bullet, essence and accidents of software engineering*. IEEE Computer.

La documentation n'est pas construite en une seule passe, mais graduellement, lors de chaque itération. A la fin du développement, les documents obtenus de manière itérative ne se distinguent pas dans leur forme de ceux obtenus de manière conventionnelle. En revanche, ils restituent une image fidèle de l'état final du projet avec lequel ils ont évolué de manière synchrone.

Le diagramme suivant montre que la documentation obtenue graduellement lors d'un développement itératif peut très bien être présentée comme issue d'un développement conventionnel. Selon les lecteurs, l'aspect itératif du développement gagne à être occulté une fois le projet terminé.

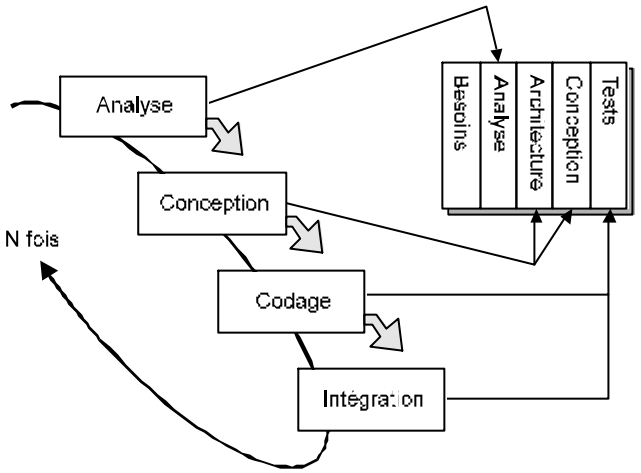


Figure 355 : La documentation générée par un développement itératif est semblable dans sa forme à la documentation traditionnelle, mais reflète plus fidèlement l'état final du projet.

Variantes du cycle itératif

Il existe plusieurs variantes du cycle itératif selon la taille du projet, la complexité du domaine et les choix d'architecture.

Dans sa variante la plus simple, la forme du cycle de vie itératif correspond à la lettre b. La forme en b a été identifiée par Birrel et Ould[41], qui réservaient l'itération uniquement à la phase de maintenance.

La forme en b est bien adaptée pour des applications de taille modeste ou pour des applications parfaitement définies, réalisées dans le cadre d'une architecture éprouvée qui ne créera pas de surprises.

---

[41] Birrel N. D. et. Ould M. A. 1985, *A Practical Handbook for Software Development*. Cambridge University Press, Cambridge, U. K.

Les cycles y sont rapides et l'intégration continue, de sorte que le principal avantage de cette forme itérative, par rapport à un cycle en cascade, est de supprimer l'effet big bang d'une phase d'intégration localisée en fin de cycle.

Le diagramme suivant représente l'enchaînement des activités au sein d'un cycle de vie itératif en b. L'itération est centrée sur la construction.

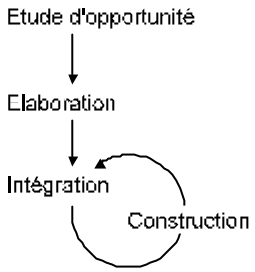


Figure 356 : Cycle itératif en b. Cette forme de cycle itératif est bien adaptée pour les projets modestes ou bien définis.

Le cycle en b concentre les itérations sur la construction. Lorsque tous les besoins ne sont pas connus dès le départ, ou pour prendre en compte des retouches d'architecture, il est fréquent de rajouter des boucles de rattrapage sur l'étude d'opportunité ou sur l'élaboration. Le flot principal de l'itération reste concentré sur la construction par incréments, les flots secondaires ayant pour vocation d'affiner l'analyse ou l'architecture plutôt que de les remettre en cause.

Le diagramme suivant montre comment le cycle en b a tendance à s'ouvrir lorsque des retouches de conception ou d'analyse doivent être apportées suite aux enseignements recueillis lors de la construction.

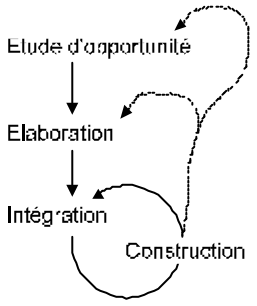


Figure 357 : Cycle itératif en b avec quelques retouches sur les activités en amont de la construction. L'itération reste principalement concentrée sur la construction.

Le cycle en b est tout à fait adapté pour les développements bien circonscrits. Une grande part d'incertitude sur les besoins est plus grande, une architecture

partiellement déterminée ou un projet mené par plusieurs équipes en parallèle, rendent prépondérantes les branches secondaires : la forme du cycle se rapproche d'une lettre en O. Le cycle en O dérive du cycle en spirale de Boehm[42]. Il a été décrit par Philippe Kruchten[43] et s'inspire d'une lettre ouverte de Mike Devlin[44].

Le diagramme suivant représente la forme d'un cycle de vie en O. L'itération comprend l'analyse et la conception.

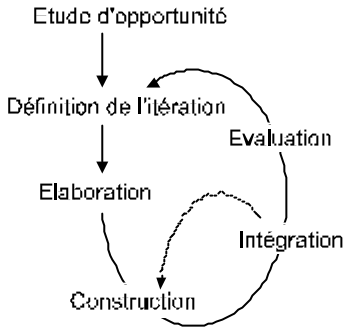


Figure 358 : Cycle de vie itératif en O. Cette forme de cycle itératif est bien adaptée pour les grands projets.

Quelle que soit la forme du cycle de vie itératif retenue, il convient de conduire l'analyse aussi loin que possible, sans être paralysé par la recherche de l'analyse parfaite. Le cycle itératif, avec son exigence de livraisons régulières de programmes exécutables, encourage les projets à décoller et à produire des résultats concrets.

Evaluation des itérations

Les critères d'évaluation d'une itération doivent être définis avant de commencer l'itération. Chaque itération est décrite par un plan détaillé, lui-même inclus dans un plan de développement général. Une itération est marquée par des étapes intermédiaires, comme des séances de relecture critique du code. Elles permettent

---

[42] Boehm B. W. May 1988, *A Spiral Model of Software Development and Enhancement*. IEEE Computer.

[43] Kruchten P. December 1991, *Un processus de développement de logiciel itératif et centré sur l'architecture*. Proceedings of the 4th International Conference on Software Engineering, Toulouse, EC2, Paris.

[44] Mike Devlin est co-fondateur et président du conseil d'administration de Rational Software Corporation.

de mesurer les progrès et d'entretenir le moral de l'équipe qui se concentre sur des objectifs à court terme.

Le diagramme suivant représente une itération d'un petit projet, délimitée par deux revues : la première, pour lancer formellement l'itération après avoir vérifié l'allocation des ressources et la deuxième, pour valider l'itération et autoriser la livraison du prototype correspondant.

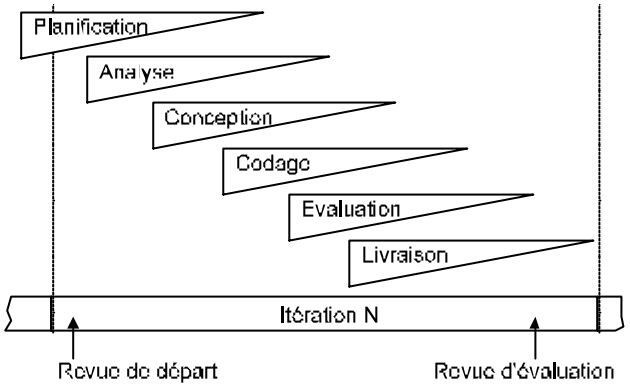


Figure 359 : Exemple d'itération délimitée par deux revues.

Le nombre d'étapes dépend de la longueur de l'itération. Pour de petites itérations – entre un et trois mois – il faut compter environ deux étapes :

- la revue de départ qui fige les objectifs de l'itération, les critères d'évaluation et la liste des scénarios à réaliser,
- la revue d'évaluation qui valide les résultats de l'itération par rapport aux critères définis avant de dérouler l'itération.

Pour des itérations plus longues, il faut prévoir plus de revues intermédiaires pour l'approbation des plans de tests, la mise en place de l'assurance qualité, etc.

Les résultats de l'itération sont évalués par rapport aux critères d'évaluation définis lors de la planification de l'itération. Ils sont fonction des performances, des capacités et des mesures de qualité. Il faut également prendre en compte les éventuels changements externes, comme l'apparition de nouveaux besoins ou la découverte des plans de la concurrence, et déterminer s'il faut reprendre certaines parties du système et les assigner aux itérations restantes.

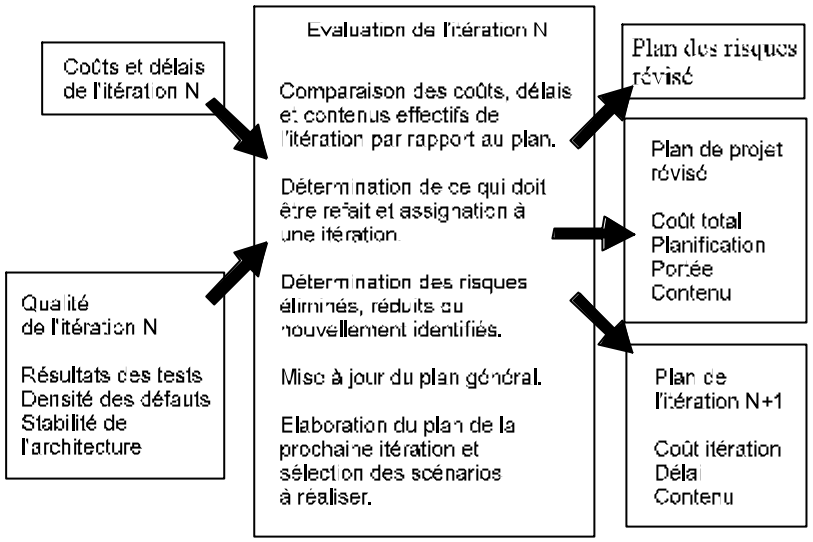


Figure 360 : Principaux produits et activités de l'évaluation d'une itération.

L'évaluation est une étape essentielle pour bénéficier pleinement de l'approche itérative. Il faut garder présent à l'esprit qu'il vaut parfois mieux réviser les critères plutôt que modifier le système. Une itération peut révéler qu'un besoin particulier n'est pas aussi important que prévu, qu'il est trop difficile à réaliser ou trop cher à satisfaire. Dans ce cas, une analyse comparative entre le coût et le bénéfice doit être menée.

La planification des itérations

Il n'y a pas de réponse toute faite. Pour un projet de 18 mois environ, il y aura entre trois et six itérations. Les itérations ont toutes à peu près la même durée.

Le diagramme suivant représente des exemples d'allocation de l'effort dans un développement itératif :

- La courbe B correspond à une première itération trop ambitieuse. Le projet B rencontre les désagréments du cycle en cascade : l'intégration échoue, le développement entre dans une période d'instabilité et finit par dépasser les délais, sans atteindre ses objectifs ou sans respecter ses impératifs de qualité.
- La courbe A correspond a un projet qui a consacré les deux premières itérations à la mise en place d'une architecture stable, au prix de petites retouches, et qui connaît une évolution itérative sans surprises. Le léger dépassement d'achèvement représente l'évolution des besoins qui ont été pris en compte.

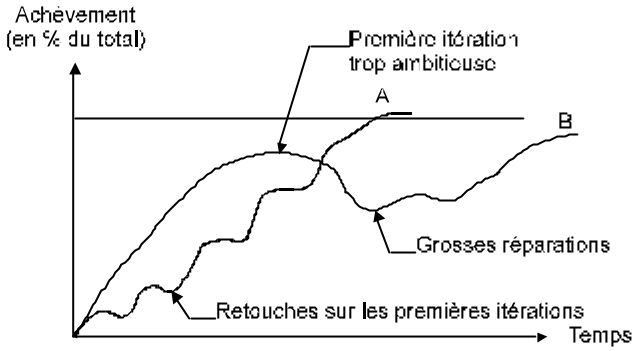


Figure 361 : Illustration de l'impact de la planification des itérations sur le déroulement du projet.

La première itération est la plus difficile à conduire car elle demande la mise en place de l'ensemble de l'environnement de développement et de l'équipe. Le projet rencontre généralement de nombreux problèmes avec les outils, l'intégration entre les outils, les relations au sein de l'équipe.

Les équipes qui appliquent une approche itérative sont généralement trop optimistes. Il faut rester modeste et ne pas attendre trop de la première itération, faute de quoi les délais vont être dépassés, le nombre d'itérations éventuellement réduit et les bénéfices de l'approche itérative limités.

L'équipe perd généralement pas mal de temps à travailler avec le nouveau compilateur, à mettre en place la gestion de versions et de configurations, les tests de non-régression et le million d'autres choses qui doivent être réglées lors de la première itération. Pour la deuxième itération, tout rentre dans l'ordre, chacun sait comment procéder avec les outils et l'équipe peut se concentrer sur les fonctionnalités.

Pilotage des projets objet

Pourquoi parler de pilotage des projets objet ? Les projets objet demandent-ils une forme particulière d'encadrement, et inversement, n'est-il pas possible de piloter les projets objet comme les projets classiques ?

En fait oui et non. L'adoption des technologies objet n'implique pas obligatoirement de revoir les procédures de développement en place. Néanmoins, il faut être conscient que le seul moyen de réellement bénéficier de l'approche objet consiste à établir une solution totalement objet.

Cette évolution des pratiques de l'entreprise exige une adhésion totale des différents intervenants.

Il n'existe pas une forme de processus de développement adaptée à tous les projets, tous les domaines d'application et toutes les cultures d'entreprise. Le processus de développement décrit dans ce chapitre est défini de manière générale. Il appartient à chaque projet de l'adapter en fonction de ses contraintes propres.

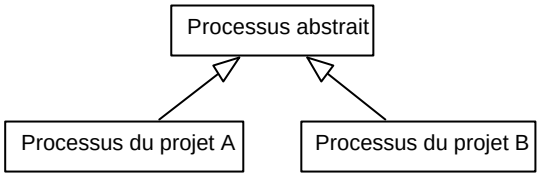


Figure 362 : Il appartient à chaque projet d'adapter le processus abstrait décrit de manière générale dans ce paragraphe.

Le processus de développement d'un logiciel peut être observé selon deux points de vue complémentaires :

- la vue de l'encadrement qui se préoccupe des aspects financiers, stratégiques, commerciaux et humains,
- la vue technique qui se concentre sur l'ingénierie, le contrôle de la qualité et la méthode de modélisation.

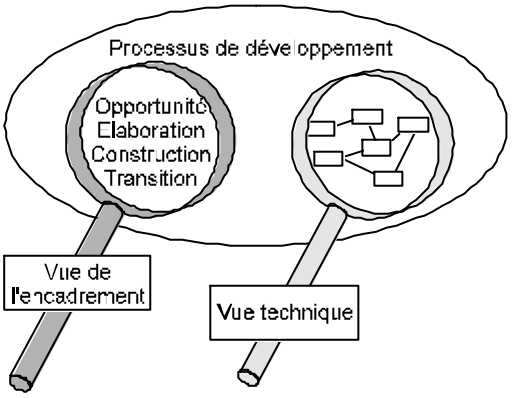


Figure 363 :Points de vue portés sur le processus de développement.

La vue de l'encadrement

La vue de l'encadrement représente le développement du logiciel comme un enchaînement de quatre phases :

- l'étude d'opportunité qui comprend l'étude du marché, la spécification du produit final et la définition de la portée du projet ;
- l'élaboration qui correspond à la spécification des particularités du produit, à la planification des activités, à la détermination des ressources, à la conception et à la validation de l'architecture du logiciel ;
- la construction qui regroupe la réalisation du produit, l'adaptation de la vision, de l'architecture et des plans, jusqu'à la livraison du produit ;
- la transition qui rassemble la fabrication à grande échelle, la formation, le déploiement dans la communauté des utilisateurs, le support technique et la maintenance.

Un cycle de développement correspond à une passe au travers des quatre phases et donne une nouvelle génération du logiciel.

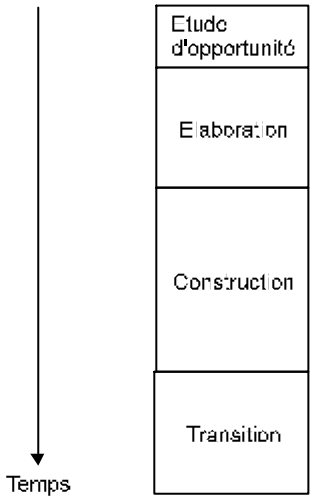


Figure 364 : Dans la vue de l'encadrement, le développement d'une génération du logiciel correspond à une passe au travers de quatre phases.

Les progrès sont mesurés en termes d'avancement dans les phases. Certains logiciels demandent plusieurs cycles pour prendre en compte des améliorations, des demandes de nouvelles fonctionnalités, etc. La phase de transition d'une génération donnée recouvre alors souvent la phase d'étude d'opportunité de la génération suivante. Dans le cas de grands projets, les cycles peuvent être menés en parallèle et plusieurs sous-systèmes sont développés indépendamment.

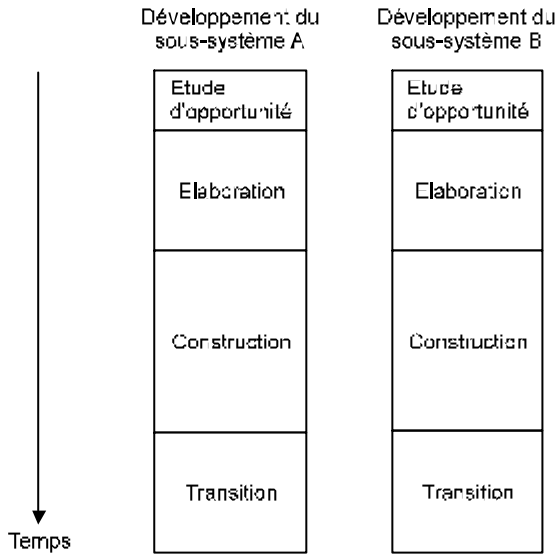


Figure 365 : Dans le cas de grands projets, des développements de sous-systèmes peuvent être menés en parallèle.

La vue technique

La vue technique se concentre sur la mise œuvre et l'ordonnancement de toutes les activités techniques qui conduisent à la livraison d'une génération du logiciel.

Le cycle de développement est perçu comme une suite d'itérations par lesquelles le logiciel évolue par incréments. Chaque itération débouche sur la livraison d'un programme exécutable. Le contenu d'une itération doit être utile pour le développement ou pour l'utilisateur. Il est déterminé en choisissant un sous-ensemble des fonctionnalités de l'application au sein des cas d'utilisation. Certaines livraisons sont purement internes, d'autres servent à démontrer l'état d'avancement, certaines enfin sont placées chez l'utilisateur. D'itération en itération, le programme grandit en fonctionnalité et en qualité, jusqu'à ce qu'une itération particulière donne la première génération du logiciel.

Les activités traditionnelles ne sont pas effectuées en séquence comme dans un cycle de vie en cascade ; elles sont distribuées – saupoudrées – sur les différentes itérations. Chaque itération comprend des activités de planification, d'analyse, de conception, d'implémentation, de test et d'intégration, dont le nombre varie en fonction de la position de l'itération dans le cycle.

Le programme exécutable qui résulte d'une itération est appelé prototype. Un prototype est une étape concrète de la construction d'un logiciel. Il fait la preuve tangible – mesurable – des progrès réalisés. Un prototype est un sous-ensemble

de l'application, utile pour la formulation des besoins ou la validation des choix technologiques. Les prototypes s'enrichissent à chaque itération.

Le premier prototype a souvent pour seul objectif d'effectuer la preuve d'un concept d'application. Il est développé le plus rapidement possible, sans tenir compte des critères de qualité habituels. Ce genre de prototype, le plus souvent jetable, est appelé maquette.

Les avant-dernières itérations produisent les versions bêta, généralement placées en test chez une sélection d'utilisateurs bienveillants.

La version livrée à l'utilisateur est un prototype comme les autres, issu de la chaîne des prototypes ; elle peut être vue comme le dernier prototype de la génération courante et le premier prototype de la prochaine génération du système.

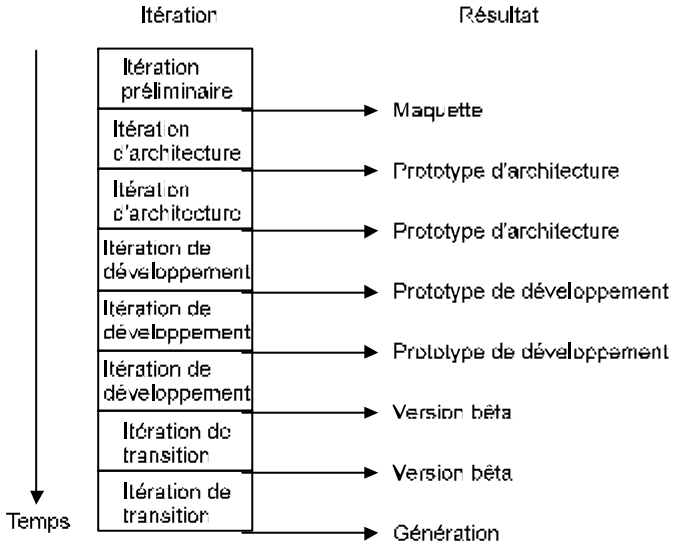


Figure 366 : Représentation de la chaîne des prototypes.

Intégration des deux points de vue

Les deux points de vue sont synchronisés à la fin de chaque phase, sur le résultat tangible d'une itération.

L'étude d'opportunité peut faire appel à un prototype pour déterminer la viabilité d'un projet ou sa faisabilité technique. L'évaluation de la maquette exécutable, livrée par l'équipe de développement, participe à la décision de poursuite ou non du projet, donnée à la fin de l'étude d'opportunité.

La phase d'élaboration a pour objectif essentiel de définir une architecture qui permette le développement et l'évolution de l'application. Il est difficile de trouver une bonne architecture du premier coup, d'où l'intérêt d'effectuer un certain nombre de prototypes d'architecture. L'élaboration se termine par la définition d'une base d'architecture, validée par un prototype d'architecture.

La phase de construction progresse au rythme des prototypes de développement, dont l'objectif premier est de mesurer les progrès et de garantir la stabilisation de la conception. La phase de construction se termine par la livraison de versions préliminaires, les versions bêta, qui sont placées chez des utilisateurs en vue d'être testées en vraie grandeur, dans l'environnement de déploiement.

La phase de transition débute par l'installation des versions bêta sur site. Elle progresse au rythme des livraisons de versions qui viennent corriger les défauts détectés par les utilisateurs. Selon les organisations, la phase de transition s'arrête lorsque le logiciel est arrivé à sa version de production, ou alors lorsque la prochaine génération est livrée aux utilisateurs.

Il appartient à chaque projet de déterminer le nombre d'itérations qu'il y a dans chaque phase ; le diagramme suivant donne un exemple typique.

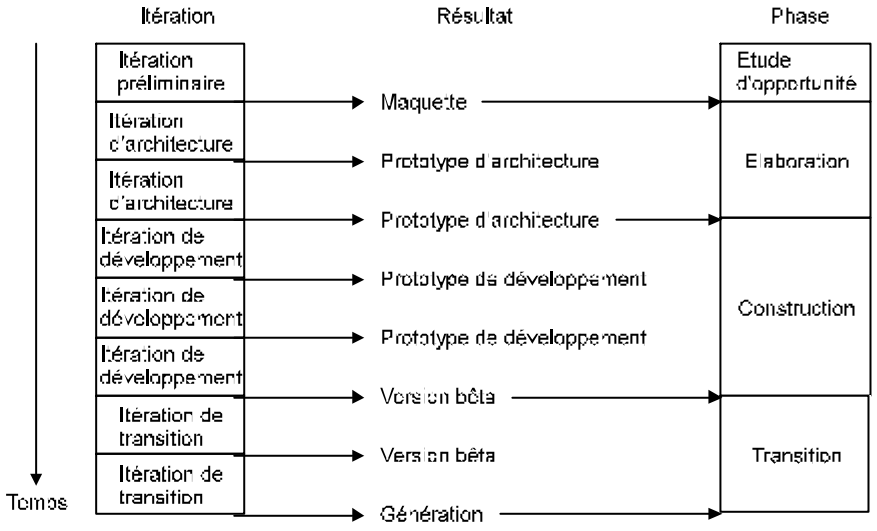


Figure 367 : Les deux points de vue sont synchronisés en fin de phase. La fin d'une phase est marquée par la validation d'un prototype.

Gestion du risque dans un développement itératif

Toute activité humaine comporte sa part de risques. Le développement de logiciels n'échappe pas à cette règle. Les projets informatiques doivent gérer les gérer afin de minimiser leurs conséquences.

L'analyse des risques consiste à évaluer le projet, la technologie et les ressources, dans le but de déterminer et de comprendre la nature et l'origine des risques. Chaque risque est décrit brièvement et les relations entre les risques sont soulignées. L'évaluation des risques détermine dans quelle mesure les conséquences d'un risque sont acceptables.

Les risques doivent faire l'objet d'une description écrite, pas nécessairement dans un formalisme très poussé. Tous les membres du projet doivent en prendre connaissance. Les risques doivent êtres quantifiés, sinon il est impossible de savoir si les risques ont été éliminés ou non. Il ne faut jamais laisser traîner des expressions floues dans la documentation, comme : le système devra être rapide, la capacité de la mémoire sera suffisante, etc.

Il existe quatre grandes catégories de risques :

- *les risques commerciaux* : la concurrence peut-elle capturer le marché avant que le produit ne soit prêt ? Vaut-il mieux sortir une livraison minimale pour occuper le terrain ?
- *les risques financiers* : l'entreprise dispose-t-elle de capacités financières suffisantes pour mener le projet à son terme ?
- *les risques techniques* : la base technologique est-elle solide et éprouvée ?
- *les risques de développement* : l'équipe est-elle suffisamment expérimentée et maîtrise-t-elle les technologies mises en œuvre ?

Mais avant tout, il faut bien comprendre que le plus grand risque consiste à ne pas savoir où sont les risques. Il faut aussi se méfier des équipes pléthoriques et prendre garde aux chefs de projet qui mesurent leur prestige au nombre de développeurs sous leur responsabilité. Il ne sert à rien d'avoir beaucoup de développeurs : il faut avoir les bons et savoir les faire travailler en équipe. La gestion des risques regroupe toutes les activités requises pour construire un plan de réduction des risques et pour conduire ce plan à terme.

Le plan de réduction des risques décrit, pour chaque risque identifié :

- l'importance par rapport au client,
- l'importance par rapport au développement,
- l'action entreprise et ses implications économiques,
- le moyen de vérifier que le risque a été supprimé ou réduit, et dans quelle mesure,

- les scénarios affectés par le risque,
- l'allocation à un prototype.

La réduction du risque pilote les itérations dans le cycle de vie itératif. Chaque itération a pour objectif d'éliminer une part du risque global associé au développement d'une application. Le diagramme suivant illustre le mécanisme de réduction du risque par les itérations. Chaque itération est construite comme un petit projet, avec pour objectif la réduction d'une partie des risques identifiés lors la définition du projet.

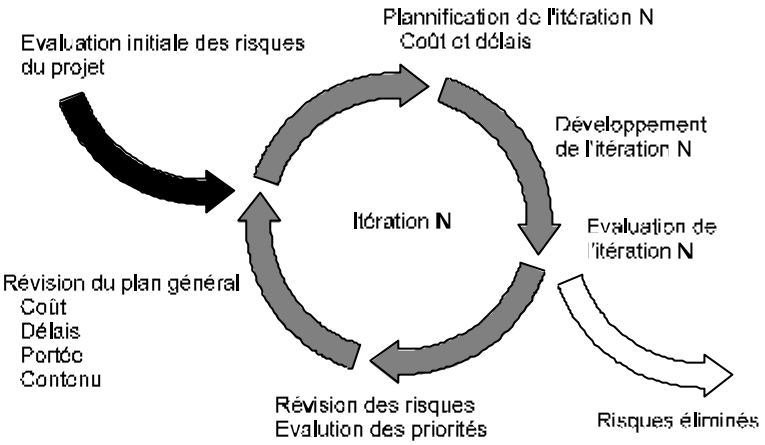


Figure 368 : Chaque itération est construite comme un petit projet, dédié à l'élimination d'une partie des risques identifiés lors de la définition du projet.

En suivant l'approche en cascade, la réduction des risques est insignifiante jusqu'à ce que la phase d'intégration commence. Les plus grands risques techniques sont liés à l'architecture : performances (vitesse, capacité, précision), intégrité des interfaces du système, qualité du système (adaptabilité, portabilité). Ces éléments ne peuvent pas être évalués avant qu'une quantité significative de code ait été développée et intégrée. Le diagramme suivant représente l'allure de la courbe de décroissance du risque dans un développement en cascade. Le risque reste élevé pendant la majeure partie du cycle de vie.

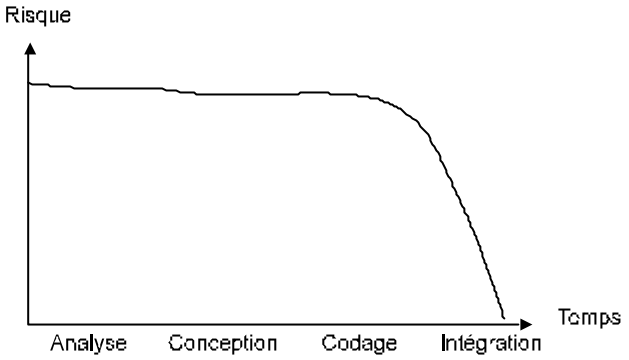


Figure 369 : Courbe de décroissance du risque dans un développement en cascade.

L'approche itérative identifie et traite les risques plus rapidement dans le cycle de vie, de sorte que le risque peut être réduit de manière significative dès la phase d'élaboration. D'itération en itération, les prototypes exécutables valident les choix du projet de manière concrète et mesurable, et le niveau de confiance dans l'application augmente. Le diagramme suivant montre la décroissance plus rapide du risque dans un développement itératif. La principale différence avec le diagramme précédent provient de l'axe des abscisses, gradué par les multiples itérations ; elles se concentrent toutes sur la réduction d'une part du risque global.

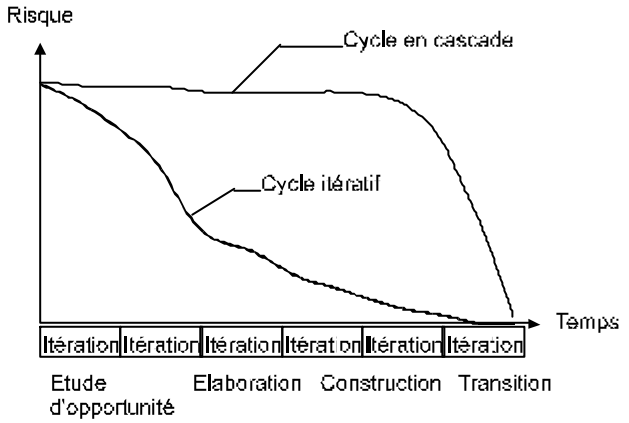


Figure 370 : Courbe de décroissance du risque dans un développement itératif.

Chaque itération est basée sur la construction d'un nombre réduit de scénarios qui s'attaquent d'abord aux risques les plus importants et déterminent les classes et les catégories à construire dans l'itération. Les catégories forment un espace de travail bien adapté aux développeurs.

Les testeurs utilisent les mêmes scénarios pour construire le plan de test et les procédures de test de l'itération. A la fin de l'itération, l'évaluation détermine les risques qui ont été éliminés ou réduits. Le plan des prototypes est révisé en conséquence.

Le diagramme suivant montre un exemple de répartition des risques entre deux prototypes consécutifs. Il illustre l'effet d'une itération en terme de réduction de risque.

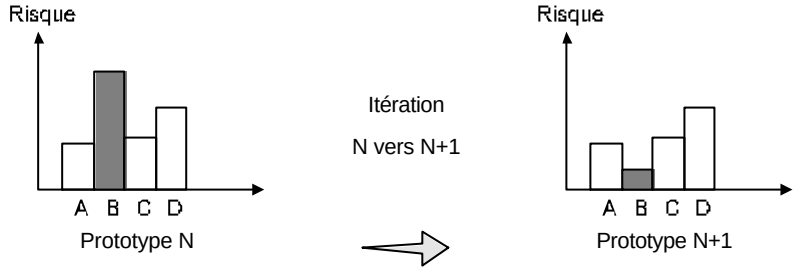


Figure 371 : Une itération a pour objectif la réduction du risque associé à une partie de l'application. La partition des fonctionnalités de l'application est réalisée selon les cas d'utilisation.

Chaque prototype explore une partie de l'application dans le but de réduire une part des risques. Selon la nature des risques à étudier, les prototypes s'enfoncent plus ou moins profondément dans l'architecture.

Le diagramme suivant représente différentes sortes de prototypes :

- le prototype d'IHM, limité à l'interface utilisateur, est souvent une maquette construite dans le but d'aider l'utilisateur à articuler l'expression de ses besoins ;
- le prototype *middleware* (une couche logicielle intermédiaire) teste et valide une couche de logiciel réutilisable, probablement achetée à l'extérieur ;
- le prototype technologique vérifie, par exemple, la qualité des encapsulations du matériel, ou des particularités des systèmes d'exploitation ;
- le prototype fonctionnel traverse toute l'architecture ; il garantit la pertinence de l'ensemble des choix stratégiques.

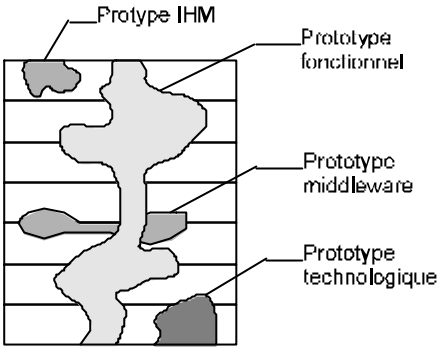


Figure 372 : Exemples de prototypes. Chaque prototype explore une partie de l'application.

Les prototypes sont enchaînés une fois que l'architecture a été validée. Chacun de ces prototypes correspond à un ou plusieurs cas d'utilisation. A chaque itération la couverture des besoins augmente.

Le diagramme suivant montre comment quatre prototypes ont été déployés pour développer une application. Les deux premiers prototypes ont dû être retouchés, pour corriger l'architecture ou pour prendre en compte de nouveaux besoins.

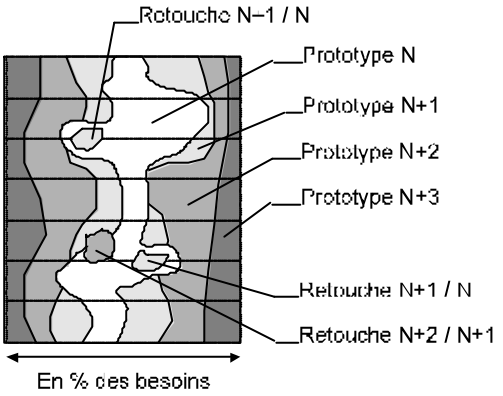


Figure 373 : Représentation de la couverture des besoins, prototype par prototype.

Le développement itératif ne consiste pas à recommencer éternellement le même travail. Bien qu'il soit normal de refaire certaines parties[45], la tendance générale est de conserver le résultat des itérations précédentes. Le premier prototype est jetable : d'où la limite supérieure placée à 100 % de retouche. Si le système reprend du code existant, la quantité de retouches doit être plus faible. Une fois

[45] Brooks, F. P. 1975, *The Mythical Man-Month*. Addison-Wesley, New York.

que l'architecture est établie et stabilisée, les retouches restent faibles et bien localisées. La somme des retouches des itérations suivantes doit rester inférieure à 25 %. Les pourcentages, indiqués dans le graphique suivant, se rapportent à l'ensemble du système et pas seulement à l'itération la plus récente.

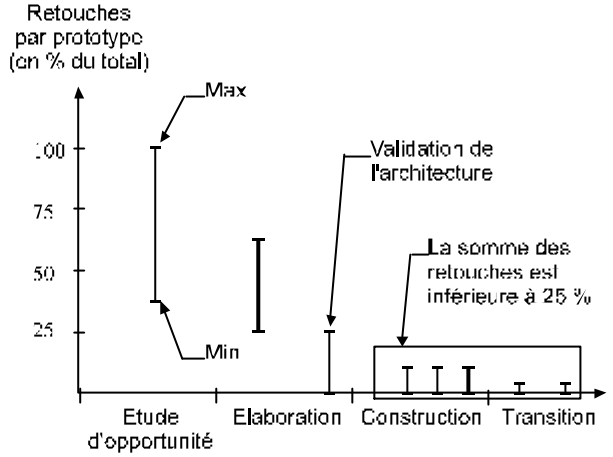


Figure 374 : Exemple de l'évolution des retouches avec le déroulement des prototypes. Une fois l'architecture validée et stabilisée, la somme des retouches reste inférieure à 25 % du total.

Gestion du risque phase par phase

Chaque phase du développement apporte sa contribution à la réduction des risques, avec à chaque fois un éclairage différent :

- *l'étude d'opportunité* essaie de délimiter les risques du projet, en construisant un prototype pour valider les concepts ;
- *l'élaboration* commence par réduire les risques d'incompréhension entre les utilisateurs et les développeurs : elle développe une vision partagée de la portée du projet et explore des scénarios avec les utilisateurs et les experts du domaine. Dans un deuxième temps, l'élaboration valide les choix d'architecture. L'architecture est raffinée par les itérations suivantes, au besoin ;
- *la construction* est le siège de la fabrication incrémentale de l'application. L'intégration progressive des divers composants supprime les risques de discordance rencontrés dans les phases dédiées spécifiquement à l'intégration, en fin de développement en cascade ;

- *la transition* : les risques de prise en main du logiciel sont réduits par le déploiement graduel de l'application en phase de transition, et cela d'autant plus lorsque les utilisateurs ont été impliqués dès les phases précédentes ;
- *le post-déploiement* : la maintenance, corrective ou évolutive, est effectuée dans le même cadre de développement que la construction, par itération et fabrication de nouveaux prototypes. La maintenance est une évolution normale du produit, similaire à la construction. Les risques de dégradation de l'architecture et de la conception s'en trouvent fortement réduits.

Exemples de risques

Tous les projets rencontrent des risques dont le nombre varie selon la taille, la complexité, le degré de nouveauté du projet et les qualifications de l'équipe de développement.

Certains risques sont fréquents, du fait de la naïveté des participants, des relations faussées entre client et fournisseur, de l'instabilité des besoins, etc.

Les grandes causes d'échec des projets sont souvent liées :

- dans le cas de très grands projets, à la situation géopolitique bien plus qu'aux aspects techniques,
- à l'absence d'une réelle vision de l'architecture, de sorte que le logiciel n'est pas intégrable,
- aux mauvaises relations au sein de l'équipe de développement,
- à l'absence de motivation, du fait d'échéances trop lointaines ou mal définies,
- au manque de visibilité sur le projet, à la fois des développeurs qui ne perçoivent pas l'image globale, et de l'encadrement qui, sans informations précises, pilote dans le brouillard,
- à un manque de personnel qualifié, suffisamment formé aux diverses technologies mises en œuvre,
- à un excès de personnes dans le projet, principalement dans les grandes organisations, où le prestige du chef de projet se mesure au nombre de développeurs placés sous ses ordres,
- au manque d'implication de l'utilisateur dans le processus de développement,
- au manque de soutien de toute l'organisation, provoqué par le manque de conviction d'un groupe donné, comme l'assurance qualité ou l'encadrement intermédiaire,
- à la sous-estimation du budget de la formation et du tutorat, nécessaires à tout projet qui aborde de nouvelles technologies,

- à la sous-estimation des délais, ce qui cause souvent la suppression d'itérations intermédiaires, et par suite, le retour aux travers du cycle en cascade,
- à la sous-estimation de l'ampleur du projet, manifestée entre autres par l'instabilité des besoins,
- à la technologie sélectionnée, quelquefois par simple effet de mode,
- à la dépendance par rapport à d'autres développements, effectués en parallèle.

Le tableau suivant résume les dix principaux risques récurrents et les actions à entreprendre pour les réduire.

Risque	Impact	Résolution
Intégration trop complexe Absence de convergence	Qualité Coût Délais	Centrage sur l'architecture Middleware Développement itératif
Démotivation	Qualité Coût Délais	Développement itératif Objectifs proches Planification des besoins
Environnement de développement inadapté	Coût Délais	Investissement dans des outils intégrés
Utilisateurs défavorables	Coût Délais	Implication précoce des utilisateurs Démonstration de prototypes
Technologie trop complexe	Coût Délais	Centrage sur l'architecture Achat de compétences externes
Activités manuelles trop lourdes	Coût	Automatisation par des outils intégrés
Inadéquation des composants réutilisables achetés	Délais	Prototypes
Performances insuffisantes	Qualité	Indépendance par rapport au matériel
Excès de bureaucratie	Délais	Centrage sur les prototypes, pas sur la documentation
Fonctions inadaptées	Qualité	Prototypes

Figure 375 : Résumé des dix principaux risques récurrents et des actions à entreprendre pour les réduire.

Chaque organisation ou projet devrait définir une liste de critères pour déterminer quelles fonctionnalités doivent être développées dans une itération donnée. Le

tableau suivant décrit des exemples de critères suffisamment généraux pour être appliqués à tous les projets.

Critère	Description
Intérêt	Degré de nécessité de l'élément pour les utilisateurs
Coût de fabrication	Estimation de la difficulté de développement de l'élément par les développeurs
Nouveauté	L'élément est-il totalement inconnu ?
Interfaces nécessaires	De quoi l'élément a-t-il besoin ?
Politique	Comment sera perçue la présence ou l'absence de l'élément ?
Technologie	Les développeurs maîtrisent-ils les technologies nécessaires ?
Planification	L'élément conditionne-t-il la livraison de prototypes ?
Tremplin	L'élément facilite-t-il le développement d'autres éléments ?

Figure 376 : Exemples de critères d'appréciation des risques pour la détermination des itérations.

Constitution de l'équipe de développement

Il existe quelques rares informaticiens virtuoses, mais quelles que soient les qualités des individus, il arrive un moment où l'ampleur de la tâche à accomplir est telle que l'effort individuel ne suffit plus. Il convient alors de travailler de concert, de coordonner les efforts et de rechercher la performance collective à partir des capacités moyennes de chacun.

Le choix des personnes qui constituent une équipe de développement détermine fortement le déroulement du projet. Au-delà des aspects techniques, le succès d'un projet dépend en grande partie des facteurs humains. Un bon processus de développement permet précisément de retirer la quintessence d'une équipe de développement, de manière reproductible et contrôlée. Les membres d'une équipe efficace doivent d'une part être complémentaires, et d'autre part être bien conscients de leur rôle dans le processus de développement. Il appartient au chef de projet de mettre sur pied cette équipe de personnes, puis d'entretenir le moral des troupes pendant l'ensemble du développement.

De manière générale, un processus de développement de logiciel peut se décomposer en trois sous-processus :

- le développement informatique proprement dit,
- la logistique qui pourvoit aux besoins du développement informatique,

- l'interface avec le reste du monde qui isole le développement des perturbations externes.

Ivar Jacobson a montré comment les processus des entreprises peuvent se modéliser avec des cas d'utilisation[46]. La figure suivante utilise ce formalisme pour illustrer les trois sous-processus du développement.

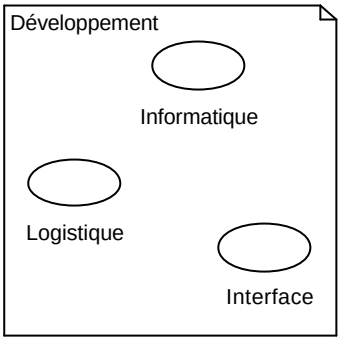


Figure 377 : Exemples de représentation des sous-processus du développement dans un diagramme de cas d'utilisation.

L'équipe de développement est organisée par rapport aux cas d'utilisation décrits précédemment. Les différents intervenants sont représentés par des acteurs qui interagissent avec les trois cas d'utilisation identifiés.

Le développement informatique

Le développement informatique est conduit par les trois acteurs suivants :

- *l'architecte* définit la forme générale du logiciel. Il est également responsable de la définition et de l'ordonnancement des itérations ;
- *les abstractionistes* (de l'anglais *abstractionist*) identifient les objets et les mécanismes qui permettront de satisfaire les besoins de l'utilisateur ;
- *les développeurs* maîtrisent les technologies et réalisent les abstractions identifiées par les abstractionistes.

[46] Jacobson I., Ericson M., Jacobson A. 1994, *The Object Advantage, Business Process Reengineering with Object Technology*. Addison Wesley.

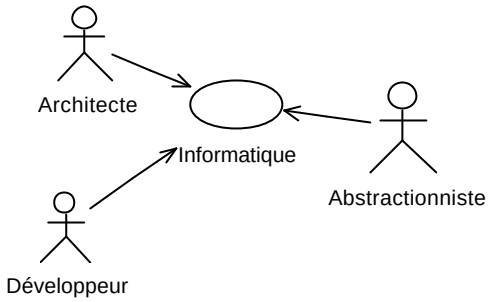


Figure 378 : Représentation des acteurs du développement informatique proprement dit.

La logistique

La logistique interagit avec les acteurs suivants :

- *le chef de projet* compose l'équipe, gère le budget et les personnes, et coordonne les diverses activités ;
- *l'analyste* est un expert du domaine, chargé de comprendre les besoins des utilisateurs ;
- *l'intégrateur* assemble les éléments produits par les développeurs, et ce durant le développement dans le cas d'un projet objet ;
- *le qualiticien* évalue tous les éléments produits par le développement et dirige les tests du système ;
- *le documentaliste* rédige la documentation à destination de l'utilisateur ;
- *l'administrateur-système* gère et entretient le parc matériel utilisé par le projet ;
- *L'outilleur* construit et adapte les outils logiciels qui forment l'environnement de développement ;
- *le bibliothécaire* assure l'archivage et la préservation de tous les artefacts du développement et de leurs règles de production.

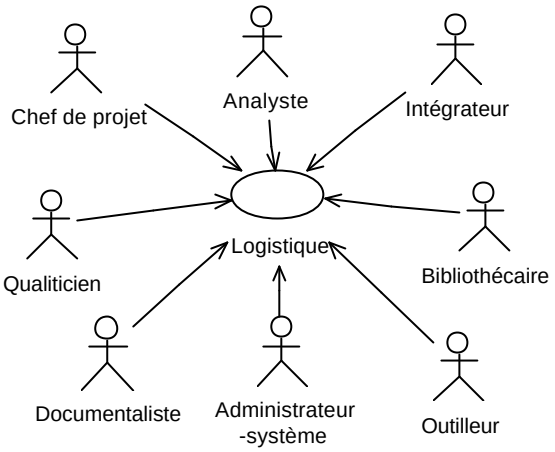


Figure 379 : Représentation des acteurs de la logistique.

L'interface

L'interface interagit avec les acteurs suivants :

- *le chef de projet* assure l'interface entre l'équipe de développement et le reste du monde ;
- *le chef de produit* supervise une ligne de produits ; il coordonne les activités de marketing, de formation et de support technique ;
- *le financier* contrôle l'allocation du budget et sa bonne utilisation ;
- *l'utilisateur/client* participe à des revues de prototypes ;
- *le support technique* résout ou contourne les problèmes rencontrés par les utilisateurs.

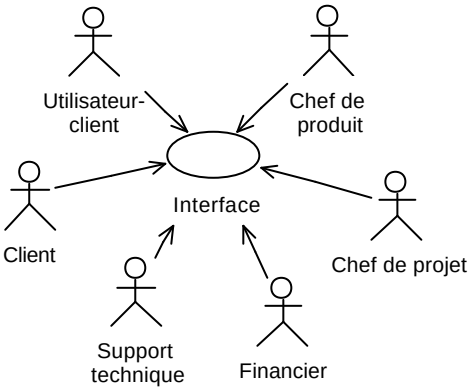


Figure 380 : Représentation des acteurs de l'interface.

Les rôles décrits précédemment peuvent être joués par la même personne. Dans les petites organisations, il est fréquent que le chef de projet remplisse également les rôles d'architecte et d'analyste. De même, les développeurs et les abstractionistes sont souvent confondus.

Description détaillée des phases

Ce paragraphe détaille les différentes phases du cycle de développement selon le point de vue général adopté par l'encadrement. Les phases apportent une vision structurée dans le temps, bien adaptée à la gestion des aspects contractuels et financiers.

L'étude d'opportunité

L'étude d'opportunité n'est généralement pas menée par les informaticiens, mais par des spécialistes de l'étude des marchés et de l'analyse de la concurrence. Ils essaient de déterminer si la construction d'un nouveau système ou une amélioration majeure d'un système existant est économiquement justifiée et présente un intérêt pour l'entreprise.

La première opération consiste à se doter d'une vision générale du produit afin de mieux le caractériser et de dégager des éléments d'appréciation. Cette vision peut se résumer par la formule suivante[47] :

Vision = Quoi + Pour qui + Combien

Les différentes composantes se définissent ainsi :

- *Quoi* exprime les grandes lignes du produit ;
- *Pour qui* détermine la population cible ;
- *Combien* estime le prix que les acheteurs du produit seront disposés à payer.

La vision générale du projet est exprimée dans un cahier des charges préliminaire, dans un cahier des non-objectifs qui précise ce que le projet n'est pas, et dans un argumentaire financier qui donne les premiers éléments économiques, par exemple en termes d'estimation du retour sur investissement.

Ces différents éléments peuvent être difficiles à apprécier en l'absence d'éléments de mesures tangibles. C'est pourquoi l'étude d'opportunité se base souvent sur une analyse préliminaire du domaine (10 à 20 % des classes). Lorsque les

[47] Morel E., communication privée.

incertitudes sur la faisabilité d'un projet sont élevées, il est fréquent de conduire un prototype conceptuel. Son but premier est de dégager des éléments concrets pour évaluer les risques liés aux fonctionnalités attendues, aux performances requises, à la taille ou à la complexité du sujet, ou encore à l'adoption de nouvelles technologies. A partir de là, le concept du produit est validé ou non.

Ce genre de prototype ne respecte pas les règles habituelles de développement. Il recherche avant tout un résultat rapide, sans insister sur la fiabilité ou la rapidité d'exécution. C'est avant tout une maquette jetable dont le code ne rentre pas dans l'application finale. Elle gagne à être réalisée dans un environnement de prototypage rapide, comme Smalltalk. Certaines applications sont développées intégralement à partir du premier prototype. Pour ce faire, des outils de développement rapide d'application (RAD) sont utilisés. Cette approche fonctionne pour des projets modestes, mais ne se prête guère à des systèmes plus conséquents.

A ce stade, l'équipe est constituée d'un petit nombre de personnes qui entourent l'architecte du système ou l'architecte du logiciel. Elles continueront de participer au développement du projet.

Par la suite, une fois que la vision du produit est définie et que les risques inhérents au projet ont été identifiés dans leurs grandes lignes, l'étude d'opportunité essaie d'estimer le coût du projet et le retour sur investissement.

L'estimation des coûts et le retour sur investissement

La détermination du retour sur investissement est assez difficile à effectuer dans le cas de l'informatique, en raison essentiellement de l'immatérialité des logiciels et du manque d'éléments objectifs pour apprécier a priori leur coût.

Il faut rechercher un équilibre entre la fiabilité de l'estimation des coûts et le délai nécessaire pour produire cette estimation. L'estimation fausse immédiate ou le coût exact connu à la fin du développement n'étant pas acceptables, il est judicieux de décomposer l'estimation en plusieurs étapes, afin de l'affiner progressivement.

La figure suivante représente, qualitativement, la précision de l'estimation du coût d'un développement logiciel.

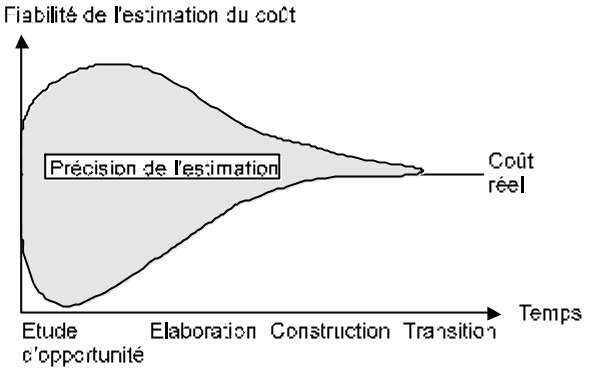


Figure 381 : Evolution de la précision de l'estimation du coût d'un projet.

Du fait du faible niveau de visibilité sur les besoins réels et sur la nature des risques durant l'étude d'opportunité, il est fort peu réaliste d'essayer d'estimer précisément les ressources nécessaires au projet.

Néanmoins, cette estimation est fréquemment demandée par le client. Il faut alors expliquer à celui-ci que la seule manière d'avoir une bonne idée du coût, c'est de financer une première tranche jusqu'à la fin de la phase d'élaboration. A la lumière de l'information récoltée, il pourra décider de poursuivre ou non l'effort de développement, avant d'aborder la montée en charge de la phase de construction.

L'approche objet seule ne permet pas d'évaluer le coût final plus rapidement que les approches traditionnelles. En revanche, associée à une démarche itérative, elle permet de garantir une détermination plus précise de ce coût à partir de la phase d'élaboration.

Pour un petit projet, l'étude d'opportunité est souvent réduite à un cahier des charges. Pour un projet moyen, d'une année environ, l'étude d'opportunité – fabrication du prototype incluse – prend approximativement un mois. Pour un gros projet, le prototype devient un petit projet à part entière, souvent dénommé démonstrateur, et passe lui-même par les différentes phases énoncées plus haut.

La phase d'élaboration

La phase d'élaboration commence par l'analyse des besoins et par la modélisation du domaine. Elle a pour objectif de définir les choix d'architecture, d'explorer et de réduire les risques du projet, et finalement de définir un plan complet qui quantifie les moyens à mettre en œuvre pour mener à bien le développement.

La phase d'élaboration est conduite par une équipe restreinte, emmenée par l'architecte logiciel. Elle est constituée d'un petit groupe de développeurs et d'un ou deux experts du domaine ou d'utilisateurs. Il est souvent judicieux d'adjoindre un représentant de l'équipe de test et d'assurance qualité, un outilleur

(responsable de la mise en place de l'environnement de développement) et un documentaliste (responsable de la rédaction de la documentation).

Activités conduites en phase d'élaboration

L'analyse des besoins est basée principalement sur l'étude des cas d'utilisation, sans pour autant exclure toutes les autres techniques qui pourraient aider les utilisateurs à articuler puis à formuler leurs désirs.

Les cas d'utilisation sont exprimés dans la terminologie des utilisateurs, sous une forme textuelle, loin de tout formalisme informatique. La transition vers une représentation plus informatique est effectuée par les analystes. Ils transforment les cas d'utilisation en collaborations entre objets du domaine de l'application. Une collaboration reste compréhensible par les utilisateurs. Il faut préalablement leur expliquer que les objets représentent une entité du monde réel, autrement dit de leur monde, et que ces objets collaborent pour construire les fonctions représentées par les cas d'utilisation.

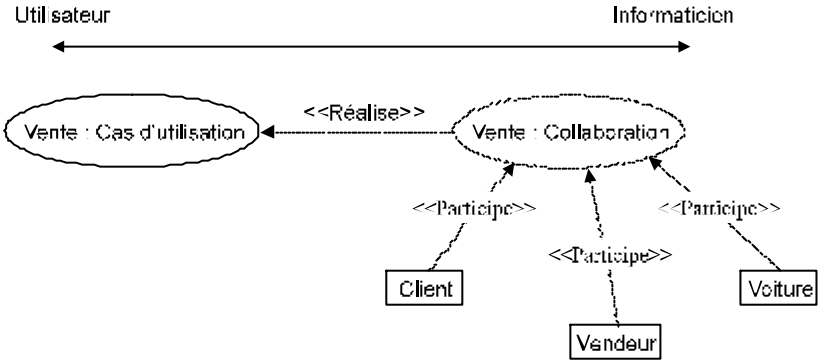


Figure 382 : Exemple de réalisation d'un cas d'utilisation par une collaboration entre trois objets.

Au fur et à mesure de l'étude détaillée des cas d'utilisation, le modèle du domaine est mis à jour pour prendre en compte les nouvelles classes identifiées dans les collaborations qui servent de support aux différentes interactions. Une deuxième passe d'analyse, après l'analyse du domaine, recherche des abstractions et des scénarios plus généraux. Elle dégage des classes abstraites qui serviront de spécifications pour l'écriture de mécanismes génériques.

La phase d'élaboration donne naissance aux produits suivants :

- la description du comportement du système, exprimée sous la forme de cas d'utilisation, le contexte du système, les acteurs, les scénarios et un modèle des classes du domaine (80 % des classes) ;

- une architecture exécutable, un document de description de l'architecture et une révision de la vision du projet et de la description du risque ;
- un plan de développement complet pour l'ensemble du projet ;
- un plan détaillé des itérations, les critères d'évaluation et le cahier des charges de chaque itération et les résultats de l'assurance qualité ;
- éventuellement un manuel utilisateur préliminaire.

Pour les projets de taille moyenne, il y a typiquement :

- quelques dizaines de cas d'utilisation ;
- une centaine de scénarios principaux;
- quelques centaines de scénarios secondaires ;
- entre 50 et 100 classes dans le domaine.

Conseils pratiques sur l'élaboration

N'importe quelle application devrait pouvoir récupérer suffisamment de composants réutilisables pour réaliser de 10 à 30 % de son implémentation. Les frameworks et les solutions clés en main permettent d'envisager un taux de réutilisation de l'ordre de 60 % pour un même type d'application et un même domaine.

La durée de la phase d'élaboration est donc très variable. Elle dépend beaucoup des types d'application et des choix d'infrastructure. Pour un projet d'un an, cette phase dure entre deux et quatre mois.

Dans tous les cas, il faut impérativement éviter la paralysie par l'analyse, c'est-à-dire de perdre trop de temps à rechercher une analyse parfaite. Il vaut mieux se jeter à l'eau (c'est-à-dire passer à la conception de l'architecture) lorsqu'environ 80 % des scénarios principaux ont été examinés.

Lors de l'étude des scénarios, il ne faut pas non plus aller trop loin dans les détails au risque de s'enfoncer trop tôt dans la conception et par suite de manquer d'abstraction dans les solutions mises en œuvre.

La phase de construction

Cette phase a pour objectif de développer un produit logiciel prêt pour la transition dans la communauté des utilisateurs. Dans les cycles itératifs en b, la phase de construction correspond à la mise en œuvre effective des itérations. Certaines itérations sont purement internes, d'autres sont également visibles à l'extérieur du projet. Chaque itération produit la livraison d'un prototype exécutable.

Un prototype force la fin d'une itération et oblige l'équipe de développement à fournir un résultat concret, mesurable. Un prototype est stable et autosuffisant ;

c'est une version exécutable du système, conduite comme un petit projet et accompagnée de tous les produits nécessaires pour son exploitation.

La réalisation d'une itération regroupe les activités suivantes :

- identification des scénarios à compléter ou à reprendre dans l'itération, en fonction de l'étude des risques et des résultats de l'itération précédente ;
- assignation de tâches précises à l'équipe de développement pour arriver au bout de l'itération ;
- définition des critères d'évaluation de chaque itération, des points de contrôle et des délais ;
- rédaction de la documentation pour l'utilisateur et de la documentation de déploiement (notes de livraison, notes d'installation).

Le flot de livraisons se caractérise par :

- une augmentation régulière des fonctionnalités, mesurée par le nombre de nouveaux scénarios pris en compte par chaque itération ;
- une plus grande profondeur, mesurée par le degré de réalisation du modèle du domaine et des mécanismes ;
- une plus grande stabilité, mesurée par la réduction dans les changements apportés aux modèles ;
- l'ajustement continu du plan du projet ;
- l'évaluation de chaque itération, avec les résultats des tests et de l'assurance qualité.

Avec la montée en charge de la construction, l'équipe est au complet. Dans les grands projets, plusieurs incréments peuvent être développés en parallèle. En règle générale, 80 % de l'équipe travaille au développement d'une livraison, alors que les 20 % restant explorent les nouveaux risques et préparent la mise en chantier des livraisons suivantes.

Pour un projet modeste, cette phase dure entre 2 et 3 mois ; pour un projet typique, elle dure entre 6 et 9 mois en moyenne. La durée totale des projets objet est aujourd'hui d'une année, avec une petite équipe de 5 ou 6 développeurs. Dans le cas de très gros projets, chaque sous-système se comporte comme un projet et possède alors ses propres itérations, au sein d'un cycle de vie itératif en O.

La gestion du changement

Dans un cycle de vie itératif, les artefacts de développement mûrissent avec les itérations. Le projet est donc confronté à un important problème de gestion de cohérence, du fait des nombreuses versions de chaque élément. Cette gestion de la cohérence peut être assurée avec succès et à un coût raisonnable uniquement

si des outils de gestion de versions et de configurations sont totalement intégrés dans le processus de développement.

Dans les projets modestes, la gestion des changements concerne principalement le code et la documentation. Dans les projets plus ambitieux, il faut se préoccuper de la gestion du changement dans l'ensemble de l'environnement de développement, depuis les versions des logiciels, comme les compilateurs ou les systèmes d'exploitation, jusqu'au matériel lui-même. Dans les cas extrêmes, par exemple pour certains projets militaires ou aéronautiques, l'ensemble de l'environnement de développement (logiciel et matériel) est conservé, sans aucune modification, pendant toute la durée de vie du système !

Les principaux types de changements apportés au logiciel sont les suivants :

- l'ajout de classe ou de collaboration avec une classe ;
- le changement de l'implémentation des opérations d'une classe ;
- le changement de la représentation d'une classe ;
- la réorganisation de la structure de classes ;
- le changement de l'interface d'une classe.

Les trois premiers types de changements sont fréquents dans la phase de construction. Dans les dernières itérations, les deux derniers types indiquent que la conception n'est pas stable.

Avec le temps, les changements doivent être de plus en plus localisés (du fait de l'encapsulation). Les changements d'architecture sont les plus coûteux en terme de retouches.

Conseils pratiques sur la construction

Chaque livraison donne aux développeurs une satisfaction liée à la notion de travail accompli. Les livraisons constituent des sous-objectifs relativement faciles à atteindre, dans un délai raisonnable par rapport à la durée totale du projet. De ce fait, le moral de l'équipe reste au beau fixe, même si la démarche par prototypes oblige à s'attaquer aux problèmes dès le départ.

Chaque livraison fournit également à l'encadrement un point de contrôle objectif sur l'état d'avancement du projet. Chaque prototype est un pas en avant vers la satisfaction des besoins.

Enfin, chaque livraison redonne confiance aux utilisateurs qui reçoivent par l'examen du prototype un retour tangible, dont ils peuvent directement transposer les bénéfices dans leur travail quotidien. Ceci suppose de ne pas jeter une livraison en pâture aux utilisateurs, sans avoir au préalable défini avec eux la portée du prototype. Il faut impérativement que les utilisateurs comprennent

qu'un prototype est une étape, pas un système complet, sinon les frustrations peuvent être grandes.

En cas d'urgence, il vaut mieux reporter des fonctionnalités que manquer un rendez-vous avec les utilisateurs. Tout prototype démontrable, même incomplet, augmente le niveau de confiance des utilisateurs, et par suite du client. Il faut toutefois prendre garde à *l'effet congère* qui se caractérise par une accumulation de fonctionnalités non satisfaites, toujours remises à la prochaine livraison. Il faut essayer de tenir le plan des livraisons car la tendance naturelle est de repousser les livraisons en raison du retard pris par rapport à la réalisation des fonctionnalités. En repoussant la fin des itérations, et par suite la livraison des prototypes, le danger est grand de revenir au modèle en cascade et de perdre les avantages de la démarche itérative.

Les catégories permettent d'allouer les responsabilités dans l'équipe. Les petits changements sont pris en compte à l'intérieur d'une classe ou d'une catégorie. Les grands changements demandent la coordination entre les responsables de chaque catégorie et une remontée de la décision au niveau de l'architecte. Il est normal de devoir jeter du code en phase de construction. En revanche, au-delà de 25 % de changements cumulés dans les livraisons de construction, l'architecture doit être considérée comme instable. L'instabilité de l'architecture est généralement le signe d'une phase d'élaboration qui n'a pas donné une place suffisante à l'expérimentation.

La phase de transition

La phase de transition consiste à transférer le logiciel dans la communauté de ses utilisateurs. Cette phase est d'une complexité variable selon le type d'application ; elle comprend la fabrication, l'expédition, l'installation, la formation, le support technique et la maintenance. L'équipe de développement se réduit à un petit groupe de développeurs et testeurs, assistés à temps partiel par l'architecte, garant de la cohérence de l'ensemble, et par un documentaliste responsable de la mise à jour de la documentation. Le relais est passé au personnel du support technique, ainsi qu'aux formateurs et aux commerciaux.

En phase de transition, le développement initial est presque terminé ; tous les artefacts ont atteint suffisamment de maturité pour être distribués largement vers deux catégories de destinataires : les utilisateurs et les responsables du projet.

Les principaux produits à destination des utilisateurs comprennent :

- des programmes exécutables, les versions bêta puis définitives ;
- les manuels pour l'utilisateur ;
- la documentation de déploiement et d'installation.

Les principaux produits à destination des responsables du projet regroupent :

- les modèles révisés ;

- les critères d'évaluation de chaque itération ;
- la description des livraisons ;
- les résultats de l'assurance qualité ;
- les résultats de l'analyse post-mortem des performances du projet.

La durée de la phase de transition est très variable selon le critère de fin de phase. Si la fin est indiquée par le client au cours d'un processus de recette, la transition dure au maximum un mois pour un projet d'une année. En revanche, si la fin du projet correspond à la fin de la vie du projet, la transition peut être beaucoup plus longue.

Conseils pratiques sur la transition

La difficulté de cette phase est inversement proportionnelle à la qualité du produit et au degré de préparation des utilisateurs. Tous les produits nécessitent une formation. Il ne faut pas négliger la phase d'installation du produit ; une installation difficile peut décourager les utilisateurs et réduire leur confiance dans le produit.

Pour un système de remplacement, beaucoup d'efforts sont nécessaires pour mettre en place le nouveau système en parallèle avec le système existant.

Dans certaines organisations, l'équipe de maintenance et l'équipe de développement sont nettement dissociées. Cette situation n'est pas optimale car elle engendre une rupture trop nette dans le cycle de développement. Elle induit surtout un moment de flottement dans le partage des responsabilités, entre les développeurs initiaux et les développeurs de maintenance. Il est plus judicieux d'assurer un recouvrement entre les deux équipes, en incluant des personnes du support technique et de la maintenance dès le début du projet et en assignant à temps partiel quelques développeurs initiaux dans l'équipe de maintenance.

Cycles de développement post-déploiement

Pour la plupart des produits logiciels, le déploiement marque le début d'une période de maintenance et d'amélioration. Cette période se caractérise par la répétition du cycle de développement, de l'étude d'opportunité jusqu'à la transition. L'importance relative de chacune de ces activités peut changer. Ainsi, dans un simple cycle de maintenance, il n'y a pas normalement d'impact sur l'architecture, tant que les modifications restent bien encapsulées.

Chaque itération démarre en établissant une liste de priorités de réparation et en désignant les équipes de développement. Elle se termine par la livraison d'une version du système qui corrige les défauts identifiés précédemment. La transition inclut la mise à jour de la documentation à destination des utilisateurs et la description des défauts corrigés.

Conseils pratiques sur le post-déploiement

Il faut impérativement corriger les défauts afin de satisfaire les attentes légitimes des utilisateurs. Il faut essayer en plus de satisfaire leurs nouveaux besoins, éventuellement au moyen de nouvelles technologies : un logiciel qui n'évolue plus devient sans intérêt. Pour ce faire, il faut se focaliser sur la conservation de l'intégrité de l'architecture sinon l'entropie prend le dessus : petit à petit, le coût des changements augmente, la conception se complique et la maintenance devient de plus en plus difficile.

Dans tous les cas, quelles que soient les mesures prises, les résolutions affichées et la bonne volonté des équipes de développement et de maintenance, il arrivera un temps ou le logiciel devra être jeté et remplacé par un autre. Les logiciels portent en eux les raisons de leur mort, et leur espérance de vie...

Répartition des activités par rapport aux phases

L'importance de chaque activité de développement varie d'une phase à une autre. Il n'y a pas de correspondance directe entre les phases de la vue de l'encadrement et les phases classiques d'un cycle en cascade. Au contraire, les activités comme l'analyse et la conception sont étalées sur plusieurs phases de la vue de l'encadrement. Les activités se distribuent sur les cycles et leur début et fin ne correspondent pas aux limites des phases de la vue de l'encadrement.

En fait, le point important est la densité d'activité réalisée dans chaque phase de la vue de l'encadrement.

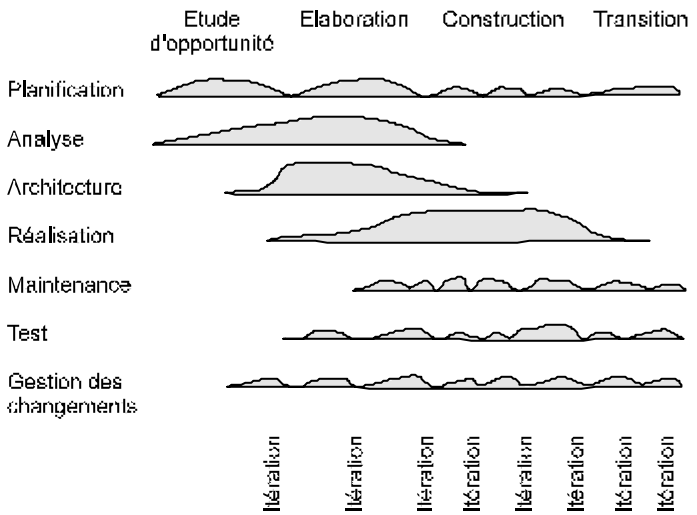


Figure 383 : Répartition typique des activités au sein des phases de la vue de l'encadrement.

La figure précédente montre que :

- la planification est effectuée tout au long des phases ;
- l'analyse est concentrée principalement dans la phase d'élaboration, bien qu'une part d'analyse puisse déjà être conduite en phase d'étude d'opportunité et que l'analyse soit affinée en phase de construction ;
- la principale partie de l'architecture est déterminée dans la phase d'élaboration ;
- la réalisation (qui comprend également les tests unitaires) commence en phase d'élaboration (pour les éléments d'architecture) et culmine en phase de construction ;
- la maintenance débute dès que la première version du logiciel a été définie, généralement en phase d'élaboration ;
- les tests et les mesures de qualité se répartissent sur toutes les phases et concernent tous les prototypes ;
- la gestion des changements (versions et configurations) enregistre l'histoire de tout le projet.

Il n'y pas de phase d'intégration proprement dite : l'intégration est continue, le logiciel est intégré par construction.

Effort relatif par activité

Le diagramme suivant illustre une situation typique, représentative d'un projet objet moyen (d'une année environ, avec une petite équipe d'environ cinq personnes). Il décrit une répartition typique de l'effort par rapport aux activités.

Dans le but de répartir l'effort selon les différentes activités, les postulats suivants sont établis :

- la planification inclut les activités de pilotage de projet, les plans de développement, la mesure des progrès et le contrôle des ressources du projet ;
- l'analyse comprend le développement des modèles objet, des modèles utilisateurs, la spécification de la vision du projet et la description des critères d'évaluation ;
- l'architecture inclut la réalisation des fondements du logiciel, la conception générale de toute l'application et la construction des mécanismes communs ;
- l'implémentation regroupe toutes les activités de conception détaillée, le codage et les tests unitaires ;
- la maintenance comprend les changements apportés au logiciel, une fois placé sous gestion de versions et de configurations ;

- les tests et l'évaluation comprennent les activités nécessaires pour démontrer que le logiciel atteint les objectifs fixés et mesurés par les critères ;
- la gestion du changement mémorise les états consécutifs de tous les artefacts de développement.

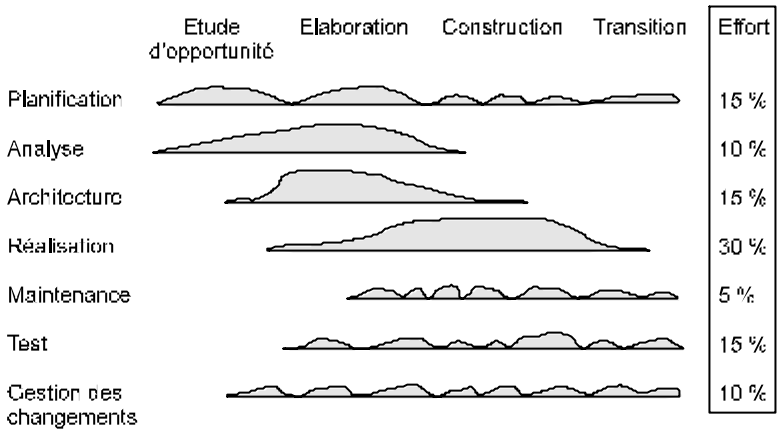


Figure 384 : Répartition typique de l'effort par rapport aux activités.

Effort relatif par phase

L'effort est réparti différemment selon les phases et cette répartition peut varier considérablement avec les caractéristiques du projet. Le diagramme suivant montre une situation typique, représentative d'un projet de taille moyenne. La fin de la phase de transition est marquée par la disponibilité générale du produit.

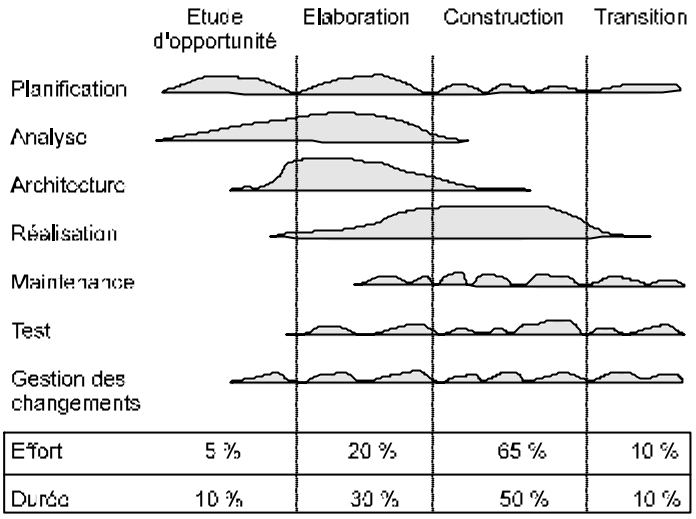


Figure 385 : Répartition typique de l'effort par rapport aux phases.

Pour un cycle de maintenance, les phases d'études d'opportunité et d'élaboration sont considérablement réduites. L'emploi d'outils de génération de code ou de construction d'application peut fortement réduire la phase de construction ; dans certains cas elle peut être plus courte que les phases d'étude d'opportunité et d'élaboration mises bout à bout.

Points de recouvrement des phases et points de décision

La fin de chaque phase est marquée par une décision planifiée qui correspond à de grandes décisions dans la conduite des affaires de l'entreprise. Pendant la phase d'étude d'opportunité, la viabilité économique et la faisabilité technique du produit sont établies. A la fin de l'étude d'opportunité, la décision d'allouer ou non des ressources pour l'élaboration doit être prise. A la fin de l'élaboration, un point de décision similaire est atteint. En fonction du travail d'architecture déjà réalisé et en fonction de l'évaluation des risques subsistants, il faut allouer ou non les ressources pour terminer le produit.

La construction s'achève lorsque le produit est suffisamment mûr pour que des utilisateurs puissent s'en servir de manière fiable.

La transition se termine soit lorsque le produit est remplacé par une nouvelle génération, soit lorsque le produit arrive en fin de vie et n'est plus utilisé.

Parallèlement au déroulement des phases, la fin de chaque itération apporte un éclairage sur les progrès techniques et la maturité du projet. L'évaluation des itérations détermine l'étendue des progrès et, dans la foulée, entraîne les réévaluations du coût global, de la planification et du contenu du projet.

Le tableau suivant résume les objectifs et les points de décisions rapportés aux itérations de la vue technique et aux phases de la vue de l'encadrement.

Itération	Phase	Objectif	Point de décision
Itération préliminaire	Etude d'opportunité	Comprendre le problème	
			Allouer les ressources pour l'élaboration
Itération d'architecture	Elaboration	Comprendre la solution	
Itération d'architecture	Elaboration	Comprendre la solution	
			Allouer les ressources pour la construction
Itération de développement	Construction	Réaliser la solution	
Itération de développement	Construction	Réaliser la solution	
Itération de développement	Construction	Réaliser la solution	
			Livrer le produit
Itération de transition	Transition	Transmettre la solution	
Itération de transition	Transition	Transmettre la solution	
			Recette par le client

Figure 386 : Tableau récapitulatif des objectifs et des points de décisions rapportés aux itérations et aux phases.

Le tableau suivant récapitule les étapes, les points de mesure et les résultats de chaque phase.

Phase	Etape	Mesure	Résultat
Etude d'opportunité	Plan marketing Prototype exécutable	Complétude d'un prototype Risque	Lancement effectif du projet
Elaboration	Modèle des cas d'utilisation Modèle du domaine Architecture	Stabilité Simplicité	80 % des scénarios 80 % du modèle du domaine
Construction	Prototypes Plan de déploiement	Complétude des livraisons Taux de défauts Densité de défauts Stabilité	Produit complet Documentation complète Qualité satisfaisante
Transition	Versions bêta	Satisfaction des utilisateurs Taux de défauts Densité de défauts Stabilité	Version définitive

Figure 387 : Tableau récapitulatif des étapes, des points de mesure des résultats de chaque phase.

5

Etude de cas : application de contrôle d'accès à un bâtiment

Ce chapitre présente l'utilisation d'UML pour la modélisation objet d'un système de contrôle des accès à un bâtiment.

L'étude de cas décrite dans ce paragraphe est inspirée d'une application déployée à l'ESSAIM (Ecole Supérieure des Sciences Appliquées pour l'Ingénieur – Mulhouse).

Le modèle de cet exemple – réalisé avec l'outil Rational Rose – est disponible sur le CD-ROM qui accompagne l'ouvrage.

Le processus

L'étude de cas débute directement par l'analyse des besoins. Les besoins du système sont déterminés à partir de l'information recueillie durant l'interview du superviseur du futur système. Ces besoins sont exprimés sous la forme de cas d'utilisation, dans un langage très proche des utilisateurs.

L'étude des cas d'utilisation demande de choisir le niveau de granularité des informations représentées dans les interactions ; ce qui pose souvent le problème *du petit rocher ou du gros caillou*. Dans l'exemple suivant, les cas d'utilisation sont délibérément considérés comme des éléments à forte granularité.

Chaque cas d'utilisation regroupe plusieurs scénarios décrits d'abord de manière générale, du point de vue de l'acteur, puis représentés de manière plus informatique afin de permettre une évaluation des coûts de réalisation de l'application.

La structure statique – exprimée par les relations entre les classes des objets qui participent aux collaborations – est représentée dans des ébauches de diagrammes de classes, puis finalisée dans un diagramme de classes qui représente les abstractions clés du domaine et leurs relations.

L'analyse de l'existant étudie les caractéristiques des lecteurs de badges mis en œuvre et permet de dégager une stratégie pour la réalisation des cas d'utilisation.

L'exemple se termine par la description de l'architecture logicielle et matérielle de l'application.

Analyse des besoins

Les espaces à protéger se répartissent sur 4 niveaux au sein d'un bâtiment d'une surface totale d'environ 5 000 m^2. Le bâtiment est divisé en cinq zones : deux ailes de recherche, une aile de travaux pratiques, une aile pour l'administration et un corps central qui abrite les salles de cours et les deux amphithéâtres.

Le site accueille environ 500 personnes tous les jours, en majorité des étudiants, mais aussi des enseignants, des chercheurs, du personnel administratif et technique, ainsi que de nombreux visiteurs.

Suite à la disparition d'objets divers, il a été décidé de restreindre les accès à certaines salles au moyen de portes à fermeture automatique. L'ouverture de chacune de ces portes est commandée par un lecteur de badges placé à proximité.

Les badges qui permettent l'ouverture des portes ne sont délivrés qu'aux personnes qui doivent accéder aux locaux protégés dans l'exercice de leurs activités. Les droits d'accès sont alloués entre les groupes de personnes et les groupes de portes, de sorte qu'une personne ou une porte doit toujours être au moins dans un groupe (le sien).

Un groupe de portes peut contenir des portes dispersées dans tout le bâtiment. Du point de vue du contrôle d'accès, seule la notion de groupe de portes est importante : les chemins et les déplacements ne sont pas contrôlés. Une porte donnée ne peut appartenir qu'à un seul groupe de portes.

Une même personne peut appartenir à plusieurs groupes, de sorte que ses droits d'accès correspondent à l'union des droits d'accès de chacun des groupes qui la contiennent.

La définition des droits d'accès est effectuée en décrivant pour chaque groupe de personnes les différents groupes de portes qui sont accessibles et sous quelles contraintes horaires. Les droits d'accès sont décrits dans un calendrier annuel qui décrit la situation semaine par semaine. Etant donné la faible variation des droits dans le temps, un calendrier peut être initialisé au moyen de semaines types qui décrivent une configuration de droits donnée. Le superviseur peut créer autant de semaines types qu'il le désire. Les changements apportés à une semaine type sont automatiquement propagés dans tous les calendriers qui utilisent cette semaine type. Les changements apportés directement dans un calendrier, par exemple pour prendre en compte un jour férié, ne sont pas affectés lors de la modification d'une semaine type.

La figure suivante représente une semaine type. Les parties en grisé correspondent aux plages horaires pendant lesquelles l'accès n'est pas autorisé.

		Lundi	Mardi	Mercredi	Jeudi	Vendredi	Samedi	Dimanche
00	01							
01	02							
...	...							
06	07							
07	08							
08	09							
...	...							
21	22							
22	23							
23	24							

Figure 388 : Exemple de semaine type.

Le système de contrôle d'accès doit fonctionner de la manière la plus autonome possible. Un superviseur est responsable de la configuration initiale et de la mise à jour des différentes informations de définition des groupes de personnes et de portes. Un gardien dispose d'un écran de contrôle et est informé des tentatives de passage infructueuses. Les alarmes sont transmises en temps légèrement différé : la mise à jour de l'information sur l'écran de contrôle est effectuée toutes les minutes.

L'interface utilisateur doit aider l'utilisateur à formuler des requêtes correctes. Les valeurs de paramètres doivent systématiquement être lues dans des listes qui définissent le domaine des valeurs correctes.

Description des cas d'utilisation

Les cas d'utilisation sont décrits de manière textuelle, agrémentée de quelques diagrammes d'interaction. A ce stade de la modélisation, les interactions représentent les principaux événements qui se produisent dans le domaine de l'application. Plus tard, lors de la conception, ces événements sont traduits en messages qui déclenchent des opérations.

Détermination des cas d'utilisation

L'analyse débute par la recherche des acteurs (catégories d'utilisateurs) du système de gestion des accès. Un acteur représente un rôle joué par une personne ou par une chose qui interagit avec le système. Il n'est pas toujours facile de déterminer la limite du système. Par définition, les acteurs sont à l'extérieur du système.

Les acteurs se recrutent parmi les utilisateurs du système et aussi parmi les responsables de sa configuration et de sa maintenance. Ils se répartissent dans les catégories suivantes :

- superviseur,
- gardien,
- porteur de badge.

Voici leur représentation graphique :

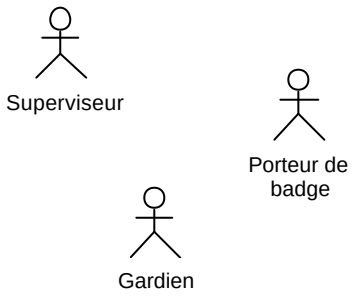


Figure 389 : Représentation des catégories d'utilisateurs.

Les acteurs interagissent avec le système. L'étude des cas d'utilisation a pour objectif de déterminer ce que chaque acteur attend du système. La détermination des besoins est basée sur la représentation de l'interaction entre l'acteur et le système. Cette approche présente l'avantage de forcer l'utilisateur à définir précisément ce qu'il attend du système.

Un cas d'utilisation est une abstraction d'une partie du comportement du système. Le cas d'utilisation est instancié à chaque utilisation du système par une instance d'un acteur.

Après interview des utilisateurs, il ressort que les catégories de besoins fonctionnels des acteurs se décomposent de la manière suivante :

Acteur	Cas d'utilisation
Superviseur	Configuration du système
Gardien	Exploitation du système
Utilisateur	Validation d'une demande d'accès

Figure 390 : Récapitulatif des cas d'utilisation du système de contrôle d'accès.

Le diagramme suivant représente les cas d'utilisation du système de contrôle d'accès.

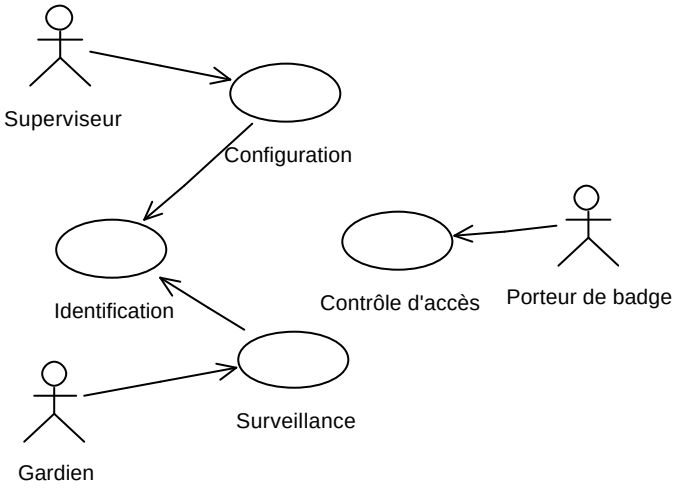


Figure 391 : Diagramme des cas d'utilisation du système de contrôle d'accès.

Configuration

La configuration est déclenchée par le superviseur.

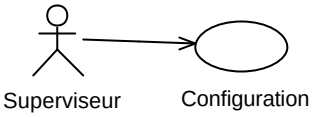


Figure 392 : Déclenchement de la configuration par le superviseur.

Identification

Le superviseur se connecte au système et donne son mot de passe.

Le système vérifie l'identité du superviseur et autorise la connexion.

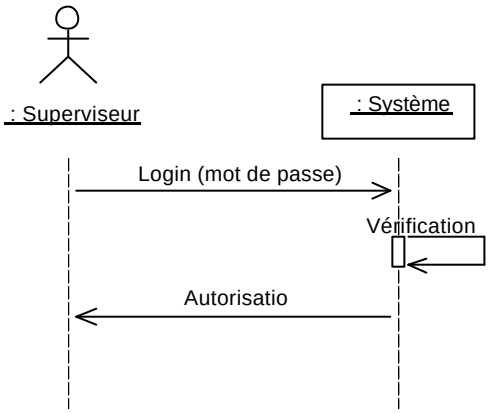


Figure 393 : Identification du superviseur.

Modification des portes

Le superviseur demande la liste des portes ou des groupes de portes.

Le système affiche la liste des portes ou des groupes de portes.

Le superviseur choisit une porte ou un groupe de portes.

Le système affiche les informations suivantes :

l'état (activé / désactivé),

la durée de la temporisation d'ouverture,

la confidentialité (un seul dedans, les autres dehors).

Le superviseur modifie les informations précédentes.

Le système enregistre les informations.

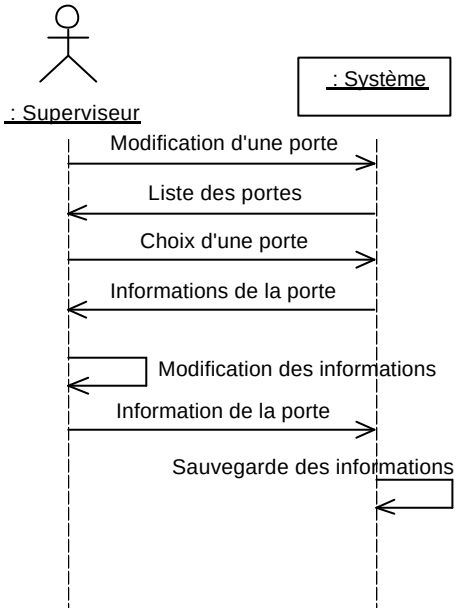


Figure 394 : Modification des portes.

Modification d'une personne

Le superviseur demande la liste des personnes.

Le système affiche toutes les personnes déjà recensées.

Le superviseur choisit une personne.

Le système affiche les informations suivantes :

- le nom de la personne,
- le prénom de la personne,
- le numéro de téléphone de la personne,
- le numéro de son badge,
- la liste des groupes auxquels appartient la personne.

Le superviseur modifie les informations précédentes.

Le système enregistre les informations.

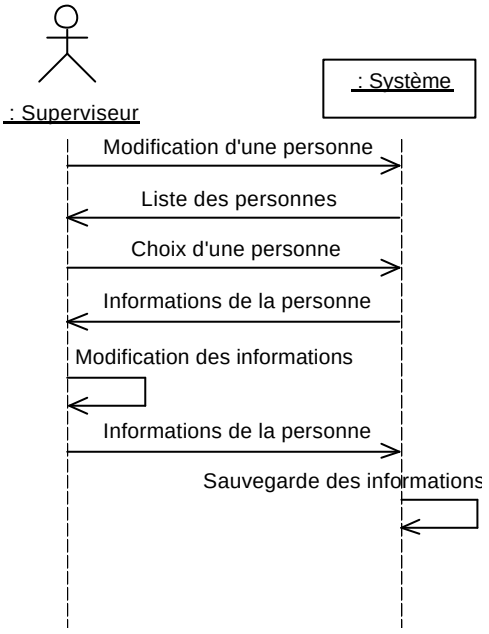


Figure 395 : Modification des personnes.

Modification des groupes de personnes

Le superviseur demande la liste des groupes de personnes.

Le système affiche tous les groupes de personnes déjà recensés.

Le superviseur choisit un groupe de personnes.

Le système affiche les informations suivantes :

le nom du groupe,

la liste des membres du groupe,

la liste des accès possibles pour le groupe.

Le superviseur modifie les informations précédentes.

Le système enregistre les informations.

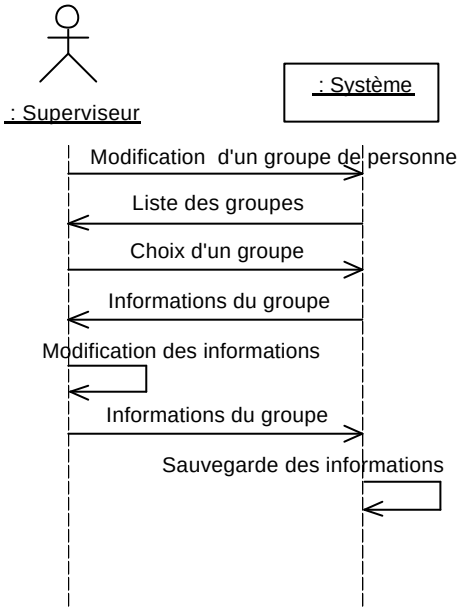


Figure 396 : Modification des groupes de personnes.

Modification des groupes de portes

Le superviseur demande la liste des groupes de portes.

Le système affiche tous les groupes de portes déjà recensés.

Le superviseur choisit un groupe de portes.

Le système affiche les informations suivantes :

- le nom du groupe,
- la liste des portes contenues par le groupe,
- la liste des accès alloués à des groupes de personnes.

Le superviseur modifie les informations précédentes.

Le système enregistre les informations.

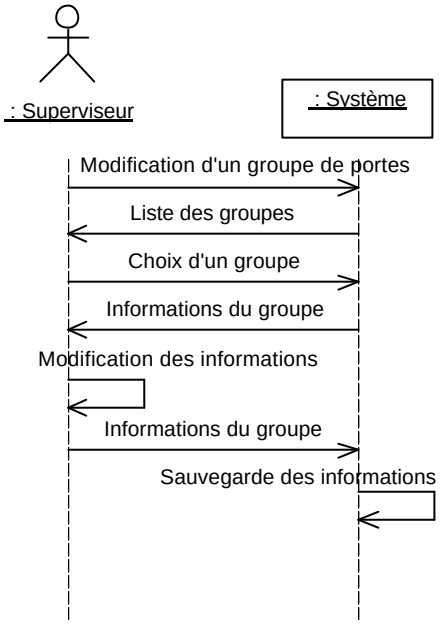


Figure 397 : Modification des groupes de portes.

Recherche d’une personne en fonction d’un badge

Le superviseur donne le numéro d’identification du badge.

Le système retrouve la personne à qui le badge a été alloué.

Le système affiche les informations suivantes :

le nom de la personne,

le prénom de la personne,

le numéro de téléphone de la personne,

le numéro de son badge,

la liste des groupes auxquels appartient la personne.

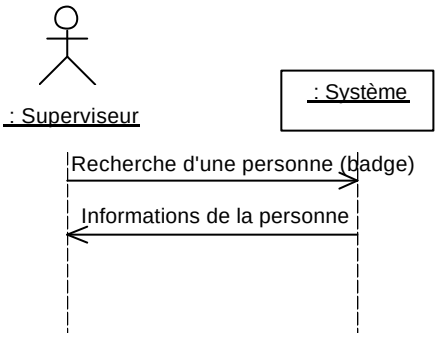


Figure 398 : Recherche d’une personne en fonction d’un badge.

Recherche des portes franchissables par une personne donnée

Le superviseur demande la liste des personnes.

Le système affiche toutes les personnes déjà recensées.

Le superviseur choisit une personne.

Le système affiche la liste des portes franchissables.

Le superviseur choisit une porte.

Le système affiche les informations suivantes :

l’état (activé / désactivé),

la durée de la temporisation d’ouverture,

la confidentialité (un seul dedans, les autres dehors),

l’accès alloué à la personne.

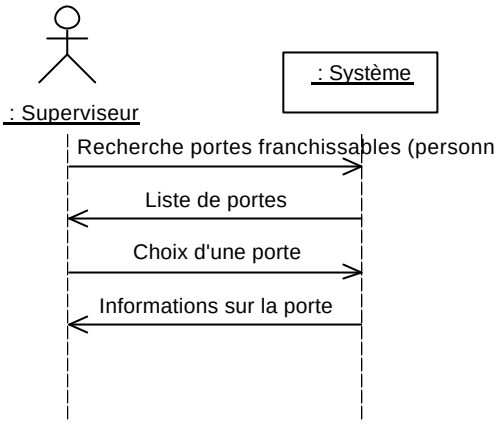


Figure 399 : Recherche des portes franchissables par une personne donnée.

Recherche des groupes qui contiennent une personne donnée

Le superviseur demande la liste des personnes.

Le système affiche toutes les personnes déjà recensées.

Le superviseur choisit une personne.

Le système affiche la liste des groupes qui contiennent la personne.

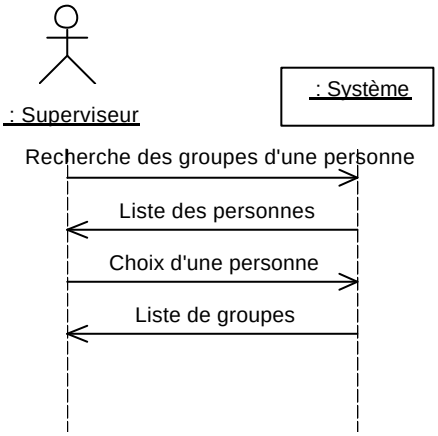


Figure 400 : Recherche des groupes qui contiennent une personne donnée.

Recherche des personnes qui appartiennent à un groupe donné

Le superviseur demande la liste des groupes de personnes.

Le système affiche tous les groupes déjà recensés.

Le superviseur choisit un groupe.

Le système affiche la liste des personnes contenues par le groupe.

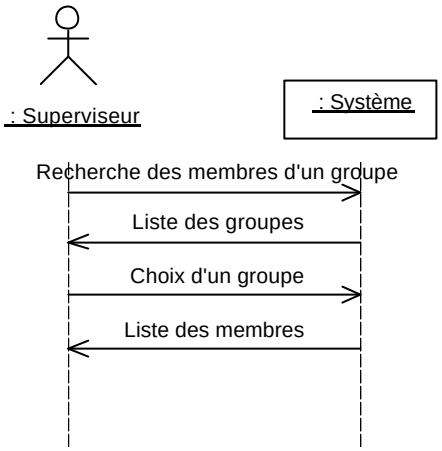


Figure 401 : Recherche des personnes qui appartiennent à un groupe donné.

Modification des accès d'un groupe de personnes à un groupe de portes

Le superviseur demande la liste des groupes de personnes.

Le système affiche tous les groupes déjà recensés.

Le superviseur choisit un groupe.

Le système affiche les informations suivantes :

- le nom du groupe,
- la liste des personnes appartenant au groupe,
- la liste des accès aux groupes de portes.

Le superviseur choisit un accès.

Le système affiche les informations suivantes :

un calendrier ouvert par défaut sur la semaine courante,

dans le calendrier, les plages horaires autorisées.

Le superviseur modifie les informations précédentes.

Le système enregistre les informations.

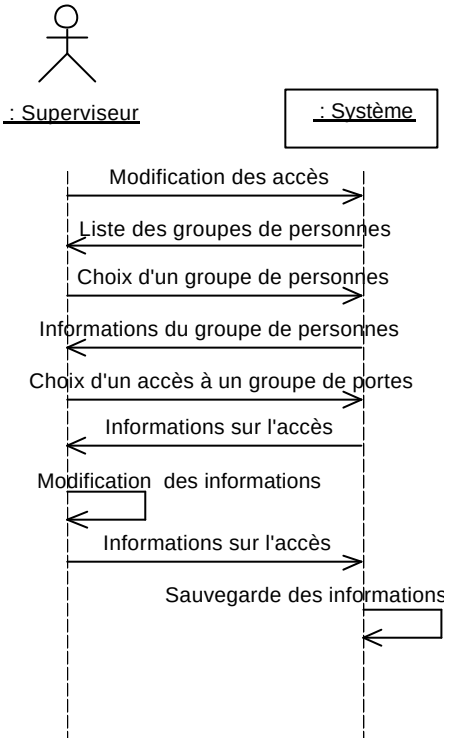


Figure 402 : Modification des accès d'un groupe de personnes à un groupe de portes.

Modification d'une semaine type

Le superviseur demande la liste des semaines types.

Le système affiche toutes les semaines types déjà recensées.

Le superviseur choisit une semaine type.

Le système affiche les informations suivantes :

le nom de la semaine,

une description textuelle de la semaine,

les jours de la semaine découpés en tranches horaires,

heure par heure la validité ou non de l'accès.

Le superviseur modifie les informations précédentes.

Le système enregistre les informations.

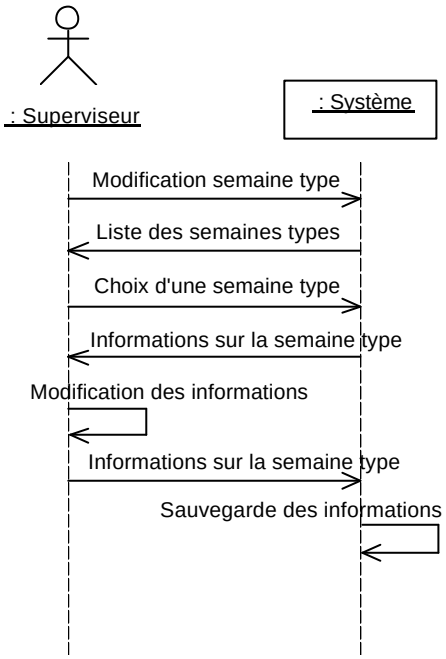


Figure 403 : Modification d'une semaine type.

Recherche des droits d'accès d'une personne pour une porte donnée

Le superviseur demande la liste des personnes et des portes.

Le système affiche toutes les personnes et portes déjà recensées.

Le superviseur choisit une personne et une porte

Le système affiche les informations suivantes :

un calendrier ouvert par défaut sur la semaine courante,

dans le calendrier, les plages horaires autorisées.

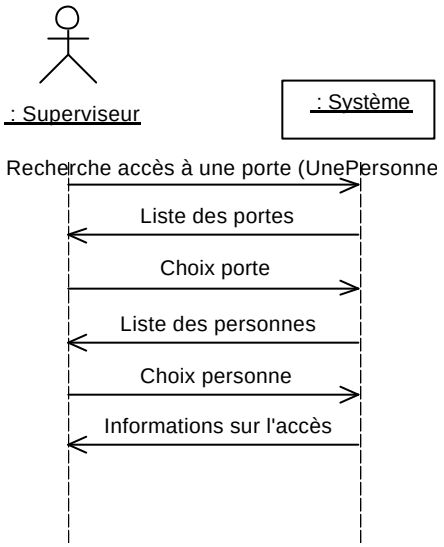


Figure 404 : Recherche des droits d'accès d'une personne pour une porte donnée.

Surveillance

La surveillance est déclenchée par le gardien.

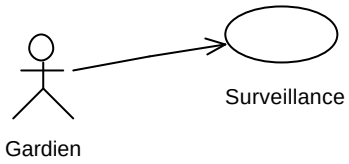


Figure 405 : Déclenchement de la surveillance par le gardien.

Identification

Le gardien se connecte au système et donne son mot de passe.

Le système vérifie l'identité du gardien et autorise la connexion.

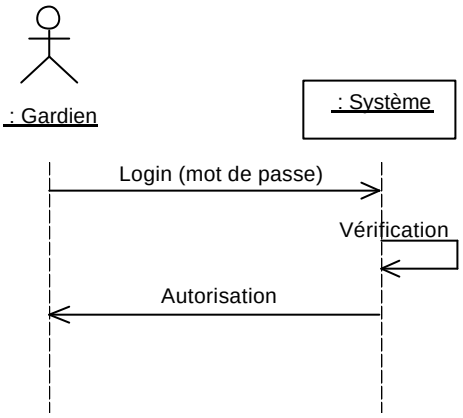


Figure 406 : Identification du gardien.

Rapport des événements

Le gardien spécifie les dates de début et de fin de la période désirée.

Tant que le gardien a envie

Le système affiche les événements dans l'ordre chronologique.

Le gardien sélectionne des filtres d'affichage.

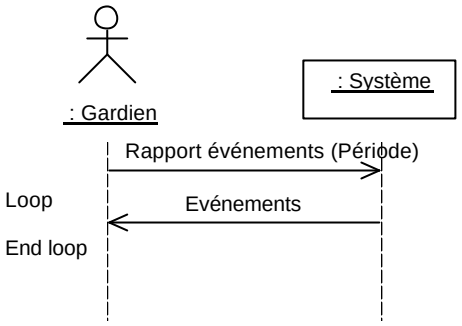


Figure 407 : Rapport des événements.

Purge des événements

Le gardien spécifie les dates de début et de fin de la période à purger.

Le système détruit les événements qui correspondent à la période.

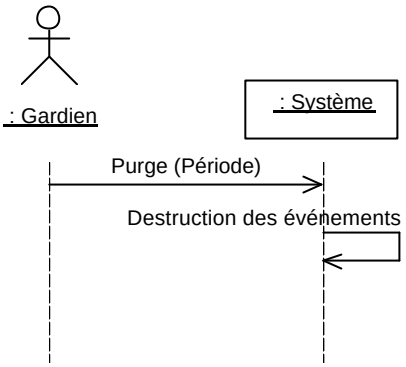


Figure 408 : Purge des événements.

Rapport des alarmes

Le gardien spécifie le délai de rafraîchissement (1..60 minutes).

Jusqu'à ce que le gardien interrompe la détection

Le système affiche régulièrement les nouvelles alarmes.

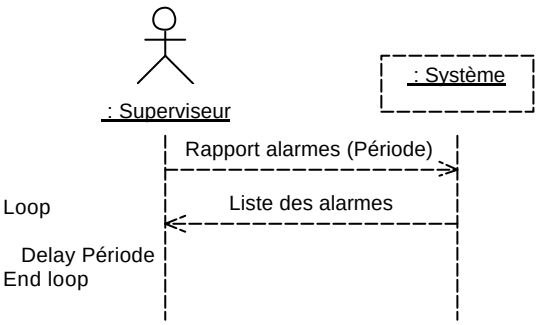


Figure 409 : Rapport des alarmes.

Ouverture manuelle des portes

Le gardien demande la liste des portes.

Le système affiche la liste des portes.

Le gardien choisit une porte.

Le système affiche les informations suivantes :

l'état (activé / désactivé),

la valeur de la confidentialité (une personne est-elle dans la salle ?),

les dix derniers événements.

Le gardien force l'ouverture de la porte.

Le système enregistre l'événement.

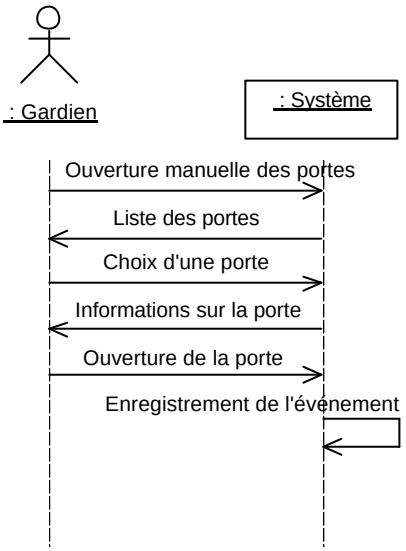


Figure 410 : Ouverture manuelle des portes.

Incendie

Le gardien déclenche l'alarme incendie.

Le système ouvre toutes les portes.

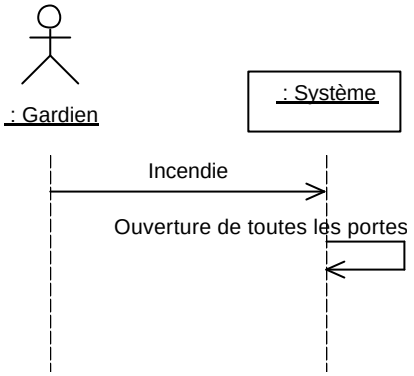


Figure 411 :Alarme incendie.

Contrôle d'accès

Le contrôle d'accès est déclenché par le porteur de badge.

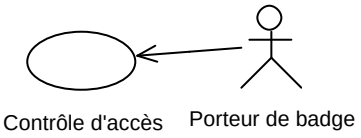


Figure 412 : Déclenchement du contrôle d'accès par le porteur de badge.

Autorisation de passage

La personne présente son badge.

Le système détermine si l'accès est autorisé.

Si l'accès est autorisé

Le système commande l'ouverture de la porte.

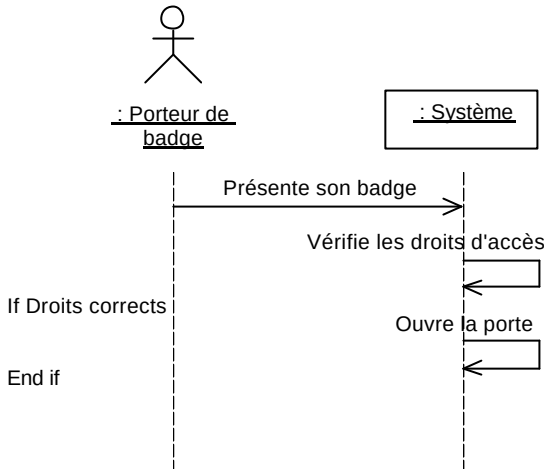


Figure 413 : Autorisation de passage.

Tableau récapitulatif des cas d'utilisation et des scénarios principaux

Cas d'utilisation	Scénarios principaux
Configuration	Identification
Configuration	Modification des portes
Configuration	Modification des personnes
Configuration	Modification des groupes de personnes
Configuration	Modification des groupes de portes
Configuration	Recherche d'une personne en fonction d'un badge

Cas d'utilisation	Scénarios principaux
Configuration	Recherche des portes franchissables par une personne donnée
Configuration	Recherche des groupes qui contiennent une personne donnée
Configuration	Recherche des personnes qui appartiennent à un groupe donné
Configuration	Modification des accès d'un groupe de personnes à un groupe de portes
Configuration	Modification d'une semaine type
Configuration	Affichage des droits d'accès d'une personne pour une porte donnée
Surveillance	Identification
Surveillance	Rapport des événements
Surveillance	Purge des événements
Surveillance	Détection des alarmes
Surveillance	Ouverture manuelle des portes
Surveillance	Incendie
Contrôle d'accès	Autorisation de passage

Contrôles de cohérence

Les contraintes suivantes doivent être prise en compte par le système :

- chaque lecteur de badges est identifié par une adresse unique,
- un lecteur de badges ne peut être associé qu'à une seule porte,
- une porte doit toujours être au moins dans son propre groupe,
- une personne doit toujours être au moins dans son propre groupe,
- un badge ne peut être alloué qu'à une seule personne,
- les plages horaires définies dans une journée ne doivent pas se chevaucher.

Description des collaborations

Les fonctionnalités décrites par les cas d'utilisation sont réalisées par des collaborations d'objets du domaine.

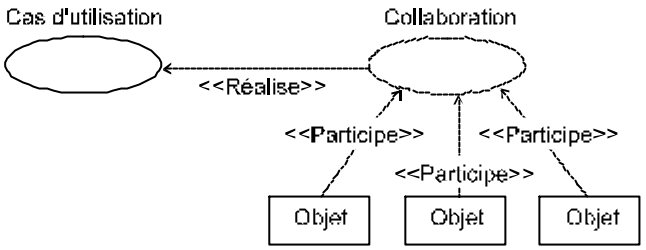


Figure 414 : Réalisation d'un cas d'utilisation par une collaboration entre objets du domaine.

La réalisation des collaborations fait intervenir des objets supplémentaires qui n'appartiennent pas au domaine de l'application mais qui sont nécessaires à son fonctionnement. Ces objets effectuent généralement l'interface entre le système et l'utilisateur, ou entre le système et un autre système. La représentation de ces objets permet d'évaluer le coût de l'application ; il est par exemple fréquent de compter une journée de travail pour la réalisation d'un écran et des mécanismes associés.

La nature particulière des classes de ces objets peut être signifiée au moyen des stéréotypes suivants :

- `<<contrôleur>>` pour le pilotage et l'ordonnancement,
- `<<dispositif>>` pour la manipulation du matériel,
- `<<interface>>` pour la traduction d'information à la frontière entre deux systèmes,
- `<<substitut>>` pour la manipulation d'objets externes au système,
- `<<vue>>` pour la représentation des objets du domaine.

Très souvent, il existe trois éléments du modèle qui correspondent au même élément du monde physique. Cette distinction est en particulier opérée dans le cas des personnes, par l'intermédiaire :

- des acteurs qui représentent les utilisateurs d'un système, vu de l'extérieur,
- des objets, instances de classes issues de l'analyse du domaine (également appelés objets métiers) qui encapsulent les informations décrivant chaque utilisateur,

- des objets d'interface qui permettent de manipuler, lire et modifier les informations contenues par les objets de la catégorie précédente.

Le diagramme suivant représente cette trilogie de classes d'objets. L'utilisateur est montré sous la forme d'un acteur afin de représenter une personne (un humain) qui interagit avec le système. Les informations sur cette personne sont stockées par un substitut et visualisées par un objet d'une classe d'interface (dit objet miroir) dont le nom est préfixé par la lettre I. Un même objet d'interface peut visualiser successivement les caractéristiques de différentes personnes.

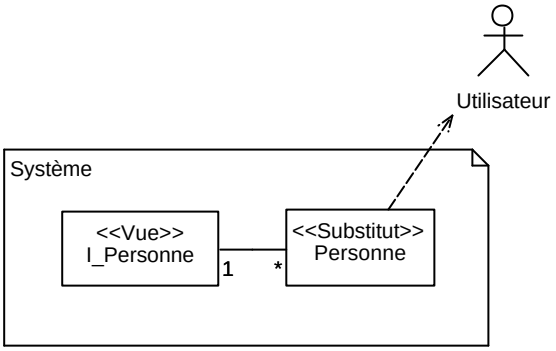


Figure 415 : Différentes catégories de classes d'objets.

Dans un système d'information, les informations qui font foi sont celles contenues dans le système. Dans un système automatisé, la vérité est toujours à l'extérieur du système. Dans les deux cas, il importe de bien synchroniser l'état de la trilogie d'objets.

Les acteurs n'interviennent plus directement dans les collaborations. L'interaction avec les utilisateurs est exprimée sous la forme de messages envoyés et reçus par les objets d'interface. Le niveau de granularité de ces objets est variable, mais en règle générale les mécanismes représentés en termes d'objets collaborants évitent d'aller trop loin dans les détails de la réalisation de l'interface utilisateur. Il y a deux raisons à cela : d'une part, faciliter la lecture des diagrammes, et d'autre part, favoriser la réutilisation des objets métiers. L'interface est toujours très dépendante de chaque application. En revanche, les objets métiers peuvent souvent être réutilisés par d'autres applications.

Configuration

Identification du superviseur

Dans un premier temps, l'acteur peut être conservé pour représenter de manière compacte l'interface utilisateur.

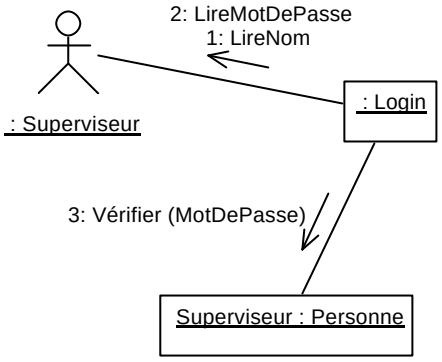


Figure 416 : Représentation d'une collaboration entre des objets du domaine et un acteur.

De manière alternative, il est également possible de créer une classe **`LimiteDuSystème`** et de l'employer en lieu et place de l'acteur.

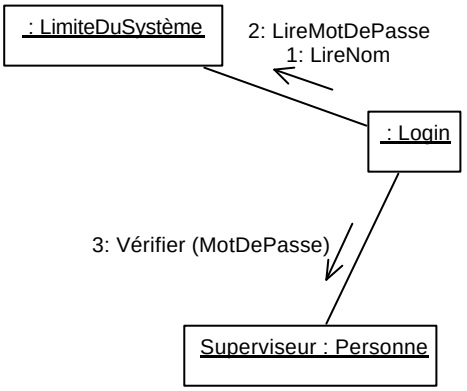


Figure 417 : Représentation globale de l'interface utilisateur au moyen d'un objet de classe ***`LimiteDuSystème`***.

Par la suite, les grandes lignes de l'interface utilisateur peuvent être décrites au moyen de classes qui représentent les différentes fenêtres. Dans ce cas, le but n'est pas de décrire de manière précise les classes graphiques de l'interface, mais plutôt de capturer la forme générale des interactions et ainsi de quantifier la charge de travail pour le développement de l'interface. La fabrication de l'interface proprement dite est du ressort d'outils spécialisés dans la construction et la génération automatique des interfaces graphiques.

Par convention, chaque objet du domaine est visualisé par un objet d'interface de même nom, préfixé par la lettre **`F`**. Dans le diagramme suivant, l'objet de classe **`Login`** est accessible par l'intermédiaire d'un objet graphique **`F_Login`**.

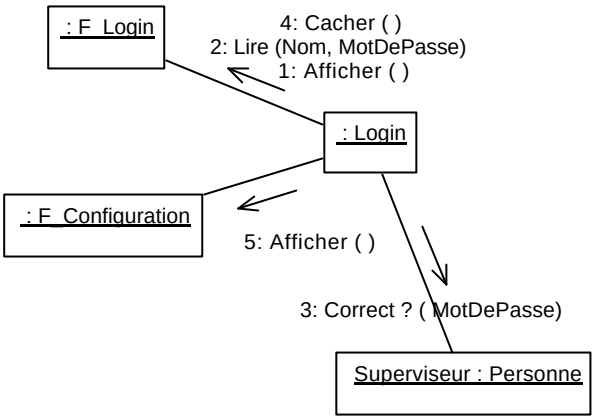


Figure 418 : Réalisation de la connexion au système par une collaboration entre objets.

Le diagramme de collaboration ci-dessus montre des objets instances de quatre classes différentes, divisées en deux classes d'objets du domaine et en deux classes d'interface utilisateur.

Un diagramme de classes préliminaire, compatible avec la collaboration précédente, est représenté ci-dessous. Etant donné l'état peu avancé de la modélisation, les informations de multiplicité ne sont pas toutes déterminées à ce stade.

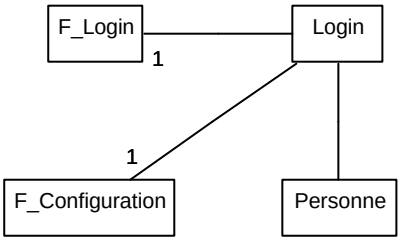


Figure 419 : Ebauche du diagramme de classes.

Le comportement des objets de la classe **`Login`** peut être décrit de manière générale par l'automate suivant. L'automate montre, entre autres, que le système ne précise pas l'origine précise du rejet d'une connexion.

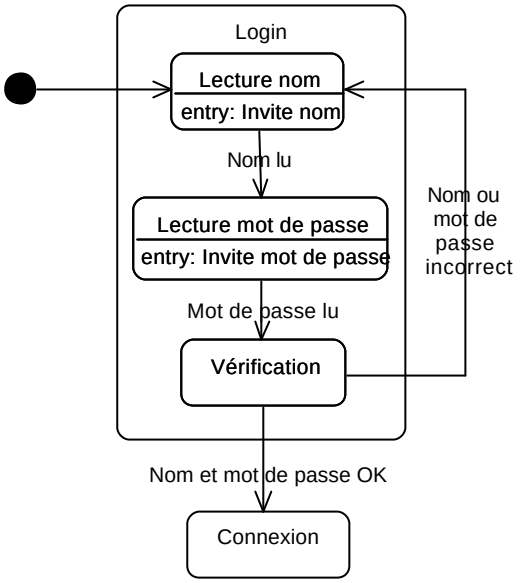


Figure 420 : Description générale du comportement des objets de la classe ***`Login`***.

Les classes d'interface utilisateur dérivent toutes les deux d'une classe **`Fenêtre`** qui définit les opérations **`Afficher()`** et **`Cacher()`**.

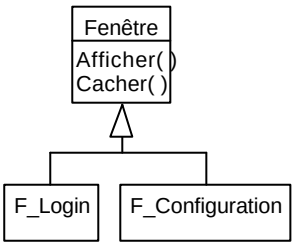


Figure 421 : Hiérarchie des classes d'interface utilisateur.

Modification des portes

Le superviseur peut modifier les portes de manière individuelle ou par groupe de portes. Les deux scénarios suivants correspondent respectivement à ces deux options.

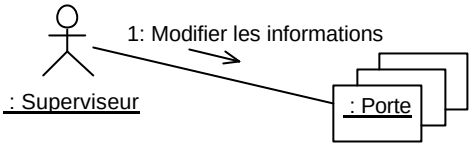


Figure 422 : Modification des informations des portes par le superviseur.

Scénario pour une seule porte

Le diagramme de collaboration suivant représente la modification des informations d'une porte.

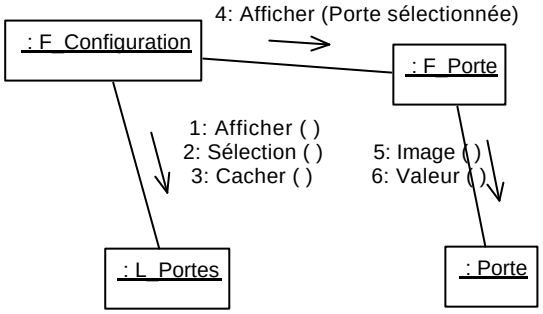


Figure 423 : Modification des informations d'une porte.

Les opérations **`Image()`** et **`Valeur()`** symbolisent la lecture et l'écriture de l'état de l'objet **`Porte`** à partir de l'interface utilisateur. L'opération **`Image()`** permet à un objet d'interface d'extraire les informations contenues par un objet du domaine afin de les montrer aux utilisateurs. Inversement, l'opération **`Valeur()`** permet à un objet d'interface de transférer des informations en provenance d'un utilisateur vers un objet du domaine.

Ces opérations sont héritées de la classe abstraite **`ObjetDuDomaine`**.

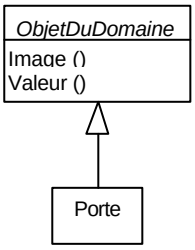


Figure 424 : La classe abstraite ***`ObjetDuDomaine`*** *spécifie les opérations* ***`Image()`*** *et* ***`Valeur()`*** *qui permettent la communication avec l'interface.*

La figure suivante représente le diagramme de classes préliminaire qui correspond au diagramme de collaboration précédent.

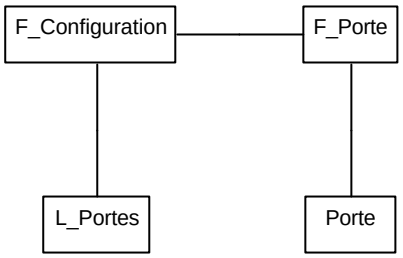


Figure 425 : Ebauche du diagramme de classes.

Les classes dont le nom est préfixé par une lettre **`L`** représentent des classes d'interface utilisateur spécialisées dans la représentation de listes d'éléments. Ces classes possèdent une opération **`Sélection()`** qui permet de récupérer l'élément sélectionné par l'utilisateur dans la liste. Les classes de visualisation de listes sont également des fenêtres, comme le montre le diagramme de classe suivant :

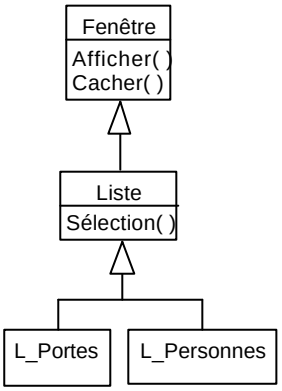


Figure 426 : Hiérarchie des classes de visualisation de listes.

Scénario pour un groupe de portes

L'opération de modification des portes peut également être réalisée de manière globale pour un ensemble de portes regroupées dans un groupe de portes.

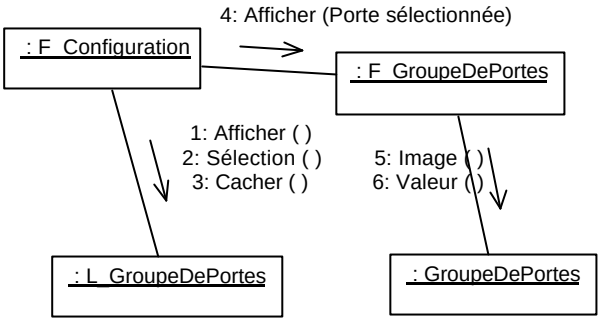


Figure 427 : Modification d'un groupe de portes.

Le diagramme de classes précédent est enrichi pour prendre en compte la manipulation des groupes de portes.

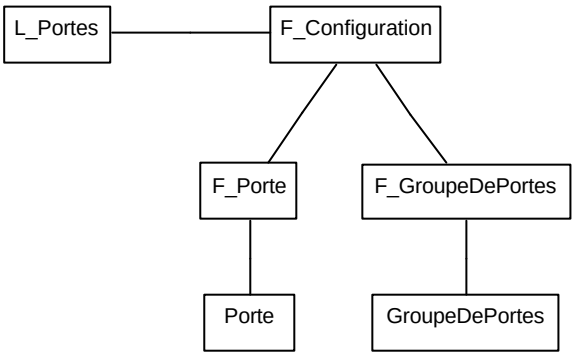


Figure 428 : Ebauche du diagramme de classes.

Modification des personnes

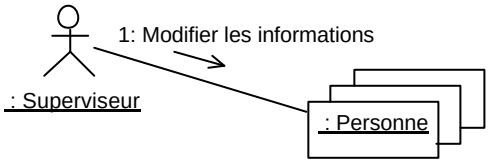


Figure 429 : Modification des informations d'une personne.

Diagramme de collaboration

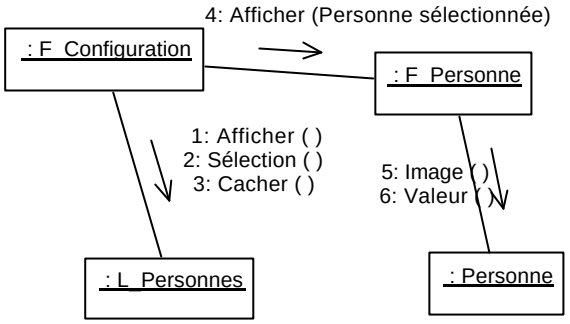


Figure 430 : Diagramme de collaboration. Modification des informations d'une personne.

Ebauche du diagramme de classes

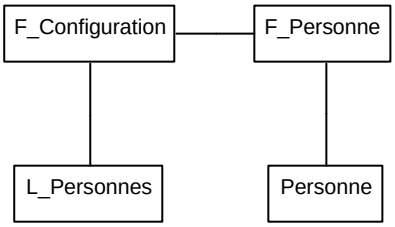


Figure 431 : Ebauche du diagramme de classes.

Modification des groupes de personnes

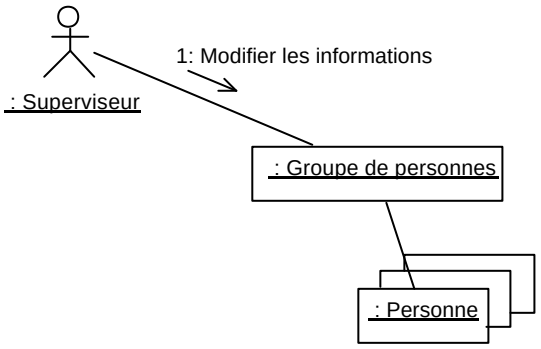


Figure 432 : Modification des groupes de personnes.

Diagramme de collaboration

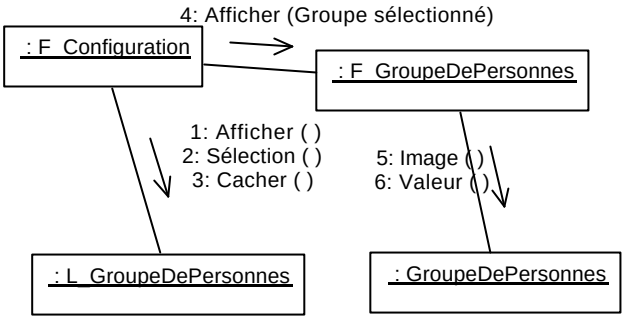


Figure 433 : Diagramme de collaboration.

Ebauche du diagramme de classes

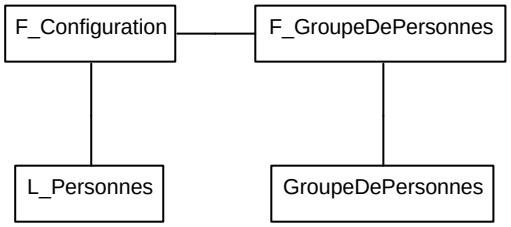


Figure 434 : Ebauche du diagramme de classes.

Modification des groupes de portes

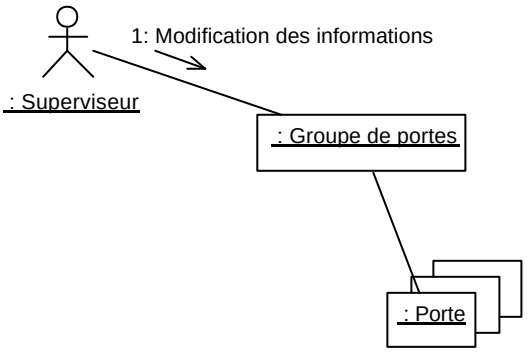


Figure 435 : Modification des groupes de portes.

Diagramme de collaboration

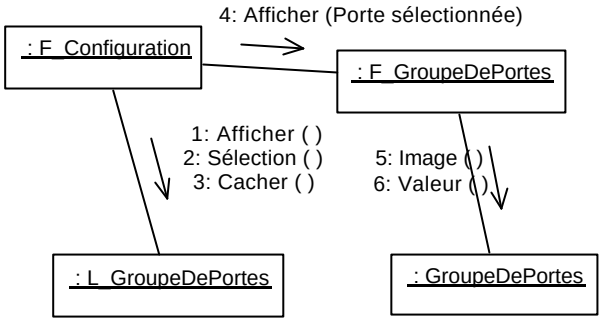


Figure 436 : Diagramme de collaboration.

Ebauche du diagramme de classes

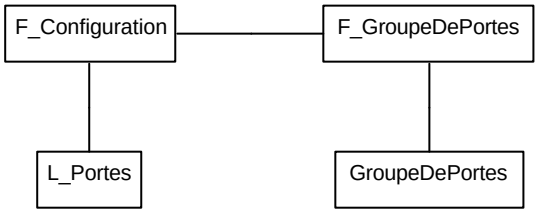


Figure 437 : Ebauche du diagramme de classes.

Recherche d’une personne en fonction d’un badge

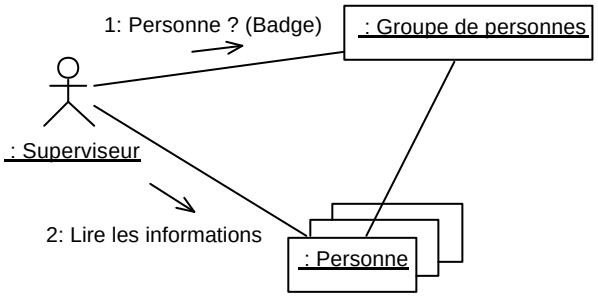


Figure 438 : Recherche d’une personne en fonction d’un badge.

Diagramme de collaboration

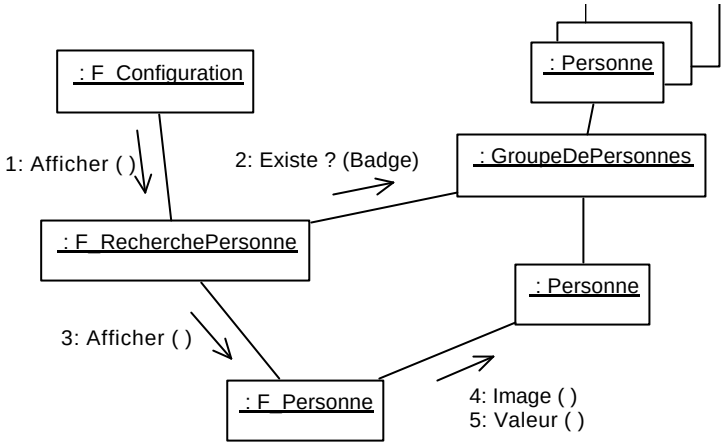


Figure 439 : Diagramme de collaboration.

Ebauche du diagramme de classes

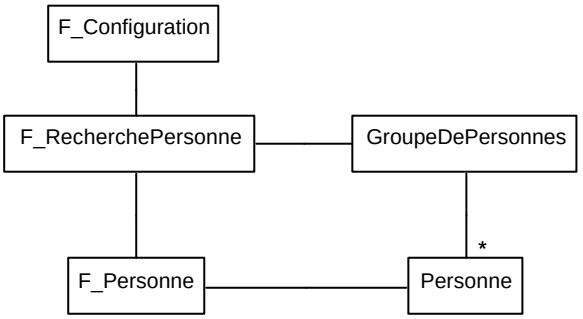


Figure 440 : Ebauche du diagramme de classes.

Recherche des portes franchissables par une personne donnée

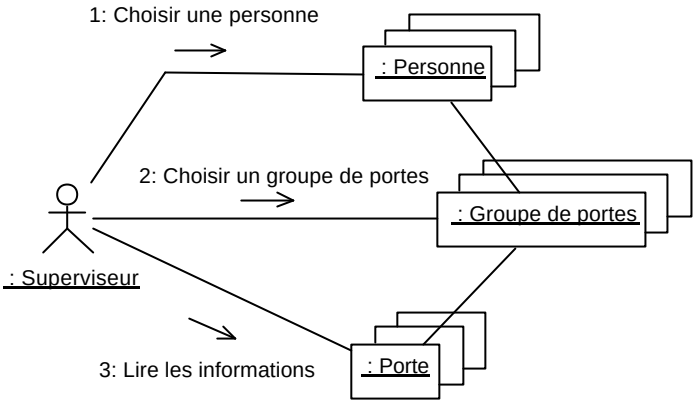


Figure 441 : Recherche des portes franchissables par une personne donnée.

Diagramme de collaboration

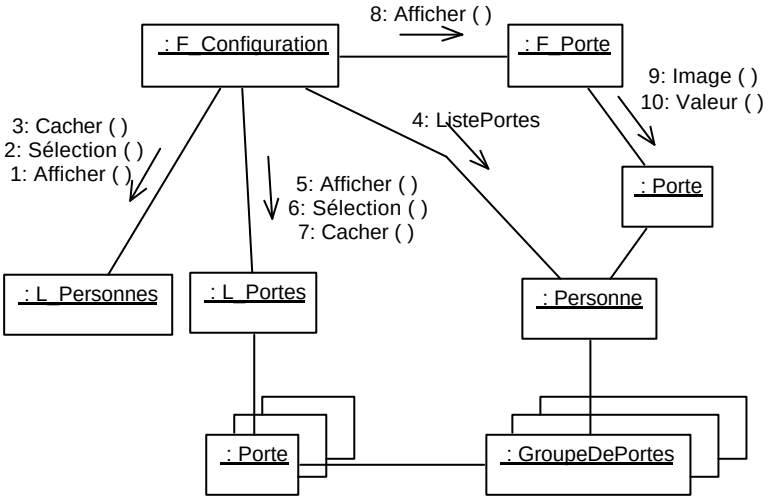


Figure 442 : Diagramme de collaboration.

Ebauche du diagramme de classes

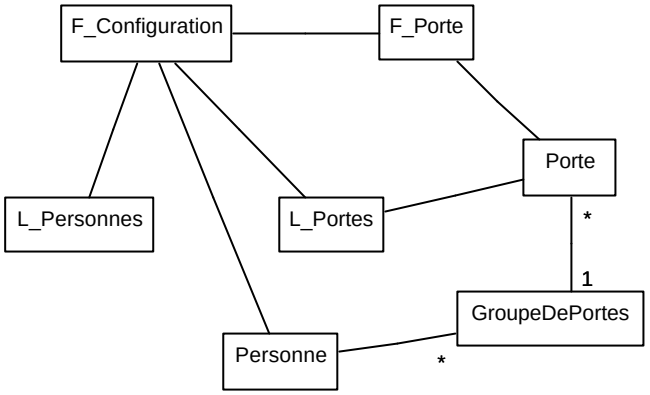


Figure 443 : Ebauche du diagramme de classes.

Recherche des groupes qui contiennent une personne donnée

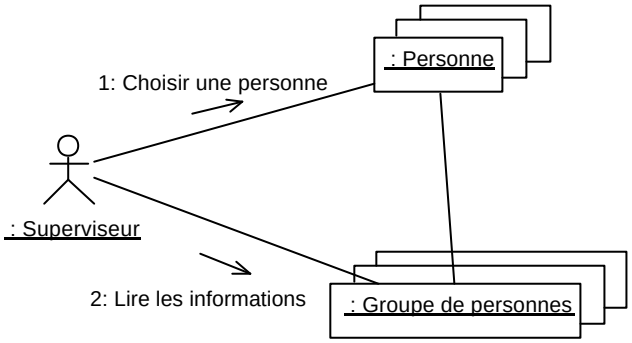


Figure 444 : Recherche des groupes qui contiennent une personne donnée.

Diagramme de collaboration

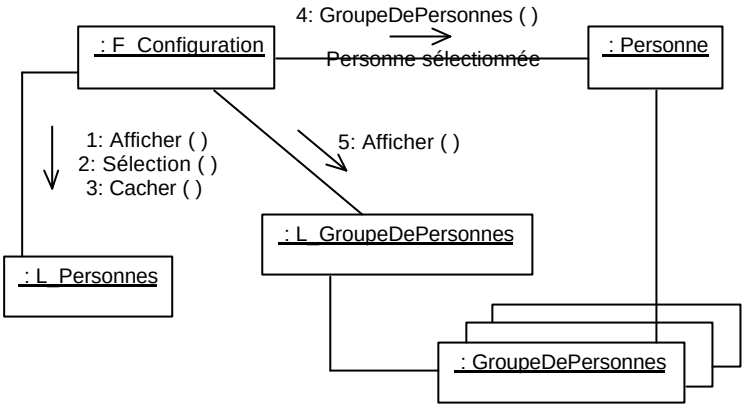


Figure 445 : Diagramme de collaboration.

Ebauche du diagramme de classes

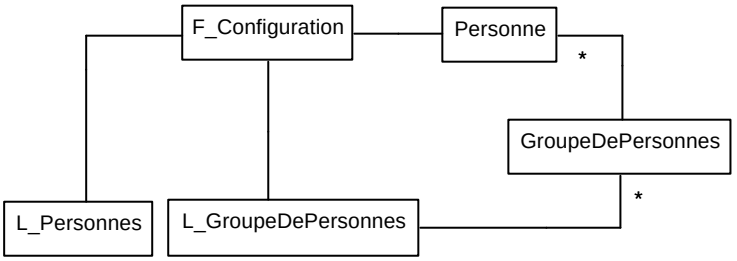


Figure 446 : Ebauche du diagramme de classes.

Recherche des personnes qui appartiennent à un groupe donné

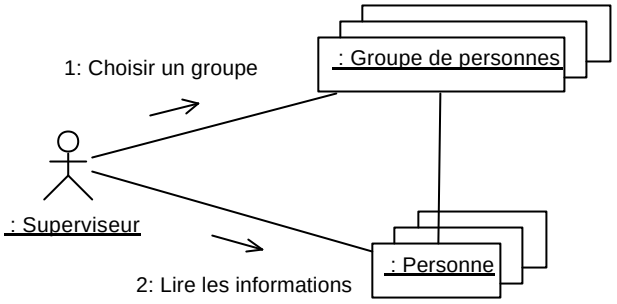


Figure 447 : Recherche des personnes qui appartiennent à un groupe donné.

Diagramme de collaboration

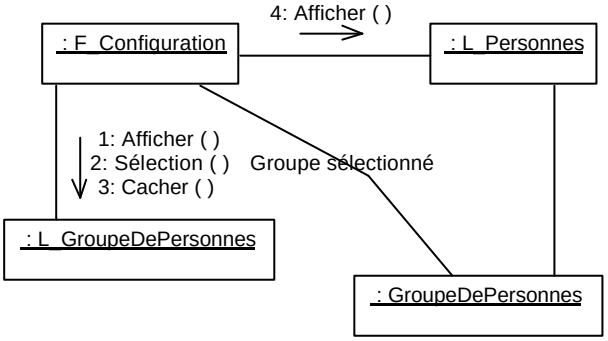


Figure 448 : Diagramme de collaboration.

Ebauche du diagramme de classes

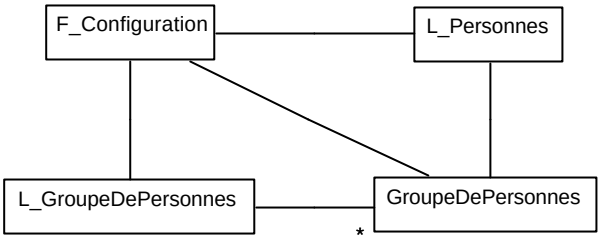


Figure 449 : Ebauche du diagramme de classes.

Modification des accès d'un groupe de personnes à un groupe de portes

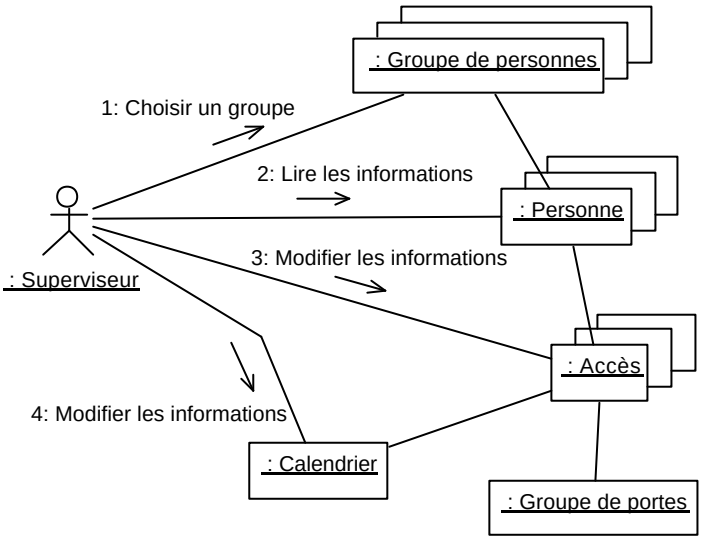


Figure 450 : Modification des accès d'un groupe de personnes à un groupe de portes.

Diagramme de collaboration

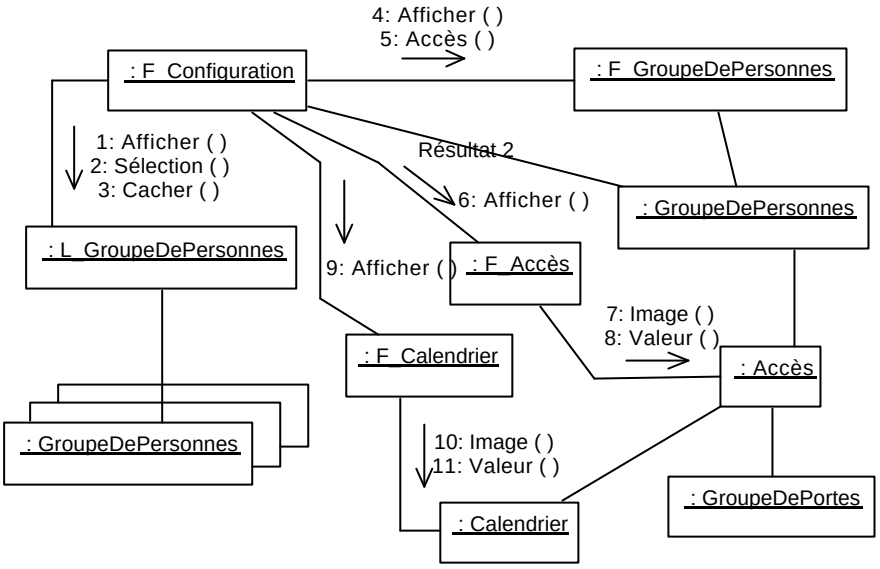


Figure 451 : Diagramme de collaboration.

Ebauche du diagramme de classes

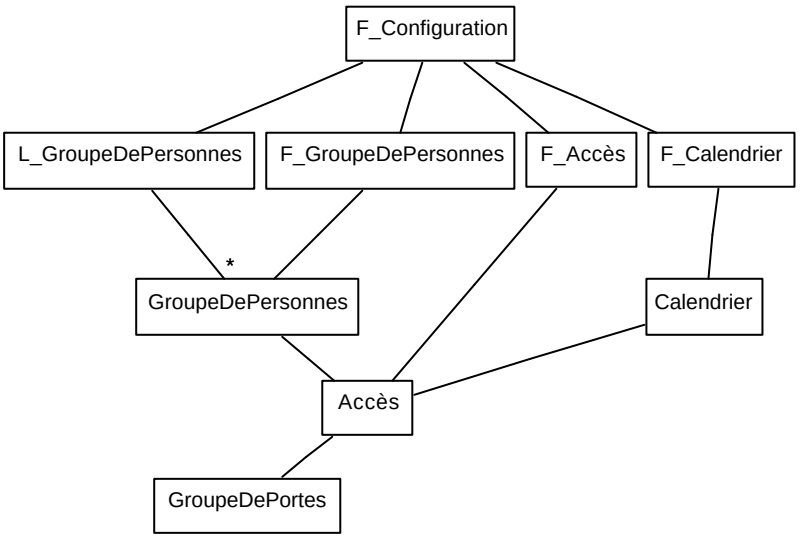


Figure 452 : Ebauche du diagramme de classes.

Modification d'une semaine type

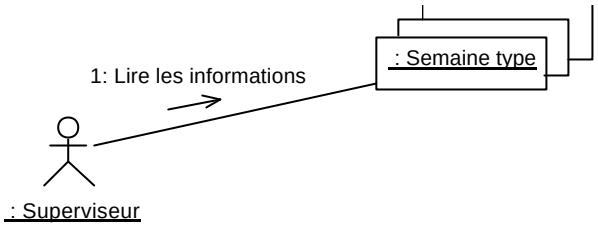


Figure 453 : Modification d'une semaine type.

Diagramme de collaboration

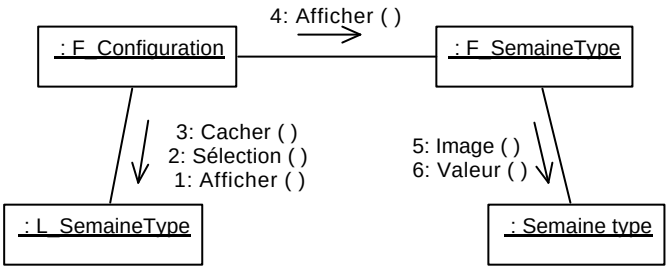


Figure 454 : Diagramme de collaboration.

Ebauche du diagramme de classes

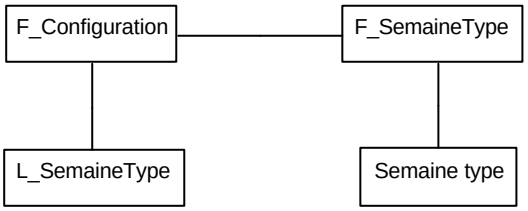


Figure 455 : Ebauche du diagramme de classes.

Recherche des droits d'accès d'une personne pour une porte donnée

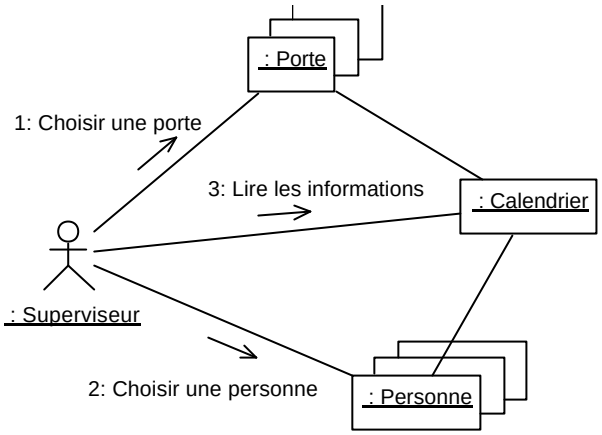


Figure 456 : Recherche des droits d'accès d'une personne pour une porte donnée.

Diagramme de collaboration

Les liens annotés par un numéro entre parenthèses ont été créés par l'opération déclenchée par le message de rang correspondant au numéro.

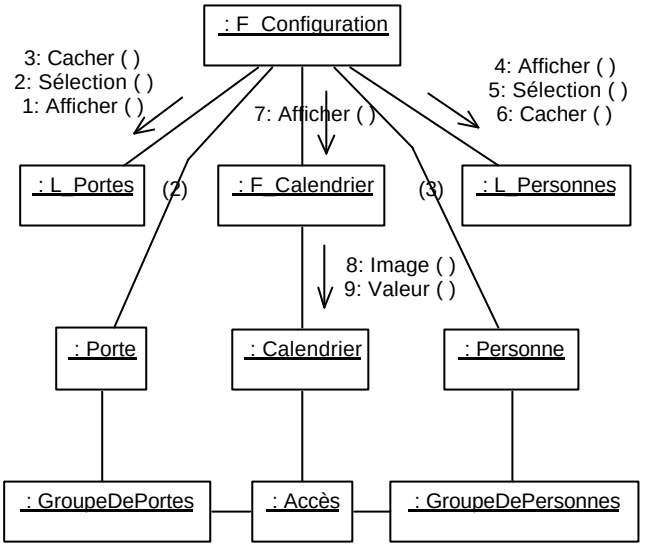


Figure 457 : Diagramme de collaboration.

Ebauche du diagramme de classes

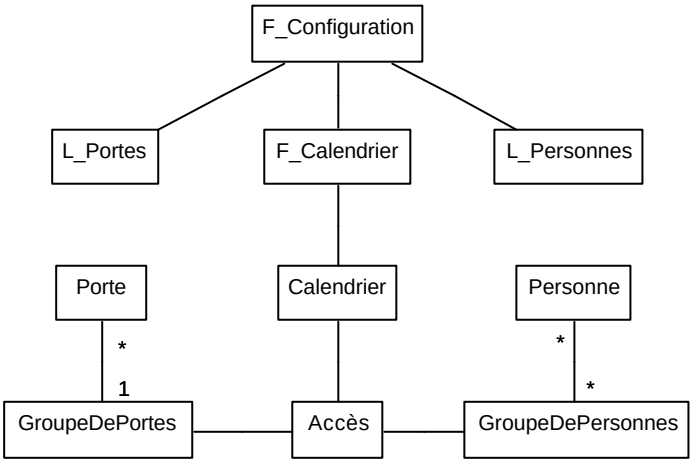


Figure 458 : Ebauche du diagramme de classes.

Surveillance

Identification du gardien

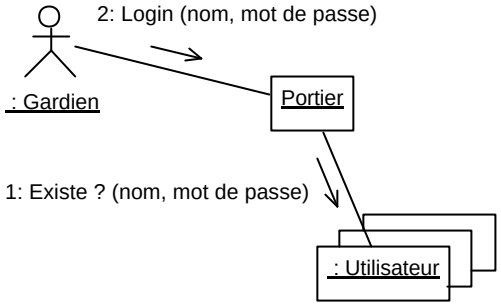


Figure 459 : Identification du gardien.

Diagramme de collaboration

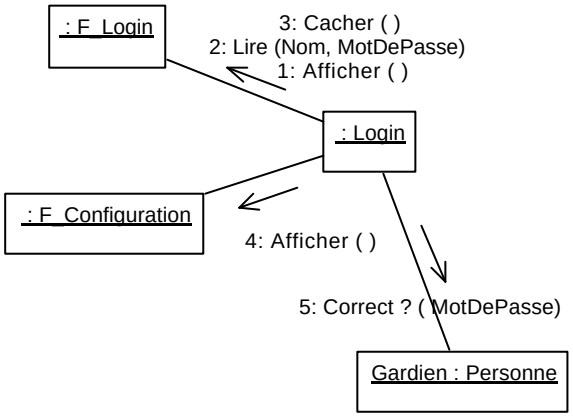


Figure 460 : Diagramme de collaboration.

Ebauche du diagramme de classes

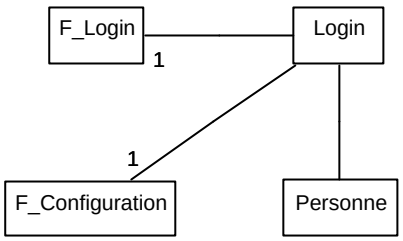


Figure 461 : Ebauche du diagramme de classes.

Rapport des événements

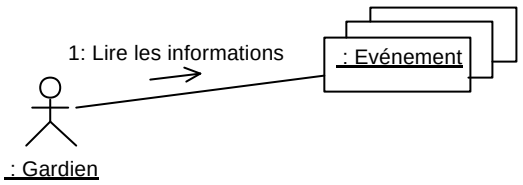


Figure 462 : Rapport des événements.

Diagramme de collaboration

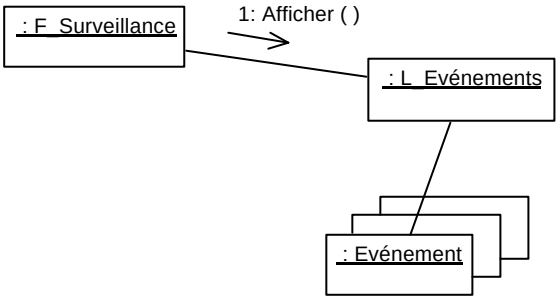


Figure 463 : Diagramme de collaboration.

Ebauche du diagramme de classes

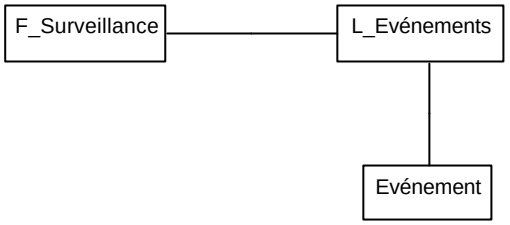


Figure 464 : Ebauche du diagramme de classes.

Purge des événements

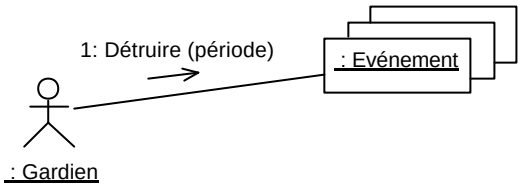


Figure 465 : Purge des événements.

Diagramme de séquence

L'interaction est représentée ici au moyen d'un diagramme de séquence qui est plus expressif qu'un diagramme de collaboration pour la représentation des structures de contrôle.

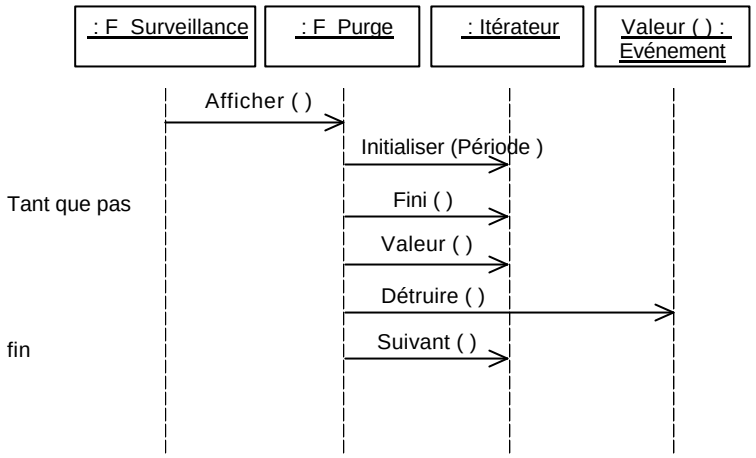


Figure 466 : Diagramme de séquence.

Ebauche du diagramme de classes

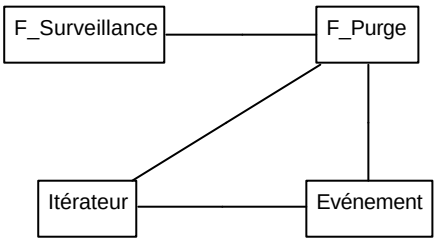


Figure 467 : Ebauche du diagramme de classes.

Rapport des alarmes

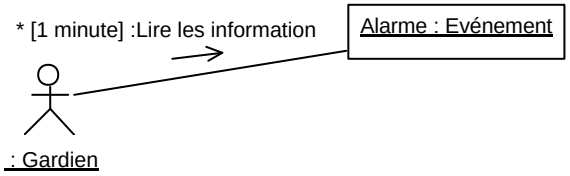


Figure 468 : Rapport des alarmes.

Diagramme de collaboration

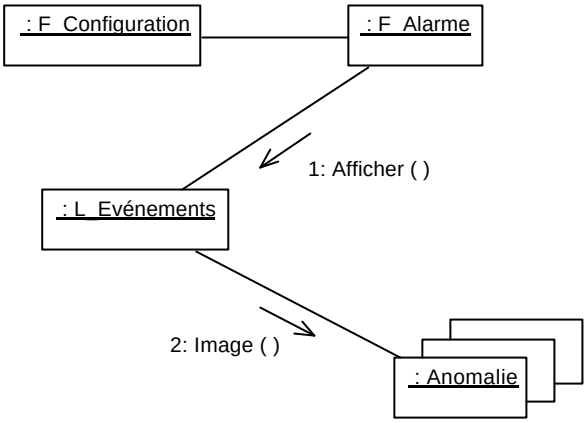


Figure 469 : Diagramme de collaboration.

Ebauche du diagramme de classes

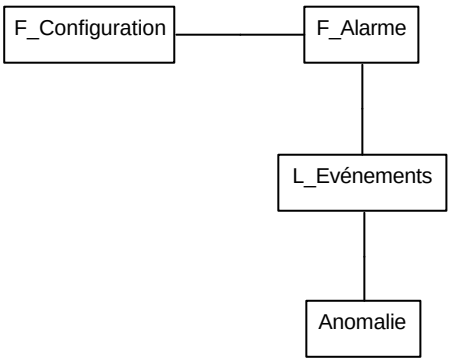


Figure 470 : Ebauche du diagramme de classes.

Ouverture manuelle des portes

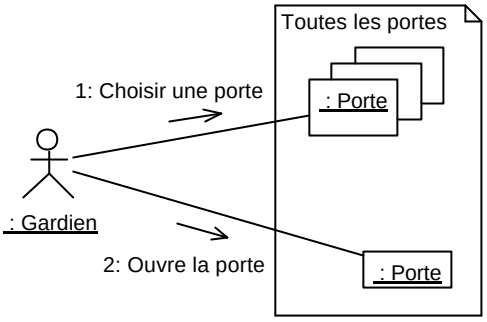


Figure 471 : Ouverture manuelle des portes.

Diagramme de collaboration

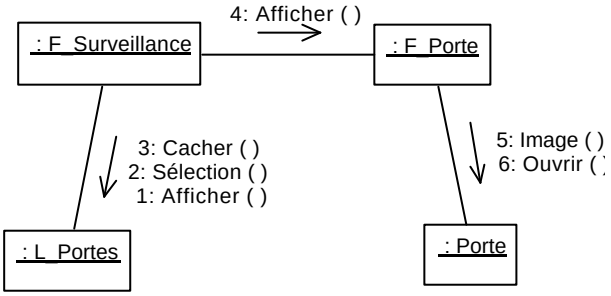


Figure 472 : Diagramme de collaboration.

Ebauche du diagramme de classes

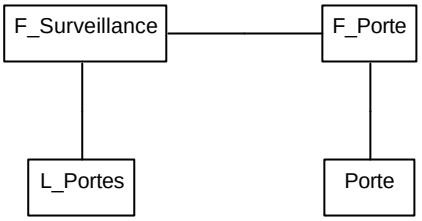


Figure 473 : Ebauche du diagramme de classes.

Incendie

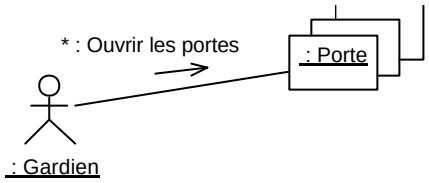


Figure 474 : Incendie.

Diagramme de séquence

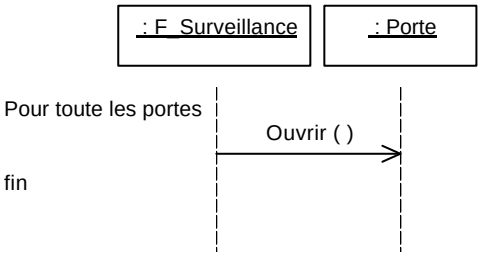


Figure 475 : Diagramme de séquence.

Ebauche du diagramme de classes

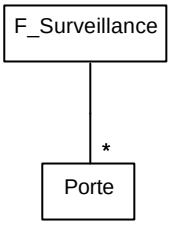


Figure 476 : Ebauche du diagramme de classes.

Contrôle d'accès

Autorisation de passage

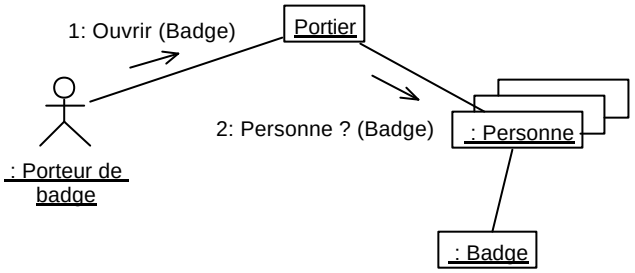


Figure 477 : Autorisation de passage.

Diagramme de collaboration

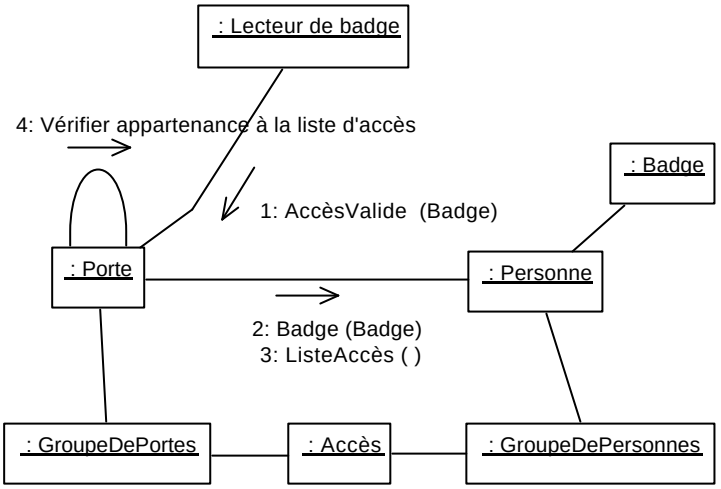


Figure 478 : Diagramme de collaboration.

Ebauche du diagramme de classes

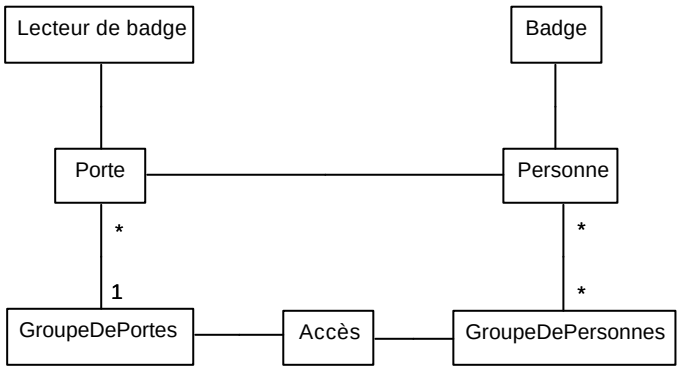


Figure 479 : Ebauche du diagramme de classes.

Analyse

Analyse du domaine

Les cas d'utilisation segmentent l'espace des besoins selon le point de vue d'un acteur à la fois. La description donnée par les cas d'utilisation est purement fonctionnelle et il convient de prendre garde à ne pas embrayer vers une décomposition fonctionnelle plutôt que vers une décomposition objet. Les cas d'utilisation doivent être vus comme des classes de comportement.

UML réalise les cas d'utilisation au moyen de collaborations entre objets issus du domaine de l'application. Chaque collaboration regroupe un contexte d'objet et une interaction entre ces objets. Le contexte des objets est exprimé de manière particulière dans les diagrammes de collaboration et de manière générale dans les diagrammes de classes. Ces diagrammes de classes ont été ébauchés dans le paragraphe précédent.

Le diagramme de classes suivant est obtenu automatiquement (grâce à l'outil Rose) à partir de ces différentes ébauches.

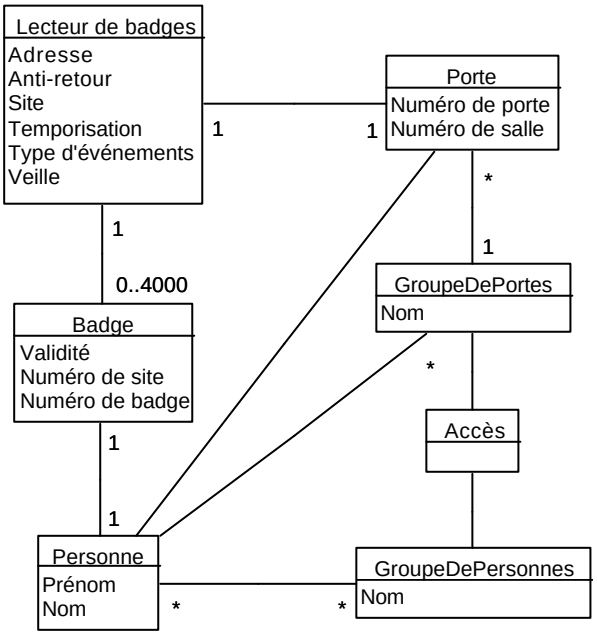


Figure 480 : Diagramme de classes obtenu par synthèse automatique des différentes ébauches.

L'examen du diagramme précédent fait apparaître deux associations redondantes (entre les classes **`Personne`**, **`Porte`** et **`GroupeDePortes`**) qui peuvent être dérivées d'autres associations. Le diagramme suivant fait état des ajustements qui ont été apportés au modèle du domaine.

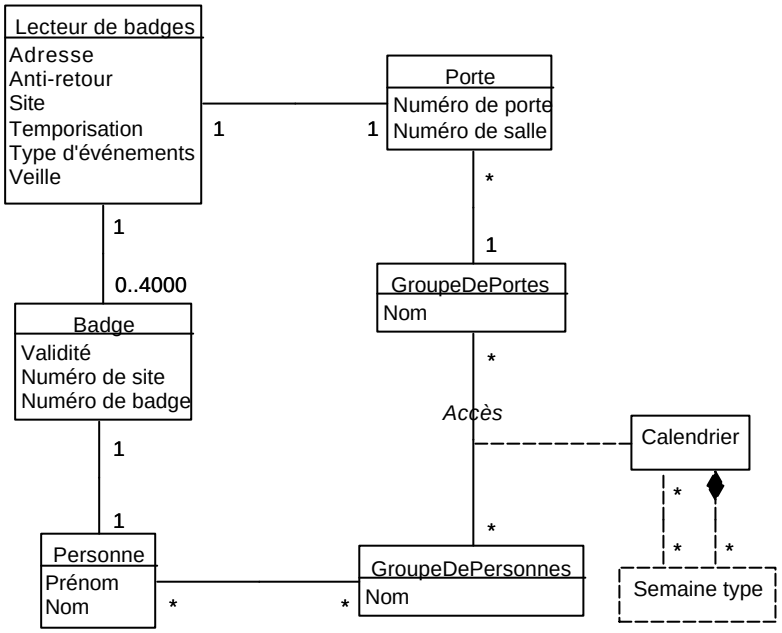


Figure 481 : Diagramme des classes du domaine.

La classe **`Accès`** a été transformée en association car elle ne contenait pas d'informations particulières. Le détail des droits est exprimé dans la classe-association **`Calendrier`**. La relation de composition représente les semaines type qui ont été modifiées localement dans un calendrier donné.

Analyse de l'existant

L'Essaim dispose déjà d'un certain nombre de lecteurs de badges et désire les réutiliser dans le nouveau système de contrôle d'accès. Ces lecteurs de badges peuvent fonctionner de manière totalement autonome ; ils sont programmables sur place au moyen de badges particuliers ou à distance via une liaison série.

Tous les lecteurs sont esclaves du système de contrôle : un lecteur n'est jamais à l'origine d'une interaction.

Caractéristiques des lecteurs de badges

Chaque lecteur de badges possède les caractéristiques suivantes :

- la mémorisation de 4 000 cartes, avec la possibilité d'invalider certaines cartes,
- la mémorisation des 100 derniers événements (un filtre permet d'enregistrer uniquement les anomalies),
- une adresse sur le réseau (il peut y avoir jusqu'à 64 lecteurs connectés au réseau),
- une horloge logicielle,
- 8 plages horaires,
- une temporisation de gâche,
- une fonction anti-retour optionnelle (entre tête principale et tête auxiliaire),
- une entrée tout-ou-rien connectée au choix à un bouton-poussoir, un contact de porte ou une boucle de détection,
- un état qui précise si le lecteur est actif ou en veille.

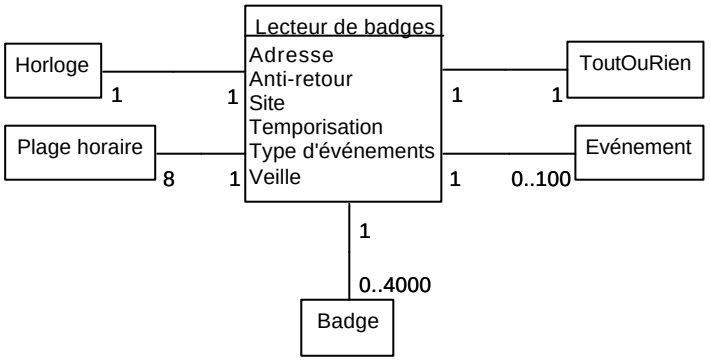


Figure 482 : Représentation des caractéristiques des lecteurs de badges.

Mise à l'heure

Chaque lecteur contient une horloge logicielle. En cas de coupure de courant, le lecteur enregistre un événement spécial lors de sa remise sous tension et ne gère plus les plages horaires. L'horloge doit alors être remise à l'heure par le système de contrôle pour que le lecteur fonctionne à nouveau avec les plages horaires. La différence entre l'heure de l'horloge et l'heure réelle donne la durée de la coupure.

Figure 483 : Horloge logicielle contenue par les lecteurs de badges.

Gestion des badges

Le lecteur peut mémoriser jusqu'à 4000 numéros de badge. Ces numéros de badge peuvent être manipulés par groupes de numéros croissants. Le lecteur propose les opérations suivantes :

- validation d'un badge,
- validation d'un groupe de badges,
- invalidation d'un badge,
- invalidation d'un groupe de badges.

Plages horaires

Une plage horaire est associée à chaque badge ou groupe de badges. Un lecteur contient la description de 8 plages horaires. Une plage horaire permet de restreindre les accès en fonction du jour et de l'heure de présentation du badge. Les 7 premières plages sont configurables par le superviseur, la dernière plage est le passe tout temps. Chaque plage horaire comporte 3 sous-plages qui ne doivent pas se recouvrir.

Les plages se présentent de la manière suivante :

Sous-plage	LU	MA	ME	JE	VE	SA	DI
00:00-00:00							
00:00-00:00							
00:00-00:00							

Figure 484 : Plage horaire.

Le diagramme de classes correspondant prend la forme suivante :

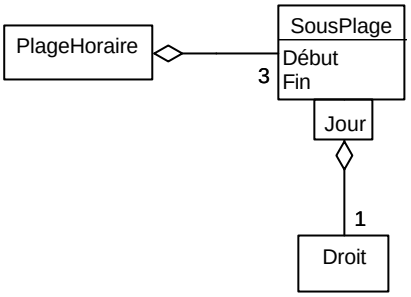


Figure 485 : Représentation des plages horaires.

Evénements

Les événements sont mémorisés suivant les critères de paramétrage (tous les événements ou seulement les anomalies). Chaque lecteur peut enregistrer au maximum 100 événements entre 2 interrogations provenant du système maître. Après interrogation les événements sont effacés. La mémoire du lecteur est organisée sous la forme d'un tampon circulaire. Les plus vieux événements sont effacés en cas de débordement de la mémoire.

Chaque événement contient les informations suivantes :

- la date,
- le numéro de badge,
- la tête concernée (principale ou auxiliaire).

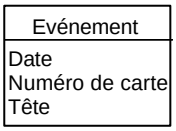


Figure 486 : Description des événements.

Le lecteur enregistre les événements suivants :

- carte acceptée,
- coupure secteur,
- carte refusée lecteur en veille,
- carte refusée hors plage,

- carte refusée mauvais code site,
- carte refusée carte non programmée,
- carte refusée défaut anti-retour,
- carte refusée mauvaise plage horaire.

Ces types d'événements sont représentés dans le diagramme suivant :

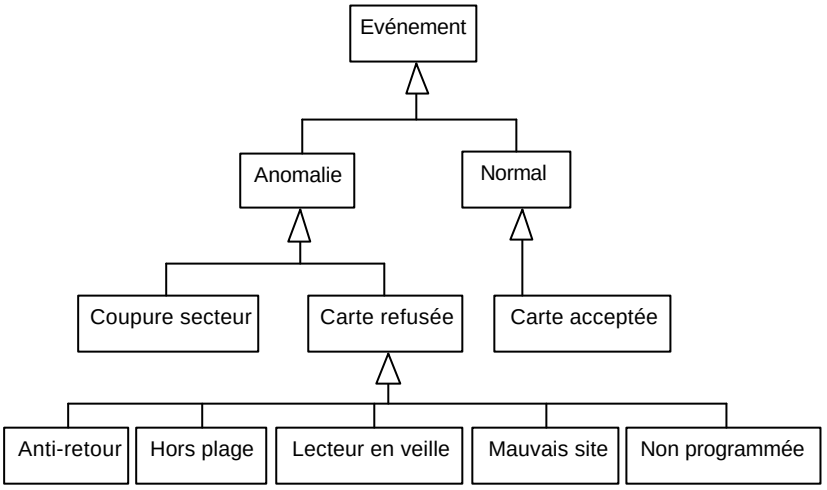


Figure 487 : Hiérarchie des types d'événements.

Types de messages

La communication entre les lecteurs de badges et le système maître est effectuée par des messages issus des trois grandes catégories suivantes :

- les messages simples qui contiennent un entête mais pas de données,
- les messages qui contiennent des données de longueur fixe,
- les messages qui contiennent des données de longueur variable.

Ces types de messages sont représentés dans le diagramme suivant :

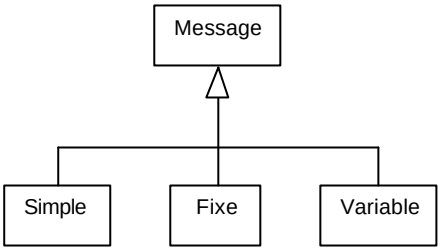


Figure 488 : Différentes catégories de messages.

Messages simples

Les messages simples regroupent des messages de synchronisation, des requêtes de rapport et des commandes.

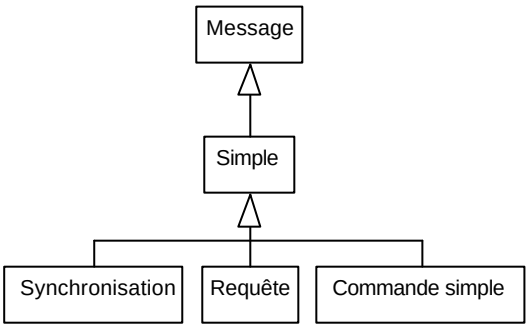


Figure 489 : Représentation des types de messages simples.

Messages reçus par le lecteur :

- acquittement,
- non acquittement,
- requête événements,
- requête réglage paramètres,
- requête cartes valides,
- requête cartes invalides,
- requête horloge,
- requête code site,
- commande mise en veille,

- commande sortie du mode veille,
- commande ouverture porte.

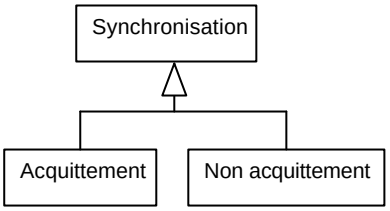


Figure 490 : Représentation des types de messages de synchronisation.

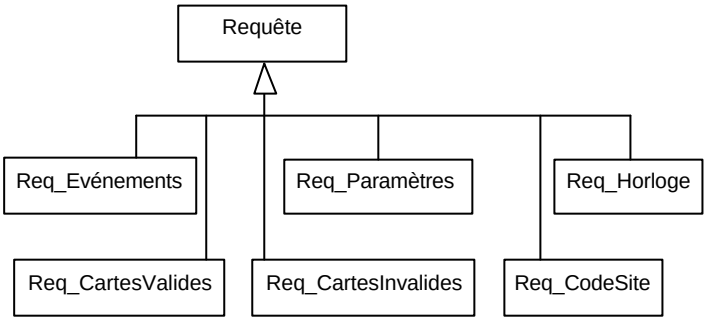


Figure 491 : Représentation des types de requêtes simples.

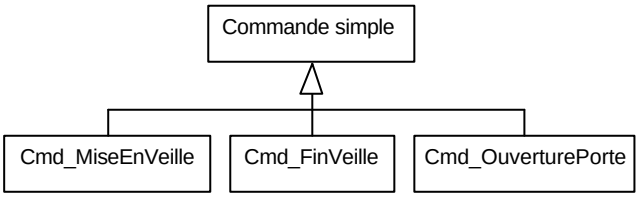


Figure 492 : Représentation des types de commandes simples.

Messages de longueur fixe

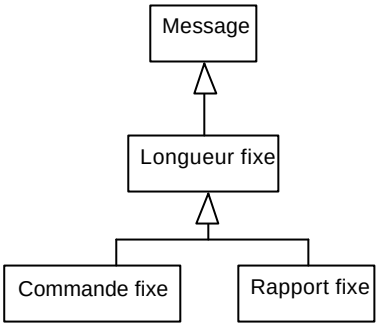


Figure 493 : Représentation des types de messages de longueur fixe.

Messages reçus par le lecteur :

- commande réglage paramètres,
- commande validation d'une carte,
- commande validation d'un groupe de cartes,
- commande invalidation d'une carte,
- commande invalidation d'un groupe de cartes,
- commande mise à l'heure,
- commande transmission d'une plage horaire,
- commande envoi du code site.

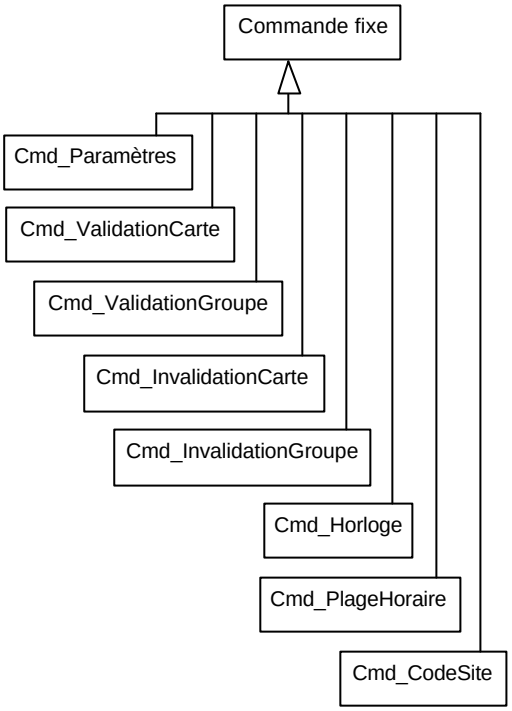


Figure 494 : Représentation des types de commandes fixes.

Messages envoyés par le lecteur :

- rapport réglage paramètres,
- rapport horloge,
- rapport d'une plage horaire du lecteur spécifié,
- rapport code site.

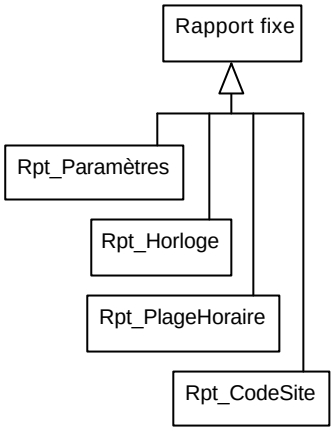


Figure 495 : Représentation des types de rapports fixes.

Messages de longueur variable

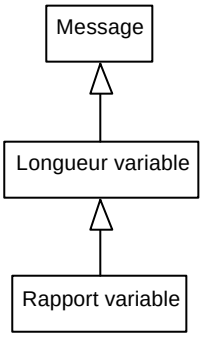


Figure 496 : Représentation du type messages de longueur variable.

Messages envoyés par le lecteur :

- rapport événement,
- rapport cartes valides,
- rapport cartes invalides.

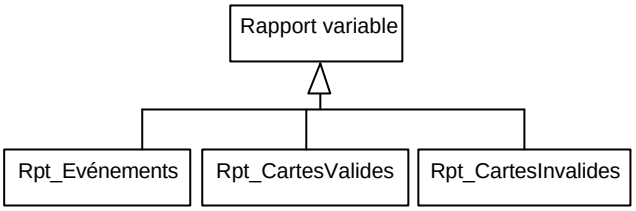


Figure 497 : Représentation des types de rapports de longueur variable.

Structure des trames

Les messages sont encapsulés par des trames qui contiennent des informations d'adressage et de vérification de la transmission.

Chaque trame prend la forme suivante :

Début	Adresse	Check-sum	Données	Fin
1 octet	1 octet	2 octets	0..n octets	2 octets
01H	00H .. 3FH			F1H – F2H

Figure 498 : Forme générale des trames de transmission.

La classe des trames offre les opérations **`Image()`** et **`Valeur()`** pour le transfert des trames. L'opération **`Image()`** transforme un objet trame en une suite d'octets susceptible d'être transmise via la liaison série qui relie les lecteurs de badges et les PC. Inversement, l'opération **`Valeur()`** reconstruit un objet trame à partir d'une suite d'octets en provenance d'un des lecteurs de badges.

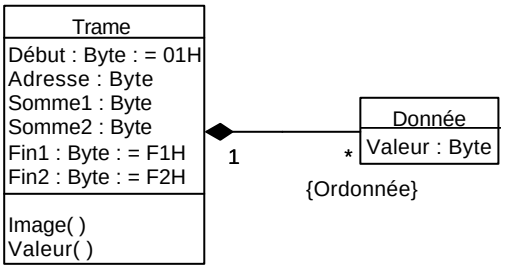


Figure 499 : Représentation de la classe des trames.

Le comportement de l'opération **`Valeur()`** est décrit par l'automate suivant :

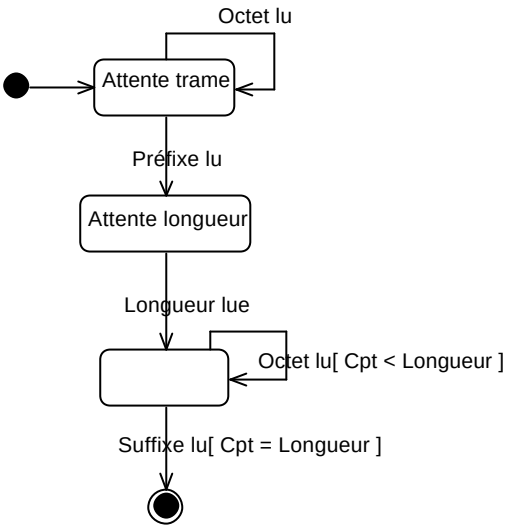


Figure 500 : Représentation du comportement de l'opération ***`Valeur()`***.

Architecture

Architecture logicielle

Ces lecteurs de badges peuvent être représentés par un acteur. Dans le diagramme suivant, le lecteur de badges est représenté par une classe stéréotypée pour insister sur le fait qu'il s'agit d'une classe de dispositifs matériels et non de personnes.

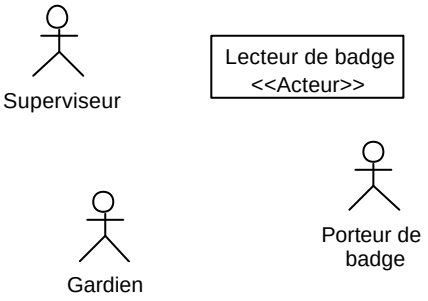


Figure 501 : Représentation des acteurs.

Le matériel utilisé permet de déporter le cas d'utilisation du contrôle d'accès vers les lecteurs de badges. Selon le point de vue retenu, le lecteur de badges se comporte comme un acteur pour le système de contrôle d'accès dans son ensemble ou comme un système indépendant avec lequel interagit le porteur de badge.

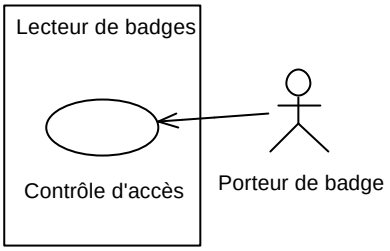


Figure 502 : Le cas d'utilisation du contrôle des accès est déporté vers les lecteurs de badges.

Du point de vue global, les lecteurs de badges apparaissent comme un acteur au même titre que le superviseur et le gardien. Le système est constitué de deux sous-système distincts. D'une part le logiciel spécifique à développer (pour exécution sur les PC) et d'autre part le système clé en main, livré avec les lecteurs de badges.

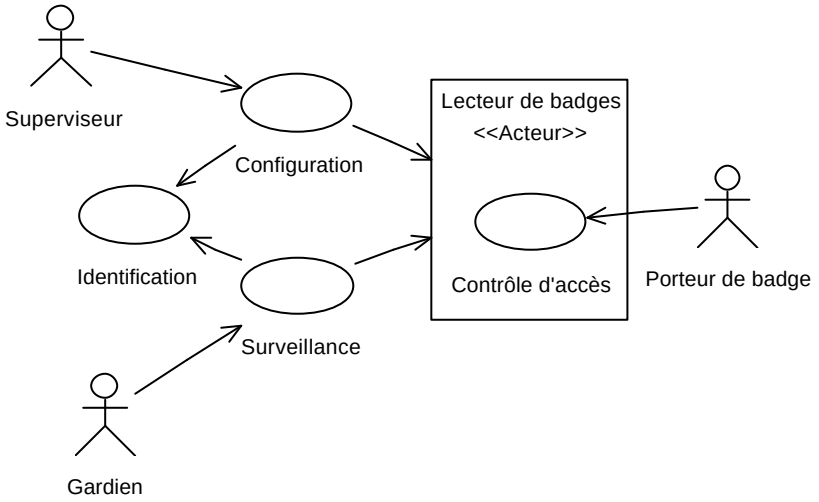


Figure 503 : Représentation définitive des cas d'utilisation.

La structure de la vue logique prend la forme suivante. Les objets métiers sont regroupés dans un paquetage **`Domaine`**. Les composants miroirs de ces objets du domaine sont contenus dans le paquetage **`IHM`**. La couche la plus basse

comprend une catégorie **`Persistance`** qui encapsule une base de donnée, un paquetage **`Machine virtuelle`** qui isole l'application des particularités du matériel et un paquetage **`Lecteur physique`** contenant des classes qui permettent la manipulation des lecteurs de badges et qui encapsulent toute la complexité de la communication.

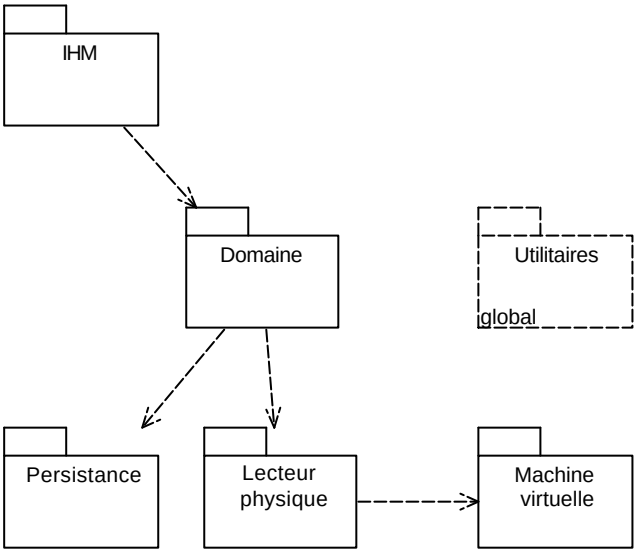


Figure 504 : Structure de la vue logique.

Architecture matérielle

Le système est constitué d'éléments logiciels et matériels, interchangeables dans une large mesure.

Les lecteurs de badges sont interconnectés au moyen d'un réseau spécifique, indépendant de l'Intranet. Le poste de travail du superviseur et la station de contrôle du gardien sont également connectés à ce réseau dédié au contrôle d'accès. Il peut y avoir jusqu'à 64 lecteurs de badges connectés au réseau.

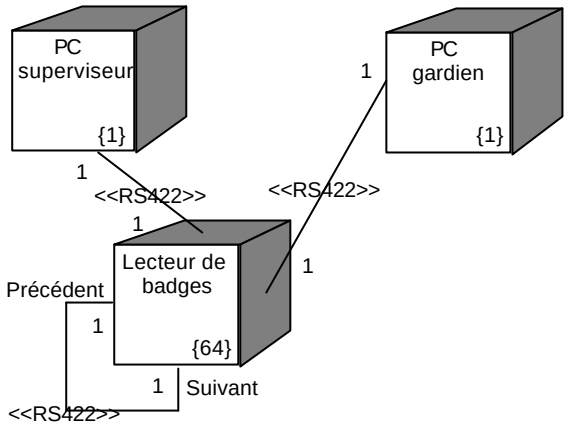


Figure 505 : Architecture matérielle.

Réalisation

Cette étude de cas a pour objectif de présenter la modélisation objet avec UML, de sorte que la réalisation n'est pas décrite ici en détail. Les grandes étapes de la réalisation comprennent :

- la génération automatique du schéma de la base de données à partir des classes du domaine désignées comme persistantes,
- la génération des écrans par un constructeur d'interfaces graphiques,
- la réalisation manuelle des interactions à partir des diagrammes de collaboration.

Annexes

A1

Les éléments standard

Les stéréotypes, les étiquettes et les contraintes sont les mécanismes mis à la disposition de l'utilisateur pour l'extension d'UML.

Stéréotypes prédéfinis

Les stéréotypes prédéfinis par UML sont présentés par ordre alphabétique dans le tableau suivant. Les trois colonnes contiennent respectivement le nom du stéréotype, l'élément de modélisation auquel il s'applique et la description détaillée de sa sémantique.

Nom	Sujet	Sémantique
acteur	type	abstraction externe au système en cours de modélisation
ami	dépendance	extension de la visibilité d'un paquetage au contenu d'un autre paquetage
appel	dépendance	opération qui appelle une autre opération
application	composant	composant qui représente un programme exécutable
besoin	note	note qui exprime un besoin ou une obligation
bibliothèque	composant	composant qui représente une bibliothèque statique ou dynamique
contrainte	note	transforme une note en contrainte
copie	dépendance	copie profonde d'une instance dans une autre

Nom	Sujet	Sémantique
dérivée	dépendance	la source est dérivée de la cible
devient	dépendance	transformation des caractéristiques d'une même instance
document	composant	composant qui représente un document
énumération	type primitif	ensemble d'identificateurs qui forment le domaine de valeur d'un type
envoi	dépendance	relation de dépendance entre une opération et un signal envoyé par l'opération
étend	généralisation	le cas d'utilisation source étend le comportement du cas d'utilisation cible
façade	paquetage	paquetage qui ne fait que référencer des éléments
fichier	composant	composant qui représente un fichier source
flot	classe active	classe active qui représente un flot de contrôle léger (thread)
import	dépendance	rend la partie publique d'un paquetage visible à un autre paquetage
instance	dépendance	relation entre une instance et son type
interface	type	vue d'un type
liaison	dépendance collaboration	relation de dépendance entre un type instancié ou une collaboration et un type modèle ou une collaboration modèle
métaclasse	dépendance type	relation de dépendance entre un type et sa métaclasse
page	composant	composant qui représente une page web
powertype	dépendance type	relation de dépendance entre une généralisation et un type dont les instances sont les sous-types qui participent à la généralisation
processus	classe active	classe active qui représente un flot de contrôle lourd
raffinement	dépendance	la source dérive de la cible et ajoute de l'information
rôle	dépendance	dépendance entre un type et un rôle d'association
signal	classe	classe qui représente un événement
souche	paquetage	paquetage entièrement transféré
sous-classe	généralisation	le sous-type hérite de la structure et du comportement du super-type, sans être un type du super-type
sous-type	généralisation	le sous-type hérite de la structure et du comportement

Nom	Sujet	Sémantique
		du super-type, et devient un type du super-type
table	composant	composant qui représente une table de base de données
trace	dépendance	relation de dépendance entre éléments de modèles différents
utilise	dépendance	le cas d'utilisation source utilise le comportement du cas d'utilisation cible
utilitaire	type	type non instanciable qui regroupe des opérations et des attributs

Etiquettes prédéfinies

Les étiquettes prédéfinies par UML sont présentées par ordre alphabétique dans le tableau suivant. Les quatre colonnes contiennent respectivement le nom de l'étiquette, le domaine de définition de sa valeur, l'élément de modélisation auquel elle s'applique et la description détaillée de sa sémantique.

Nom	Valeur	Sujet	Sémantique
documentation	chaîne	élément	commentaire, description ou explication
invariant	non interprété	type	prédicat qui doit être toujours vrai pour toutes les instances du type
localisation	composant nœud	élément de modélisation composant	assignation de la réalisation d'un élément de modélisation dans un composant assignation d'un composant sur un nœud
persistance	énumération {transitoire, persistant}	type instance attribut	permanence de l'état
post-condition	non interprété	opération	prédicat qui doit être vrai après l'invocation d'une opération
pré-condition	non interprété	opération	prédicat qui doit être vrai avant l'invocation d'une opération
responsabilité	chaîne	type	obligation liée à un type
sémantique	non interprété	type	spécification de la sémantique

		opération	
sémantique spatiale	non interprété	type opération	spécification de la complexité spatiale
sémantique temporelle	non interprété	type opération	spécification de la complexité temporelle

Contraintes prédéfinies

Les contraintes prédéfinies par UML sont présentées par ordre alphabétique dans le tableau suivant. Les trois colonnes contiennent respectivement le nom de la contrainte, l'élément de modélisation auquel elle s'applique et la description détaillée de sa sémantique.

Nom	Sujet	Sémantique
association	rôle de lien	l'instance correspondante est visible par une association
chevauchement inclusif	généralisation	les instances peuvent avoir plusieurs types parmi les sous-types
complète	généralisation	tous les sous-types ont été spécifiés
diffusion	message	l'ordre d'invocation des messages n'est pas spécifié
disjointe exclusif	généralisation	les instances n'ont qu'un seul type parmi les sous-types
global	rôle de lien	l'instance correspondante est visible, car placée dans une portée globale
implicite	association	l'association n'est pas manifeste, mais conceptuelle
incomplète	généralisation	tous les sous-types n'ont pas été spécifiés
local	rôle de lien	l'instance correspondante est visible, car elle est une variable locale d'une opération
ordonnée	rôle d'association	la collection, représentée par l'association, est ordonnée
ou	association	associations mutuellement exclusives
paramètre	rôle de lien	l'instance correspondante est visible, car elle est paramètre d'une opération
self	rôle de lien	l'instance correspondante est visible car elle participe à la distribution du message
vote	collection de messages	la valeur de retour est choisie par vote majoritaire parmi les valeurs retournées par la collection de messages

A2

Guide de transition de Booch et OMT vers UML

Ce guide a pour objectif de faciliter la transition des utilisateurs des notations de Booch'93 ou OMT-2 vers la notation UML.

Les notations de Booch, OMT et UML fournissent trois vues différentes de concepts objet très proches. En fait, les notations de Booch et OMT pourraient être utilisées pour représenter une grande partie des éléments de modélisation définis dans le métamodèle d'UML.

Graphiquement, UML est plus proche d'OMT que de Booch, car les icônes en forme de nuage ont été abandonnées au profit de rectangles plus faciles à dessiner. Au-delà des aspects graphiques, UML peut être considérée comme un sur-ensemble des deux autres notations.

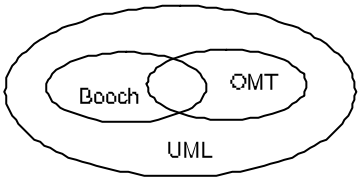


Figure 506 : UML est un sur-ensemble de Booch et OMT.

Les tableaux qui suivent illustrent les différences de notations concernant les principaux concepts objet.

Les mécanismes de base

Contraintes

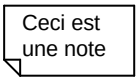
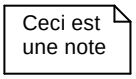
Dans les trois notations, les contraintes se représentent par des expressions entre accolades.

Booch	OMT	UML
Texte entre accolades {Ceci est une contrainte}	Texte entre accolades {Ceci est une contrainte}	Texte entre accolades {Ceci est une contrainte}

Notes

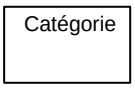
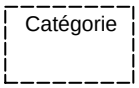
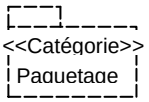
En Booch et en UML, les notes se représentent par des rectangles, avec un coin replié. En OMT, les notes se représentent comme les contraintes.

En Booch le coin bas-gauche est replié, alors qu'en UML, le coin haut-droit est replié.

Booch	OMT	UML
	texte entre accolades {Ceci est une note}	

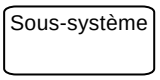
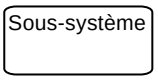
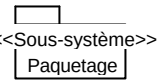
Catégories

En Booch et en OMT, les catégories font partie des éléments de modélisation. En UML, les catégories se réalisent par stéréotypage des paquetages.

Booch	OMT	UML
		

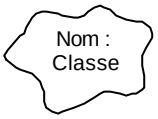
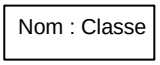
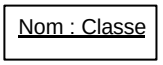
Sous-systèmes

En Booch et en OMT, les sous-systèmes font partie des éléments de modélisation. En UML, les sous-systèmes se réalisent par stéréotypage des paquetages.

Booch	OMT	UML
		

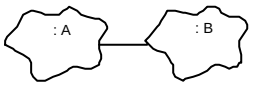
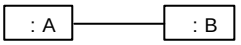
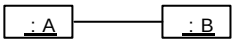
Les objets

En Booch, les objets se représentent par des nuages. En OMT, comme en UML, les objets se représentent par des rectangles. En UML, le nom des objets est souligné.

Booch	OMT	UML
		

Les liens

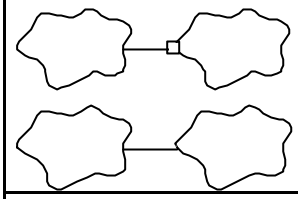
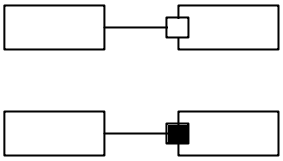
Dans les trois notations, les liens se représentent par une ligne continue, tracée entre les objets.

Booch	OMT	UML
		

Spécification de réalisation

Booch et UML permettent de préciser la construction retenue pour la réalisation d'un lien, au moyen d'un petit carré placé sur le rôle. Le symbole placé dans le carré indique la nature du lien. Un carré noir indique une réalisation par valeur, un carré blanc indique une réalisation par référence.

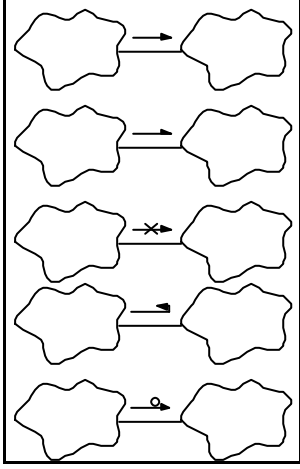
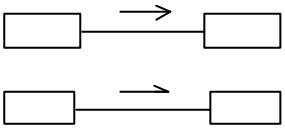
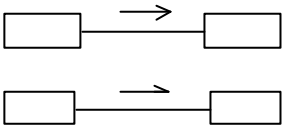
OMT ne propose pas d'attribut graphique particulier, en dehors des contraintes.

Booch	OMT	UML
		
F membre (*field*) G global L local P paramètre		A association F membre (*field*) G global L local P paramètre S self

Les messages

Dans les trois notations, les messages se représentent au moyen de flèches placées à proximité des liens.

La forme de synchronisation de l'envoi de message est symbolisée par une forme de flèche particulière.

Booch	OMT	UML
		 Stéréotypage pour les autres formes de synchronisation

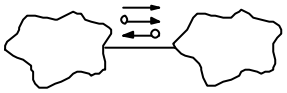
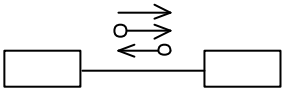
Ordre d'envoi

L'ordre d'envoi est représenté dans les trois notations par une expression placée en tête du message.

Booch	OMT	UML
Notation décimale	Notation décimale pointée	Notation décimale pointée modifiée `caractère` flot `chiffre` étape dans un flot * itération

Flots de données

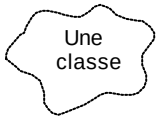
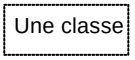
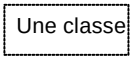
Booch et UML permettent de représenter les flots de données parallèlement aux flots de contrôle (les messages), au moyen de petits cercles accolés à une flèche dirigée dans le sens du flot de données. Cette notation est optionnelle, car redondante avec la représentation des paramètres des messages.

Booch	OMT	UML
	Par les paramètres des messages	

Les classes

En Booch, les classes se représentent par des nuages pointillés. En OMT, comme en UML, les classes se représentent par des rectangles.

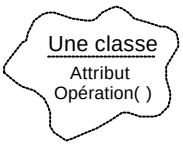
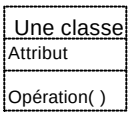
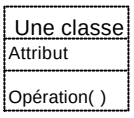
Classe simple

Booch	OMT	UML
		

Attributs et opérations

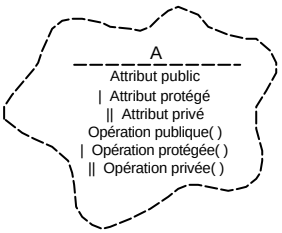
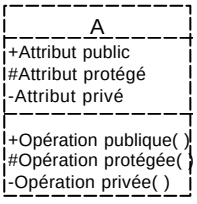
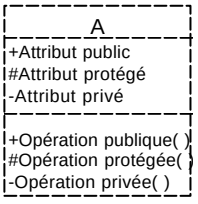
Dans les trois notations, les attributs et les opérations sont représentés au sein de l'icône de la classe. Certains attributs et certaines opérations peuvent être masqués pour ne pas surcharger les diagrammes.

OMT et UML proposent des compartiments pour distinguer les attributs des opérations.

Booch	OMT	UML
		

Visibilité

Les trois notations proposent les niveaux public, protégé et privé, avec la même sémantique que celle du langage C++. Booch possède en plus un niveau implémentation.

Booch	OMT	UML
		`rien` non spécifié
`rien` public	+ public	+ public
`|` protégé	# protégé	# protégé
`||` privé	- privé	- privé
`|||` implémentation		
`Métaclasse` classe	$ classe	`souligné` classe
		

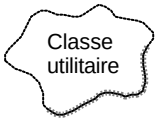
Classe abstraite

En Booch, les classes abstraites sont désignées par un petit triangle qui contient la lettre A. En OMT, une classe abstraite possède une multiplicité de valeur nulle. En UML, le nom des classes abstraites figure en italique.

Booch	OMT	UML
	Classe abstraite 0	*Classe abstraite*

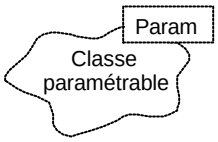
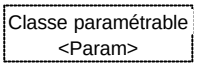
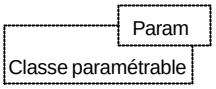
Classe utilitaire

En Booch et en UML, une classe utilitaire se représente comme une classe simple, avec en plus une bordure grisée. OMT ne propose pas d'attribut graphique particulier, en dehors des contraintes.

Booch	OMT	UML
	Classe utilitaire {utilitaire}	Classe utilitaire <<Utilitaire>> Classe utilitaire

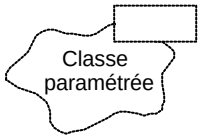
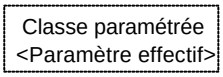
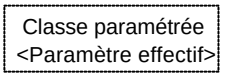
Classe paramétrable

Booch et UML proposent une notation spéciale, dérivée de la représentation des classes simples, avec en plus un petit rectangle pointillé qui contient les paramètres formels. OMT suffixe le nom des classes paramétrables par les paramètres formels.

Booch	OMT	UML
		

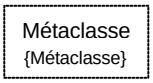
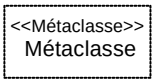
Classe paramétrée

L'instanciation d'une classe paramétrable donne une classe paramétrée. Booch propose une icône spéciale, avec un rectangle en trait continu ; le paramètre effectif peut être représenté au moyen d'une relation d'utilisation. OMT et UML représentent les classes paramétrées au moyen d'une classe simple en suffixant le nom de la classe par les valeurs des paramètres effectifs.

Booch	OMT	UML
		

Métaclasse

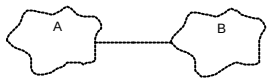
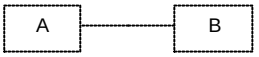
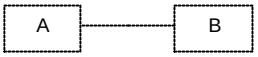
Booch représente une métaclasse au moyen d'une classe grisée. OMT ne possède pas d'attribut graphique particulier, en dehors des contraintes. UML stéréotype une classe simple pour obtenir une métaclasse.

Booch	OMT	UML
		

Les relations

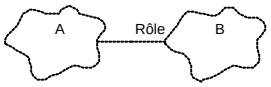
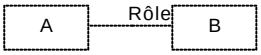
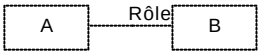
Association

Dans les trois notations, les associations se représentent par une ligne continue, tracée entre les classes qui participent à l'association.

Booch	OMT	UML
		

Rôle

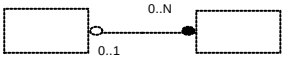
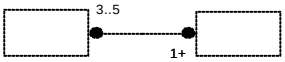
Dans les trois notations, les noms de rôles sont placés près des extrémités des relations.

Booch	OMT	UML
		

Multiplicité

Booch et UML sont identiques, à part pour la valeur illimitée qui se représente par **`N`** en Booch et **`*`** en UML. OMT propose une représentation graphique à base de cercles. Il faut faire attention à ne pas confondre ces cercles avec ceux des relation *has* et *use* de Booch, car il n'y aucun rapport entre les deux représentations.

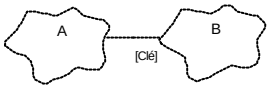
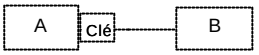
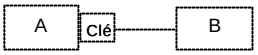
Booch	OMT	UML

`1`
`0..1`
`N`
`3..5, 7, 15`
`3..N`
	`1` par défaut
	 
	 
		`1`
		`0..1`
		`*`
		`3..5, 7, 15`
		`3..*`

Restriction

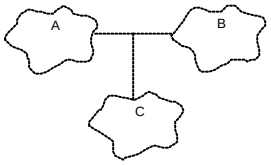
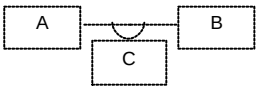
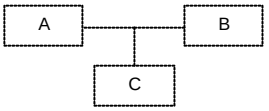
La restriction d'une association se représente en Booch au moyen d'une condition entre crochets. En OMT et en UML, une restriction se représente au moyen d'un compartiment rectangulaire.

La clé se représente du côté source en OMT et UML, et du côté destination en Booch.

Booch	OMT	UML
		

Classe-association

Dans les trois notations, une classe-association se représente au moyen d'une classe reliée à une association.

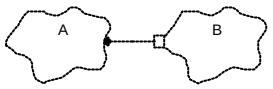
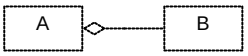
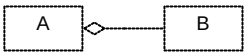
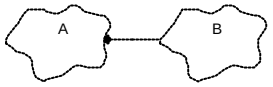
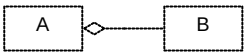
Booch	OMT	UML
		

Agrégation

Il n'y a pas d'équivalence stricte entre Booch d'une part, et OMT et UML d'autre part.

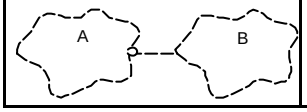
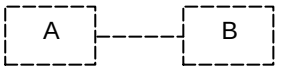
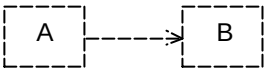
Du point de vue des agrégations, Booch est plus proche de la conception, OMT est plus proche de l'analyse, et UML couvre à la fois l'analyse et la conception.

Le tableau suivant représente les deux situations les plus courantes : l'agrégation par référence et l'agrégation par valeur (la composition en UML). Il faut noter que dans le cas d'OMT, la vue graphique ne distingue pas la forme d'agrégation.

Booch	OMT	UML
		
		

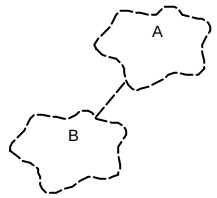
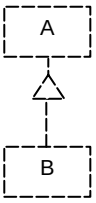
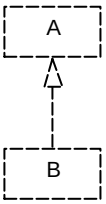
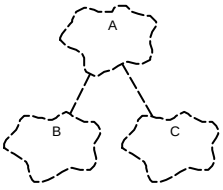
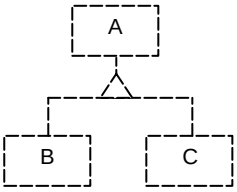
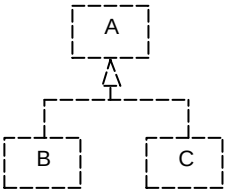
Dépendance

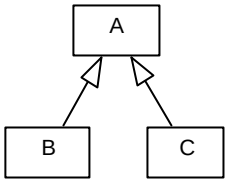
Booch représente la dépendance au moyen d'une association décorée d'un petit cercle, placé du côté client. OMT représente la dépendance au moyen d'une flèche en pointillé, à tête pleine. UML représente la dépendance au moyen d'une flèche en pointillé, à tête ouverte. En OMT et en UML, la flèche désigne le fournisseur.

Booch	OMT	UML
		

Héritage

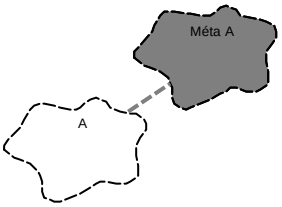
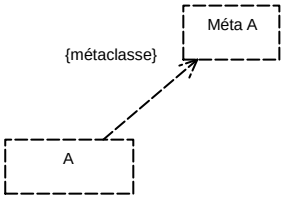
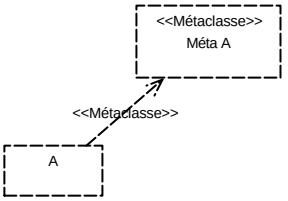
Dans les trois notations, l'héritage se représente au moyen d'une flèche qui pointe de la sous-classe vers la super-classe.

Booch	OMT	UML
		
		

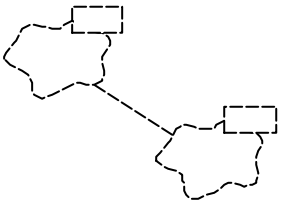
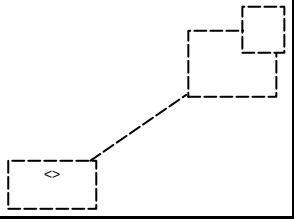
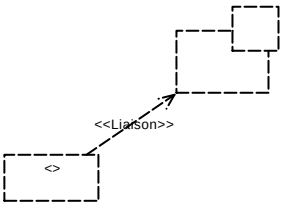
Instanciation de Métaclasse

Booch emploie une flèche grisée pour représenter la relation ente une classe et sa métaclasse. OMT ne prévoit pas de notation particulière, en dehors des contraintes. UML stéréotype une relation de dépendance.

Booch	OMT	UML
		

Instanciation de générique

Booch propose une flèche pointillée pour représenter la relation entre les classes paramétrées et les classes paramétrables. OMT emploie une relation de dépendance. UML stéréotype une relation de dépendance.

Booch	OMT	UML
		

Construction dérivée

Les associations et les attributs dérivés sont préfixés par le caractère **/** en OMT comme en UML. Cette notion n'existe pas en Booch.

Booch	OMT	UML
	/	/

A3

Génération de code C++

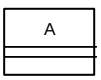
Les exemples de code qui suivent ont été générés automatiquement par l'outil Rational Rose 4.0, à partir de modèles UML. Ces exemples n'illustrent pas l'ensemble des capacités de génération de code de Rose, mais décrivent les grandes lignes des correspondances entre UML et le langage C++.

Classe

Classe vide

Le générateur de code a été configuré pour générer un constructeur, un constructeur par copie, un destructeur, un opérateur d'affectation et deux opérateurs d'égalité.

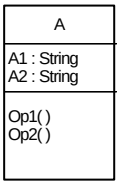
Ces opérations ne sont pas représentées dans les paragraphes suivants, afin de ne pas surcharger les exemples.




```
#ifndef A_h
#define A_h 1
class A
{
  public:
    //## Constructors (generated)
      A();
      A(const A &right);
    //## Destructor (generated)
```

```
    ~A();
  //## Assignment Operation (generated)
    const A & operator=(const A &right);
  //## Equality Operations (generated)
    int operator==(const A &right) const;
    int operator!=(const A &right) const;
};
#endif
```

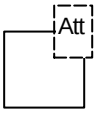
Classe avec attributs et opérations




```
class A
{
  public:
    ...
  //## Other Operations (specified)
    void Op1();
    void Op2();

    const String get_A1() const;
    void set_A1(const String value);
    const String get_A2() const;
    void set_A2(const String value);

  private:
    String A1;
    String A2;
};

inline const String A::get_A1() const
{
  return A1;
}

inline void A::set_A1(const String value)
{
  A1 = value;
}

inline const String A::get_A2() const
{
```

```
  return A2;
}

inline void A::set_A2(const String value)
{
  A2 = value;
}
```

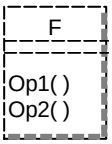
Classe paramétrable




```
template <argtype Att>
class D
{
  public:
    D();
    D(const D<Att> &right);
    ~D();

    const D<Att> & operator=(const D<Att>
&right);

    int operator==(const D<Att> &right) const;
    int operator!=(const D<Att> &right) const;
...
};
```

Classe utilitaire

Toutes les opérations d'une classe utilitaire sont préfixées par le mot-clé **`static`**.

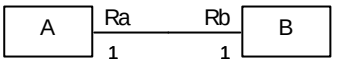



```
class F
{
  public:
    static void Op1();
    static void Op2();
 };
```

Association

Association 1 vers 1

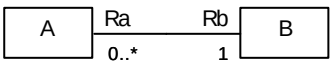
Le générateur de code réalise l'association par des pointeurs placés dans les parties privées des classes qui participent à l'association.




```
...
class A
{
 ...

     const B * get_Rb() const;
     void set_Rb(B *const  value);

  private:
       B *Rb;
};

inline const B * A::get_Rb() const
{
   return Rb;
}

inline void A::set_Rb(B *const
value)
{
  Rb = value;
}
```

```
...
class B
{
 ...

     const A * get_Ra() const;
     void set_Ra(A *const  value);

  private:
       A *Ra;
};

inline const A * B::get_Ra() const
{
   return Ra;
 }

inline void B::set_Ra(A *const
value)
{
  Ra = value;
}
```

Association N vers 1

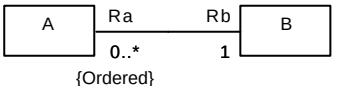


Le générateur de code réalise l'association par des pointeurs placés dans les parties privées des classes qui participent à l'association. La multiplicité **`0..*`** est réalisée par un ensemble de taille non contrainte.

```
...
class B
{
 ...
     const UnboundedSetByReference<A> get_Ra() const;
     void set_Ra(const UnboundedSetByReference<A> value);

  private:
       UnboundedSetByReference<A> Ra;
 };

inline const UnboundedSetByReference<A> B::get_Ra() const
{
  return Ra;
}

inline void B::set_Ra(const UnboundedSetByReference<A> value)
{
  Ra = value;
}
```

Association N vers 1 avec une contrainte

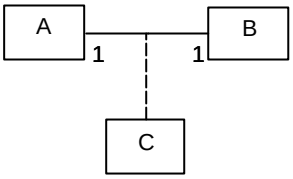
Le générateur de code réalise l'association par des pointeurs placés dans les parties privées des classes qui participent à l'association. Du fait de la contrainte **`{Ordered}`**, la multiplicité **`0..*`** est réalisée par une liste de taille non contrainte.




```
...
class B
{
  ...
     const UnboundedListByReference<A> get_Ra() const;
     void set_Ra(const UnboundedListByReference<A> value);

   private:
     UnboundedListByReference<A> Ra;
```

```
};
```

Classe-association

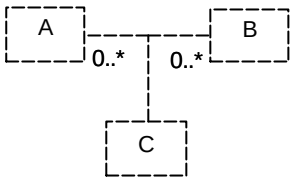



```
class A; class B;

class C
{
  ...
    const B * get_the_B() const;
    void set_the_B(B *const  value);

    const A * get_the_A() const;
    void set_the_A(A *const
value);

 private:
    A *the_A;
    B *the_B;
};
```

```
#include "C.h"
class A
{
  ...
    const C * get_the_C() const;
    void set_the_C(C *const  value);

 private:
    C *the_C;
};
```

```
#include "C.h"
class B
{
  ...
    const C * get_the_C() const;
    void set_the_C(C *const  value);

 private:
    C *the_C;
};
```

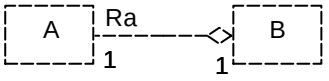
Classe-association N vers N




```
#include "C.h"
class B
{
   ...
     const UnboundedSetByReference<C> get_the_C() const;
     void set_the_C(const UnboundedSetByReference<C> value);

  private:
     UnboundedSetByReference<C> the_C;
};
```

Agrégation

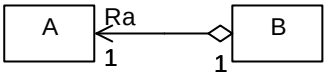
Agrégation 1 vers 1




```
#include "B.h"
class A
{
...
     const B * get_the_B() const;
     void set_the_B(B *const  value);

  private:
     B *the_B;
};
```

```
#include "A.h"
class B
{
   ...
     const A * get_Ra() const;
     void set_Ra(A *const  value);

  private:
     A *Ra;
};
```

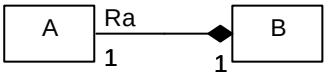
Agrégation à navigabilité restreinte




```
class A
{
  ...
 private:
};
```

```
#include "A.h"
class B
{
   ...
     const A * get_Ra() const;
     void set_Ra(A *const  value);
 private:
     A *Ra;
};
```

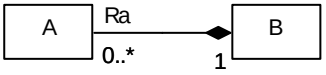
Agrégation par valeur




```
#include "B.h"
class A
{
...
     const B * get_the_B() const;
     void set_the_B(B *const  value);

 private:
     B *the_B;
};
```

```
#include "A.h"
class B
{
...
     const A get_Ra() const;
     void set_Ra(const A value);

 private:
     A Ra;
};
```

Agrégation par valeur 1 vers N

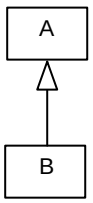



```
#include "A.h"
class B
```

```
{
  ...
    const UnboundedSetByValue<A> get_Ra() const;
    void set_Ra(const UnboundedSetByValue<A> value);

  private:
    UnboundedSetByValue<A> Ra;
};
```

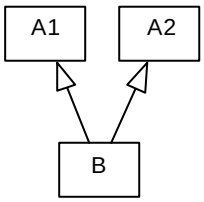
Héritage

Héritage simple




```
#include "A.h"
class B : public A
{
   ...
};
```

Héritage multiple




```
#include "A1.h"
#include "A2.h"
class B : public A2, public A1
{
   ...
};
```

A4

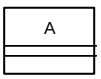
Génération de code Java

Les exemples de code qui suivent ont été générés automatiquement par l'outil Rational Rose 4.0, à partir de modèles UML. Ces exemples n'illustrent pas l'ensemble des capacités de génération de code de Rose, mais décrivent les grandes lignes des correspondances entre UML et le langage Java.

Classe

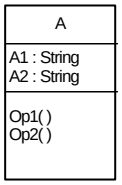
Classe vide

Le générateur de code a été configuré pour générer un constructeur et un destructeur. Ces opérations ne sont pas représentées dans les paragraphes suivants, afin de ne pas surcharger les exemples.



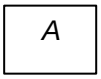

```
public final class A {

   public A() {
      super();
     ...
   }

   protected void finalize() throws Throwable {
      super.finalize();
     ...
   }
   ...
}
```

Classe avec attributs et opérations



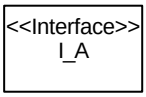

```
public final class A {

   private String m_A1;
   private String m_A2;

   public void Op1() {
   ...
   }

   public void Op2() {
   ...
   }
```

Classe abstraite




```
public abstract class A {
   ...
}
```

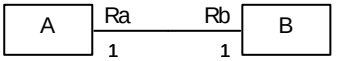
Interface




```
public interface I_A {
   ...
}
```

Association

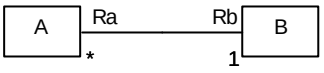
Association 1 vers 1




```
public class A {

   public B m_Rb;

}
```

```
public class B {

   public A m_Ra;

}
```

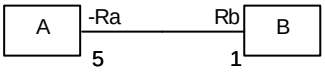
Association N vers 1




```
public class B {

  public Vector m_Ra = new
Vector();

}
```


```
public class B {

   private Vector m_Ra = new
Vector();

}
```

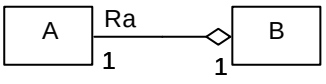
Lorsque la multiplicité est limitée l'association est réalisée par un tableau.




```
public class B {

    private A[] m_Ra = new A[5];

}
```

Agrégation

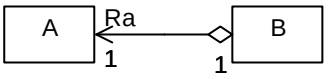
Agrégation 1 vers 1




```
public class A {

    public B m_B;

}
```

```
public class B {

    public A m_Ra;

}
```

Agrégation à navigabilité restreinte

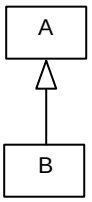



```
public class A {
   ...
}
```

```
public class B {

   public A m_Ra;

}
```

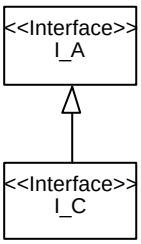
Héritage

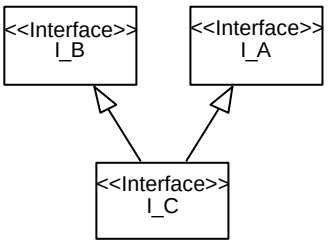
Héritage simple




```
public class B extends A {
   ...
}
```

Héritage entre interfaces

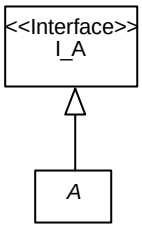





```
public interface I_C extends I_A {
   ...
}
```

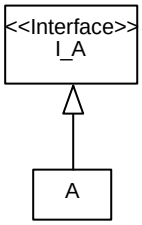
```
public interface I_C extends I_A, I_B
{
   ...
}
```

Réalisation d'une interface par une classe abstraite



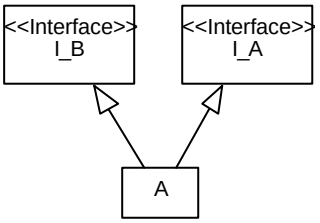

```
public abstract class A implements
I_A {
  ...
}
```

Réalisation d'une interface par une classe




```
public class A implements I_A {
   ...
}
```

Réalisation de plusieurs interfaces par une classe




```
public class A implements I_A, I_B {
   ...
}
```

A5

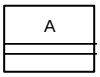
Génération de code IDL

Les exemples de code qui suivent ont été générés automatiquement par l'outil Rational Rose 4.0, à partir de modèles UML. Ces exemples n'illustrent pas l'ensemble des capacités de génération de code de Rose, mais décrivent les grandes lignes des correspondances entre UML et le langage IDL.

Classe

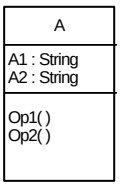
Classe vide

Une classe est traduite en interface IDL.




```
interface A {
  ...
};
```

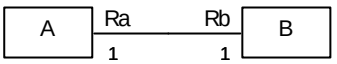
Classe avec attributs et opérations




```
interface A {
  attribute String Att1;
  attribute String Att2;

  void Op1();
  void Op2();
};
```

Association

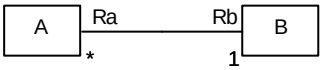
Association 1 vers 1




```
interface A {
  attribute B Rb;
};
```

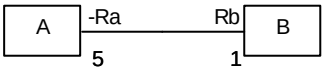
```
interface B {
  attribute A Ra;
};
```

Association N vers 1




```
interface B {
  attribute sequence<A> Ra;
};
```

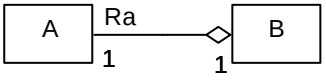
Association 5 vers 1




```
interface B {
  attribute sequence<A,5> Ra;
};
```

Agrégation

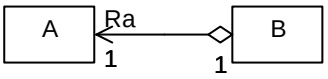
Agrégation 1 vers 1



```
interface A {

  attribute B the_B;

};
```

```
interface B {

  attribute A Ra;

};
```

Agrégation à navigabilité restreinte

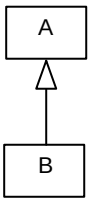



```
interface A {

};
```

```
interface B {

  attribute A Ra;

};
```

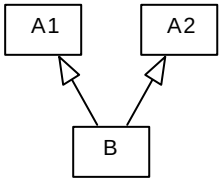
Héritage

Héritage simple




```
interface B : A {

};
```

Héritage multiple




```
interface B : A1, A2 {

};
```

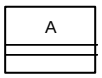
A6

Génération de code Visual Basic

Les exemples de code qui suivent ont été générés automatiquement par l'outil Rational Rose 4.0, à partir de modèles UML. Ces exemples n'illustrent pas l'ensemble des capacités de génération de code de Rose, mais décrivent les grandes lignes des correspondances entre UML et le langage Visual Basic.

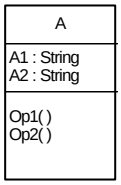
Classe

Classe vide



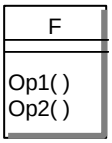

```
Option Base 0

Private Sub Class_Initialize()
End Sub

Private Sub Class_Terminate()
End Sub
```

Classe avec attributs et opérations




```
Option Base 0

Public A1 As String

Public A2 As String

Private Sub Class_Initialize()
End Sub

Private Sub Class_Terminate()
End Sub

Public Sub Op1()
   On Error GoTo Op1Err
   ...

   Exit Sub
Op1Err:
   Call RaiseError(MyUnhandledError, "A:Op1
Method")
End Sub

Public Property Get Op2() As Boolean
   On Error GoTo Op2Err
   ...

   Exit Property
Op2Err:
   Call RaiseError(MyUnhandledError, "A:Op2
Property")
End Property
```

Classe utilitaire

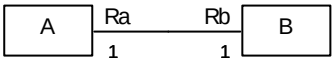



```
Option Base 0

Private Sub Class_Initialize()
End Sub
Private Sub Class_Terminate()
End Sub
```

```
Public Sub Op1()
End Sub

Public Property Get Op2() As Boolean
End Property
```

Association

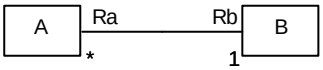
Association 1 vers 1




```
Option Base 0

Public Rb As B

Private Sub Class_Terminate()
End Sub

Private Sub Class_Initialize()
End Sub
```

```
Option Base 0

Public Ra As A

Private Sub Class_Terminate()
End Sub

Private Sub Class_Initialize()
End Sub
```

Association N vers 1

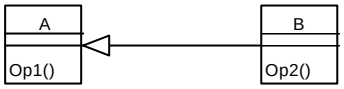



```
Option Base 0

Public Rb As B

Private Sub Class_Initialize()
End Sub

Private Sub Class_Terminate()
End Sub
```

```
Option Base 0

Public Ra As Collection

Private Sub Class_Initialize()
End Sub

Private Sub Class_Terminate()
End Sub
```

Héritage

Héritage simple




```
Option Base 0
Implements A

'local superclass object (generated)
Private mAObject As New A

Private Sub Class_Initialize()
End Sub

Private Sub Class_Terminate()
End Sub

Public Sub Op2()
End Sub

Private Sub A_Op1()
   mAObject.Op1
End Sub
```

A7

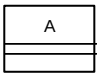
Génération de code SQL

Les exemples de code qui suivent ont été générés automatiquement par l'outil Rational Rose 4.0, à partir de modèles UML. Ces exemples n'illustrent pas l'ensemble des capacités de génération de code de Rose, mais décrivent les grandes lignes des correspondances entre UML et le langage SQL ANSI.

Pour une discussion détaillée sur la génération de code SQL à partir d'un diagramme de classes et les différentes stratégies possibles, voir le chapitre 17 du livre Object-Oriented Modeling and Design[48].

Classe

Classe vide

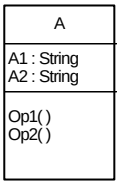



```
CREATE TABLE T_A (
   A_Id NUMBER (5),
   PRIMARY KEY (A_Id)
)
```

[48] Rumbaugh J., Blaha M., Premerlani W., Eddy F., Lorensen W. 1991, *Object-Oriented Modeling and Design*. Prentice Hall.

Classe avec attributs et opérations

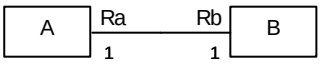
Le générateur construit la structure statique, les opérations sont ignorées.




```
CREATE TABLE T_A (
   A_Id NUMBER (5),
   Att1 VARCHAR (),
   Att2 VARCHAR (),
   PRIMARY KEY (A_Id)
)
```

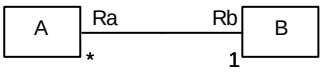
Association

Association 1 vers 1




```
CREATE TABLE T_B (
   B_Id NUMBER (5),
   PRIMARY KEY (B_Id)
)

CREATE TABLE T_A (
   A_Id NUMBER (5),
   B_Id NUMBER (5) REFERENCES T_B (B_Id),
   PRIMARY KEY (A_Id)
)
```

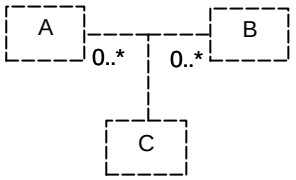
Association N vers 1




```
CREATE TABLE T_B (
   B_Id NUMBER (5),
```

```
    PRIMARY KEY (B_Id)
)

 CREATE TABLE T_A (
    B_Id NUMBER (5) REFERENCES T_B (B_Id),
    A_Id NUMBER (5),
    PRIMARY KEY(A_Id)
)
```

Classe-association N vers N

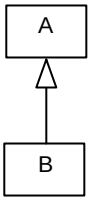



```
CREATE TABLE T_A (
    A_Id NUMBER (5),
    PRIMARY KEY (A_Id)
)

CREATE TABLE T_B(
    B_Id NUMBER (5),
    PRIMARY KEY (B_Id)
)

CREATE TABLE T_C (
    A_Id NUMBER (5) REFERENCES T_A (A_Id) ON DELETE CASCADE,
    B_Id NUMBER (5) REFERENCES T_B (B_Id) ON DELETE CASCADE,
    PRIMARY KEY(A_Id, B_Id)
)
```

Héritage

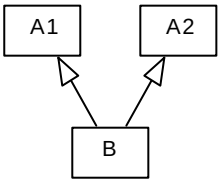
Dans les exemples suivants chaque classe est réalisée par une table. L'identité d'un objet est préservée dans la hiérarchie de classes par l'emploi d'un identifiant partagé.

Héritage simple




```
CREATE TABLE T_A(
   A_Id NUMBER(5),
   PRIMARY KEY(A_Id)
)

CREATE TABLE T_B(
   A_Id NUMBER (5) REFERENCES T_A (A_Id),
   PRIMARY KEY(A_Id))
)
```

Héritage multiple




```
CREATE TABLE T_A1(
   A1_Id NUMBER(5),
   PRIMARY KEY(A1_Id))

CREATE TABLE T_A2(
   A2_Id NUMBER(5),
   PRIMARY KEY(A2_Id)
)

CREATE TABLE T_B(
   A1_Id NUMBER (5) REFERENCES T_A1 (A1_Id),
   A2_Id NUMBER (5) REFERENCES T_A2 (A2_Id),
   PRIMARY KEY(A1_Id ,A2_Id)
 )
```

Glossaire

Abstraction Faculté des humains de se concentrer sur l'essentiel et d'oublier les détails. Parfois employé comme synonyme de classe.

Abstraite Se dit d'une classe qui ne peut pas être instanciée directement.

Acteur 1) Classe de personnes ou de systèmes qui interagissent avec un système.

2) Objet toujours à l'origine d'une interaction.

Action Opération exécutée instantanément lors d'une transition d'un état vers un autre état. Une action ne peut pas être interrompue.

Activité Opération exécutée au sein d'un état. Une activité est interruptible.

Agent Objet qui peut être origine et destination d'une interaction.

Agrégation Forme d'association non symétrique qui exprime un couplage fort et une relation de subordination.

Algorithme Enchaînement des actions nécessaires à une tâche.

Analyse Détermination du *quoi* et du *quoi faire* d'une application.

Analyse des besoins Détermination des besoins des utilisateurs.

Analyse du domaine Partie de l'analyse qui se concentre sur l'environnement de l'application.

Ancêtre Synonyme de super-classe.

Application Système logiciel élaboré dans un but précis.

Architecte Spécialiste de l'architecture.

Architecture 1) Art de construire ; en informatique, art de construire les logiciels.

	2) Structure d'un logiciel.
Artefact	Elément d'information, produit ou consommé par une des activités lors du développement d'un logiciel.
Association	Relation entre classes qui décrit un ensemble de liens.
Association dérivée	Association déduite d'autres associations.
Asynchrone	Forme de communication non bloquante et sans accusé de réception.
Attribut	Information contenue par un objet.
Attribut de lien	Attribut appartenant à un lien entre objets, plutôt qu'aux objets eux-mêmes.
Attribut dérivé	Attribut déduit d'autres attributs.
Automate	Forme de représentation abstraite du comportement.
Big bang	Se dit de la technique d'intégration lorsqu'elle est concentrée en fin de développement.
Cardinalité	Nombre d'éléments dans une collection.
Cas d'utilisation	Technique d'élaboration des besoins fonctionnels, selon le point de vue d'une catégorie d'utilisateurs.
Catégorie	Sorte de paquetage de la vue logique ; une catégorie encapsule des classes.
Classe	Description abstraite d'un ensemble d'objets ; réalisation d'un type.
Classe abstraite	Classe qui ne s'instancie pas directement et qui sert uniquement de spécification.
Classe active	Classe dont les objets sont actifs.
Classe concrète	Par opposition aux classes abstraites, classe qui s'instancie pour donner des objets.
Classe de base	Classe racine d'une hiérarchie de classes.
Classe paramétrable	Classe modèle pour la construction de classes.
Classe paramétrée	Classe résultant de la paramétrisation d'une classe paramétrable.
Classe racine	Classe racine d'une hiérarchie de classes.
Classe utilitaire	Classe dégradée, réduite au concept de module.
Classe-association	Association promue au rang de classe.
Classification	Action d'ordonner dans le but de comprendre.
Classification dynamique	Forme de classification par laquelle un objet peut changer de type ou de rôle.
Classification	Forme de classification par laquelle un objet ne peut pas

statique changer de type ou de rôle.

Clé Tuple des attributs qui réalisent la restriction d'une association.

Clé naturelle Clé primaire issue du domaine de l'application.

Clé primaire Information désignant un objet de manière bijective.

Client Objet à l'origine d'une requête.

Collaboration 1) Se dit d'une interaction entre objets réalisée dans le but de satisfaire un besoin de l'utilisateur.

2) Elément structurant d'UML pour la description du contexte d'une interaction.

Collection Terme générique désignant tous les regroupements d'objets sans préciser la nature du regroupement.

Composant Elément physique constitutif d'une application, représenté principalement dans la vue de réalisation.

Composition 1) Forme d'organisation complémentaire de la classification.

2) Agrégation par valeur.

Conception Détermination du *comment* d'une application.

Condition Expression booléenne qui valide ou non une transition dans un automate d'états finis.

Constructeur Opération de classe qui construit des objets.

Construction Phase de la vue de l'encadrement dans laquelle le logiciel est réalisé et amené à un état de maturité suffisant pour être livré à ses utilisateurs.

Conteneur Structure de données qui contient des objets.

Contexte Ensemble d'éléments de modélisation qui sert de support à une interaction.

Contrainte Expression qui précise le rôle ou la portée d'un ou plusieurs éléments de modélisation.

Contrat Engagement entre classes rédigé selon les termes de la spécification du fournisseur.

Contrôle Synonyme de flot de contrôle.

Couche Segmentation horizontale des modèles.

Couplage Dépendance entre éléments de modélisation.

CRC Abréviation de *Class Responsibilities Collaborators.*

Cycle Passage complet par les quatre phases de la vue de l'encadrement.

Cycle de vie Etapes du développement et ordonnancement de ces étapes.

Décomposition Séparation en éléments plus petits dans le but de réduire la complexité.

Délégation	Mécanisme de communication entre objets qui permet à un objet fournisseur de faire réaliser des tâches par un objet sous-traitant, sans le dire à l'objet client.
Dépendance	Relation d'obsolescence entre deux éléments de modélisation.
Déploiement	Phase de la vue de l'encadrement qui comprend la transition de l'application dans son environnement final.
Destructeur	Opération de classe qui détruit des objets.
Diagramme	Représentation graphique d'éléments de modélisation.
Discret	Contraire de continu.
Discriminant	Synonyme de critère.
Documentation	Description textuelle des modèles.
Domaine	Synonyme de champ d'application.
Elaboration	Phase de la vue de l'encadrement dans laquelle l'architecture est définie.
Elément	Brique de base d'un modèle.
Elément de modélisation	Représentation d'une abstraction issue du domaine du problème.
Elément de visualisation	Projection graphique d'une collection d'éléments de modélisation.
Encapsulation	Occultation des informations contenues par un objet.
Entité	Terme emprunté aux méthodes de modélisation par les données ; il est employé comme synonyme d'objet du domaine.
Enumération	Liste de valeurs nommées ; domaine de définition d'un type.
Erreur	Occurrence anormale.
Espace de nom	Partie de modèle dans laquelle un nom peut être défini : au sein d'un espace de nom, un nom ne possède qu'une seule signification.
Espace des états	Ensemble des états possibles.
Etat	Condition instantanée dans laquelle se trouve un objet, un système.
Etend	Relation d'extension entre cas d'utilisation.
Etude d'opportunité	Phase de la vue de l'encadrement dans laquelle la vision du produit est définie.
Evénement	Occurrence qui engendre un changement d'état.
Exception	Condition exceptionnelle qui correspond à une anomalie lors de l'exécution.
Export	Partie visible d'un paquetage.

Expression Chaîne dont l'interprétation donne une valeur d'un type donné.

Extension Forme de programmation qui facilite la prise en compte de nouveaux besoins.

Factorisation Identification puis extraction de similitudes entre classes.

Flot d'exécution Description de la répartition de l'activité entre les objets.

Flot de contrôle Synonyme de flot d'exécution.

Flot de données Description des données qui transitent d'un objet vers un autre objet.

Fonction Expression d'un besoin en termes impératifs, sous la forme de tâches à accomplir.

Framework Macro-architecture générique.

Garde Condition booléenne qui valide une transition dans un automate.

Généralisation Point de vue ascendant porté sur une classification ; nom donné par UML à la relation de classification utilisée pour construire des hiérarchies de classes ; souvent synonyme d'héritage, bien que plus abstrait.

Généralisation multiple Forme de généralisation dans laquelle une classe dérive de plusieurs ancêtres. Souvent synonyme d'héritage multiple, bien que plus abstrait.

Génération Dernière version exécutable produite par un cycle de développement.

Générique Solution générale ; se dit aussi de composants modèles.

Gestion de versions Enregistrement de l'histoire d'un élément.

Gestion de configurations Etablissement de relations entre les constituants d'un système dans le but de maintenir la cohérence.

Héritage Relation entre classes qui permet le partage de propriétés définies dans une classe ; principale technique de réalisation de la généralisation.

Héritage d'interface Héritage de l'interface d'un élément plus général sans sa réalisation.

Héritage de (pour la) réalisation Héritage de l'interface et de la réalisation d'un élément plus général.

Héritage multiple Relation entre classes qui permet le partage de propriétés définies dans plusieurs classes.

Hiérarchie Arborescence de classes ordonnées par une relation de généralisation.

Identité Caractéristique fondamentale d'un objet qui le distingue de tous les autres objets.

Idiome Construction élégante, propre à un langage donné.

Import Relation de dépendance entre paquetages qui rend visible les

	exports d'un paquetage au sein d'un autre paquetage.
Instance	Synonyme d'objet ; un objet est instance d'une classe.
Intégration	Qualité de l'interdépendance entre éléments de modélisation.
Interaction	Description d'un comportement dans le contexte d'une collaboration.
Interface	Partie visible d'une classe, d'un groupe d'objets ; parfois synonyme de spécification.
Interruption	Déroutement du flot d'exécution normal, consécutif à la détection d'une condition matérielle.
Invariant	1) Expression booléenne dont le changement de valeur déclenche une exception. 2) Critère pour la détection des objets ; un objet est un invariant du domaine.
Invocation	Mécanisme par lequel un message déclenche une opération.
Itérateur	Objet ou mécanisme qui permet de visiter tous les éléments d'une collection, sans en dévoiler la structure interne.
Itération	1) Action de traverser une collection d'objets. 2) Séquence d'activités de la vue technique qui aboutit à la livraison d'un prototype exécutable.
Liaison dynamique	Association entre un nom d'objet et une classe réalisée à l'exécution.
Liaison statique	Association entre un nom d'objet et une classe réalisée à la compilation.
Lien	Connexion sémantique entre un tuple d'objets par laquelle un objet peut communiquer avec un autre objet.
Ligne de vie	Représentation de l'existence d'un objet dans un diagramme de séquence.
Livraison	Résultat exécutable d'une itération ; parfois synonyme de prototype.
Maintenance	Phase du cycle de vie du logiciel qui suit le déploiement ; la maintenance regroupe des activités de correction de défauts et de prise en compte des demandes d'évolution.
Mécanisme	Synonyme de collaboration entre objets.
Membre	Terminologie C++ pour désigner un attribut ou une opération contenue par une classe.
Message	Elément de communication entre objets qui déclenche une activité dans l'objet destinataire ; la réception d'un message correspond à un événement.
Métaclasse	Classe d'une classe ; elle contient les données et les opérations qui concernent la classe plutôt que les instances de la classe.

Métamodèle Modèle qui décrit des éléments de modélisation.

Métamodélisation Modélisation récursive des éléments de modélisation à partir d'eux-mêmes.

Méthode 1) Souvent synonyme d'opération ; quelquefois utilisé pour distinguer la spécification de l'opération des multiples réalisations – les méthodes – implantées dans les sous-classes.

2) Ensemble de démarches raisonnées pour parvenir à un but.

Méthode d'instance Opération qui concerne les objets.

Méthode de classe Opération qui concerne la classe plus que les objets.

Mode Caractérise les paramètres selon la direction du flot de données : en entrée, en entrée et en sortie, en sortie.

Modèle Construction descriptive à partir des éléments de modélisation.

Modélisation Synonyme d'analyse ; par extension, élaboration des modèles, y compris de conception.

Modificateur Opération qui modifie l'état interne d'un objet.

Modularité Qualité d'un environnement de réalisation qui permet la partition.

Module Espace lexical dans lequel d'autres constructions peuvent être déclarées.

Monomorphisme Concept de la théorie des types, selon lequel un nom ne peut référencer que des objets de la même classe.

Multiplicité Désigne le nombre d'objets qui peuvent participer à une relation.

Navigabilité Qualité d'une relation qui permet le passage d'une classe vers une autre classe.

Niveau de maturité Description de la qualité d'un processus de développement.

Nœud Dispositif matériel susceptible d'exécuter un programme.

Non interprété Représentation d'un type non spécifié par UML, sous la forme d'une chaîne.

Notation Ensemble des signes et symboles qui composent un langage. Dans le cas d'UML, ensemble des représentations graphiques et textuelles qui constituent les diagrammes.

Note Information textuelle qui peut être associée à tout élément ou combinaison d'éléments de modélisation ; la note appartient à la vue, elle ne véhicule aucune sémantique.

Objet Entité atomique constituée d'un état, d'un comportement et d'une identité.

Objet actif Objet qui possède son propre flot de contrôle ; instance d'une classe active.

Occultation Synonyme d'encapsulation.

d'information

Opération	Elément de comportement des objets, défini de manière globale dans la classe.
Opération abstraite	Opération définie dans une classe, mais dont la réalisation est reportée dans une sous-classe.
Paquetage	Elément d'organisation des modèles.
Partie privée	Partie de la spécification d'une classe qui regroupe des propriétés invisibles de l'extérieur.
Partie protégée	Partie de la spécification d'une classe qui regroupe des propriétés invisibles de l'extérieur, sauf des sous-classes.
Partie publique	Partie de la spécification d'une classe qui regroupe des propriétés visibles de l'extérieur.
Partition	Segmentation verticale des modèles ; par extension, sous-ensemble d'un modèle.
Pattern	Micro-architecture ; élément de conception (ou d'analyse) récurrent.
Persistance	Qualité d'un objet à transcender le temps ou l'espace.
Phase	Ensemble des activités entre deux points de contrôle d'un processus de développement.
Polymorphisme	Concept de la théorie des types, selon lequel un nom peut référencer des objets instances de plusieurs classes regroupées dans une hiérarchie de généralisation.
Post-condition	Condition booléenne qui doit être vraie après l'exécution d'une opération.
Pré-condition	Condition booléenne qui doit être vraie avant l'exécution d'une opération.
Processus	1) Flot d'exécution lourd. 2) Suite d'étapes, plus ou moins ordonnées, dédiées à la satisfaction d'un objectif.
Projection	Relation entre un ensemble et un sous-ensemble.
Propriété	Caractéristique d'une classe.
Prototype	Résultat d'une itération ; version partielle d'un système.
Pseudo-état	Désigne des états particuliers comme l'état initial, l'état final, l'historique.
Récursivité	Application d'une règle à ses propres résultats pour générer une séquence infinie de résultats.
Réflexive	Se dit d'une relation dont les rôles concernent la même classe.
Réification	Action de chosifier un concept, une fonction.
Responsabilité	Obligation d'une classe ; partie de sa raison d'être.

Restriction Se dit d'une relation dont la portée a été réduite par une contrainte ou une clé.

Rétro-ingénierie Reconstruction des artefacts des activités en amont, à partir des artefacts des activités en aval.

Réutilisation Usage poursuivi ou répété d'un artefact du développement.

Revue Révision formelle d'une documentation, d'un modèle.

Risque Elément susceptible de perturber le bon déroulement du développement.

Rôle Extrémité d'une relation ; par extension, manière dont les instances d'une classe voient les instances d'une autre classe au travers d'une relation.

Scénario Interaction simple entre objets.

Schème Traduction française de *pattern*.

SEI *Software Engineering Institute*.

Sélecteur 1) Opération qui renseigne sur l'état interne d'un objet, sans le modifier.

2) Dans une expression de navigation, association qui partitionne un ensemble d'objet à partir de la valeur d'une clé.

Serveur Objet qui n'est jamais à l'origine d'une interaction.

Signal Evénement nommé qui peut être déclenché explicitement.

Signature Identifiant non ambigu d'une opération, construit à partir du nom de l'opération et de ses paramètres.

Sous-classe Classe spécialisée, reliée à une autre classe plus générale par une relation de généralisation.

Sous-état Etat englobé par un super-état.

Sous-système Sorte de paquetage de la vue de réalisation ; un sous-système contient des éléments de réalisation ; il est le pendant de la catégorie définie dans la vue logique.

Spécialisation Point de vue descendant porté sur une classification.

Spécification Description exhaustive d'un élément de modélisation.

Stéréotype Extension de la sémantique d'un élément du métamodèle.

Structure Relations statiques entre éléments de modélisation.

Structurée Décrit une technique de décomposition, basée sur la notion de modules et la description des flots de données entre ces modules.

Super-classe Classe générale reliée à une autre classe plus spécialisée par une relation de généralisation.

Super-état Etat contenant des sous-états.

Surcharge	Emploi d'un même nom pour désigner différentes constructions ; la surcharge est résolue statiquement par les compilateurs en fonction du contexte et de la signature des opérations.
Synchrone	Forme de communication bloquante, avec accusé de réception implicite.
Synchronisation	Expression de la forme de communication entre deux objets.
Temps réel	Caractéristique d'un logiciel dont le temps de réponse est compatible avec la dynamique du système.
Test	Ensemble des mesures et des activités qui visent à garantir le fonctionnement satisfaisant du logiciel.
Topologie	Décrit la répartition des modules, ainsi que des données et des opérations dans ces modules, au sein d'une application.
Transition	1) Connexion entre deux états d'un automate, déclenchée par l'occurrence d'un événement. 2) Phase de la vue de l'encadrement dans laquelle le logiciel est transféré à ses utilisateurs.
Transition automatique	Transition déclenchée dès que l'activité en cours dans l'état s'achève.
Typage	Manière dont les langages de programmation supportent la notion de type.
Type	Description d'un ensemble d'instances qui partagent des opérations, des attributs abstraits, des relations et des contraintes.
Type de donnée abstrait	Description d'une donnée en termes opératoires.
Type primitif	Type de base prédéfini par UML.
Use case	Voir cas d'utilisation.
Variable d'instance	Attribut d'objet.
Variable de classe	Attribut qui concerne la classe plus que les objets.
Visibilité	Niveau d'encapsulation des attributs et des opérations dans les classes.
Vision	Définition d'un produit et de sa portée.
Vue	Manière de regarder des éléments de modélisation, éventuellement issus de modèles différents.

Bibliographie

Pour en savoir plus

Bibliographie

C. ALEXANDER, S. ISHIKAWA, M. SILVERSTEIN, M. JACOBSON, I. FISKDAHL-KING, S. ANGEL, *A Pattern Language*, Oxford University Press, 1977.

N.D. BIRREL, M.A. OULD, *A Practical Handbook for Software Development*, Cambridge University Press, 1985.

G. BOOCH, *Object-Oriented Analysis and Design with Applications*, 2e édition, Addison-Wesley, 1994.

BOOCH, *Object Solutions*, Addison-Wesley, 1996. [Traduction française : *Des solutions objets : gérer les projets orientés objets*, ITP France, 1997].

M. BOUZEGHOUB, G. GARDARIN, P. VALDURIEZ, *Les objets*, Eyrolles, 1997.

F.P. BROOKS, *The Mythical Man-Month*, 2e édition, Addison-Wesley, 1995.

F. BUSCHMANN *et al*., *Pattern-Oriented Software Architecture: A System of Patterns*, Wiley, 1996.

P. COAD, E. YOURDON, *Object Oriented Analysis*, Yourdon Press, Prentice Hall, 1991

W.E. DEMING, *Quality, Productivity, and Competitive Position*, Massachusetts Institute of Technology, 1982.

DESFRAY, *Modélisation par objets*, Masson, 1996.

D. EMBLEY, B. D. KURTZ, S.N. WOODFIELD, *Object-Oriented Systems Analysis : A Model-Driven Approach*, Prentice Hall, 1992.

E. GAMMA, R. HELM, R. JOHNSON, J. VLISSIDES, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995. [Traduction française : *Design patterns : catalogue de modèles de conception réutilisables*, ITP France, 1996].

I. GRAHAM, *Object Oriented Methods*, 2e édition, Addison-Wesley, 1994.

W.S. HUMPHREY, *Managing the Software Process*, Addison-Wesley, 1989.

I. JACOBSON, M. CHRISTERSON, P. JONSSON, G. OVERGAARD, *Object-Oriented Software Engineering: A Use Case Driven Approach*, Addison-Wesley, 1992.

I. JACOBSON, M. ERICSON, A. JACOBSON, *The Object Advantage: Business Process Reengineering with Object Technology*, Addison Wesley, 1994.

I. JACOBSON, M. GRISS P. JONSSON, *Software Reuse: Architecture, Process and Organisation for Business Success*, Addison-Wesley, 1997.

J. MARTIN, J.J. ODELL, *Object-Oriented Methods: Pragmatic Considerations*, Prentice Hall, 1996.

B. MEYER, *Object-Oriented Software Construction*, Prentice Hall, 1988. [Traduction française : *Conception et programmation par objets*, InterEdition, 1990].

RUMBAUGH *et al*., *Object Oriented Modeling and Design*, Prentice Hall, 1991. [Traduction française : *OMT – Tome 1 : Modélisation et conception orientées objet*, Masson, 1996].

S. SHLAER, S. J. MELLOR, *Object Oriented Systems Analysis : Modeling the World in Data*, Prentice Hall, 1988.

Adresses utiles sur Internet

Spécifications d'UML 1.0 (Rational Software)
Document en anglais aux formats HTML ou PDF composé de six sections :

• UML Summary : introduction à UML.

• UML Notation Guide : description de la notation unifiée avec exemples d'illustration.

• UML Semantics : description du métamodèle qui est à la base de la sémantique d'UML.

• UML Glossary : glossaire de la terminologie UML.

• UML Process-Specific Extensions : extensions spécifiques pour la représentation des processus de développement.

`http://www.rational.com/uml/`

Rational Rose

Outil de modélisation supportant les notations Booch'93, OMT-2 et UML.

`http://www.rational.com/pst/products/rosefamily.html`

Site web de l'ESSAIM

(Ecole supérieure des sciences appliquées pour l'ingénieur – Mulhouse)

Informations générales sur UML (site de l'auteur).

`http://www.essaim.univ-mulhouse.fr`

Index

Contenu du CD-Rom

Tous les efforts ont été faits pour tester avec soin le CD-Rom et les programmes qu'il contient. Néanmoins, les Editions EYROLLES ne pourront être tenues pour responsables des préjudices ou dommages de quelque nature que ce soit pouvant résulter de l'utilisation de ces programmes.