PAGE D'ACCUEIL DU COURS
*La version finalisée, largement enrichie et corrigée de cette première ébauche de cours est parue, dans la collection Info+ chez les éditions Ellipses, sous le titre* ***UML 2 - de l'apprentissage à la pratique (cours et exercices)*** *(FNAC, amazon.fr)*

# UML 2

Laurent AUDIBERT

## Avant-propos

Les techniques de programmation n'ont cessé de progresser depuis l'époque de la programmation en langage binaire (cartes perforées, switch) à nos jours. Cette évolution a toujours été dictée par le besoin de concevoir et de maintenir des applications toujours plus complexes.

La programmation par cartes perforées, switch ou câblage (de 1800 à 1940) a ainsi fait place à des techniques plus évoluées, comme l'assembleur (1947), avec l'arrivée de l'ordinateur électronique né des besoins de la guerre. Des langages plus évolués ont ensuite vu le jour comme Fortran en 1956 ou Cobol en 1959. Jusque là, les techniques de programmation étaient basées sur le branchement conditionnel et inconditionnel (goto) rendant les programmes importants extrêmement difficiles à développer, à maîtriser et à maintenir.

La programmation structurée (Pascal en 1970, C en 1972, Modula et Ada en 1979, …) a alors vu le jour et permis de développer et de maintenir des applications toujours plus ambitieuses. L'algorithmique ne se suffisant plus à elle seule à la fin des années 1970, le génie logiciel est venu placer la méthodologie au cœur du développement logiciel. Des méthodes comme Merise (1978) se sont alors imposées.

La taille des applications ne cessant de croître, la programmation structurée a également rencontré ses limites, faisant alors place à la programmation orientée objet (Simula 67 en 1967, Smalltalk en 1976, C++ en 1982, Java en 1995, …). La technologie objet est donc la conséquence ultime de la modularisation dictée par la maîtrise de la conception et de la maintenance d'applications toujours plus complexes. Cette nouvelle technique de programmation a nécessité la conception de nouvelles méthodes de modélisation.

UML (*Unified Modeling Language* en anglais, soit langage de modélisation objet unifié) est né de la fusion des trois méthodes qui s'imposaient dans le domaine de la modélisation objet au milieu des années 1990 : OMT, Booch et OOSE. D'importants acteurs industriels (IBM, Microsoft, Oracle, DEC, HP, Rational, Unisys etc.) s'associent alors à l'effort et proposent UML 1.0 à l'OMG (*Object Management Group*) qui l'accepte en novembre 1997 dans sa version 1.1. La version d'UML en cours en 2008 est UML 2.1.1 qui s'impose plus que jamais en tant que langage de modélisation standardisé pour la modélisation des logiciels.

Ce document constitue le support du cours d'UML 2 dispensé aux étudiants du département d'informatique de l'institut universitaire de technologie (IUT) de Villetaneuse en semestre décalé.

Ce support a été réalisé en utilisant les ouvrages cités en bibliographie. Il est en partie basé sur le livre de [7] qui constitue une bonne introduction au langage UML. Aomar Osmani est à l'origine du cours d'UML dans notre IUT.

[27], [5], [3] [26] et [16] ont également été largement utilisés. [27] est un ouvrage de référence assez complet et contient un dictionnaire détaillé de la terminologie UML 2.0. [5], également écrit par les créateurs du langage UML, est un guide d'apprentissage complétant bien le premier ouvrage. [16] est un cours d'UML 2.0 bien expliqué et plus complet et détaillé que [7] mais, en contrepartie, moins accessible. [3] constitue une approche pratique et critique d'UML très intéressante. [25] constitue une excellente approche concrète d'UML comportant des exercices corrigés de très bonne facture que nous reprenons parfois dans les travaux dirigés de ce cours. Pascal Roques est probablement l'un des auteurs les plus prolifique [23, 26, 24, 25] et compétant concernant la mise en œuvre d'UML.

Agréable à lire, [28] s'intéresse à la place de l'informatique dans notre société et plus particulièrement dans l'entreprise.

Enfin, diverses sources trouvées sur Internet, inépuisable source d'information en perpétuel renouvellement, m'ont également été d'un grand secours. Parmi ces dernières, certaines sont incontournables, comme le cours de [21] ou encore le site [9].

## Table des matières

# Chapitre 1 Introduction à la modélisation objet

## 1.1 Le génie logiciel

### 1.1.1 L'informatisation

L'informatisation est le phénomène le plus important de notre époque. Elle s'immisce maintenant dans la pluspart des objets de la vie courante et ce, que ce soit dans l'objet proprement dit [1], ou bien dans le processus de conception ou de fabrication de cet objet.

Actuellement, l'informatique est au cœur de toutes les grandes entreprises. Le système d'information d'une entreprise est composé de matériels et de logiciels. Plus précisément, les investissements dans ce système d'information se réparatissent généralement de la manière suivante : 20% pour le matériel et 80% pour le logiciel. En effet, depuis quelques années, la fabrication du matériel est assurée par quelques fabricants seulement. Ce matériel est relativement fiable et le marché est standardisé. Les problèmes liés à l'informatique sont essentiellement des problèmes de logiciel.

### 1.1.2 Les logiciels

Un logiciel ou une application est un ensemble de programmes, qui permet à un ordinateur ou à un système informatique d'assurer une tâche ou une fonction en particulier (exemple : logiciel de comptabilité, logiciel de gestion des prêts).

Les logiciels, suivant leur taille, peuvent être développés par une personne seule, une petite équipe, ou un ensemble d'équipes coordonnées. Le développement de grands logiciels par de grandes équipes pose d'importants problèmes de conception et de coordination. Or, le développement d'un logiciel est une phase absolument cruciale qui monopolise l'essentiel du coût d'un produit[2] et conditionne sa réussite et sa pérennité.

En 1995, une étude du *Standish Group* dressait un tableau accablant de la conduite des projets informatiques. Reposant sur un échantillon représentatif de 365 entreprises, totalisant 8 380 applications, cette étude établissait que :

- 16,2% seulement des projets étaient conformes aux prévisions initiales,
- 52,7% avaient subi des dépassements en coût et délai d'un facteur 2 à 3 avec diminution du nombre des fonctions offertes,
- 31,1% ont été purement abandonnés durant leur développement.

Pour les grandes entreprises (qui lancent proportionnellement davantage de gros projets), le taux de succès est de 9% seulement, 37% des projets sont arrêtés en cours de réalisation, 50% aboutissent hors délai et hors budget.

L'examen des causes de succès et d'échec est instructif : la plupart des échecs proviennent non de l'informatique, mais de la maîtrise d'ouvrage[3] (*i.e.* le client). Pour ces raisons, le développement de logiciels dans un contexte professionnel suit souvent des règles strictes encadrant la conception et permettant le travail en groupe et la maintenance[4] du code. Ainsi, une nouvelle discipline est née : le génie logiciel.

### 1.1.3 Le génie logiciel

Le génie logiciel est un domaine de recherche qui a été défini (fait rare) du 7 au 11 octobre 1968, à Garmisch-Partenkirchen, sous le parrainage de l'OTAN. Il a pour objectif de répondre à un problème qui s'énonçait en deux constatations : d'une part le logiciel n'était pas fiable, d'autre part, il était incroyablement difficile de réaliser dans des délais prévus des logiciels satisfaisant leur cahier des charges.

L'objectif du génie logiciel est d'optimiser le coût de développement du logiciel. L'importance d'une approche méthodologique s'est imposée à la suite de la crise de l'industrie du logiciel à la fin des années 1970. Cette crise de l'industrie du logiciel était principalement due à :

- l'augmentation des coûts ;
- les difficultés de maintenance et d'évolution ;
- la non fiabilité ;
- le non respect des spécifications ;
- le non respect des délais.

La maintenance est devenue une facette très importante du cycle de vie d'un logiciel. En effet, une enquête effectuée aux USA en 1986 auprès de 55 entreprises révèle que 53% du budget total d'un logiciel est affecté à la maintenance. Ce coût est réparti comme suit :

- 34% maintenance évolutive (modification des spécifications initiales) ;
- 10% maintenance adaptative (nouvel environnement, nouveaux utilisateurs) ;
- 17% maintenance corrective (correction des bogues) ;
- 16% maintenance perfective (améliorer les performance sans changer les spécifications) ;
- 6% assistance aux utilisateurs ;
- 6% contrôle qualité ;
- 7% organisation/suivi ;
- 4% divers.

Pour apporter une réponse à tous ces problèmes, le génie logiciel s'intéresse particulièrement à la manière dont le code source d'un logiciel est spécifié puis produit. Ainsi, le génie logiciel touche au cycle de vie des logiciels :

- l'analyse du besoin,
- l'élaboration des spécifications,
- la conceptualisation,
- le développement,
- la phase de test,
- la maintenance.

Les projets relatifs à l'ingénierie logicielle sont généralement de grande envergure et dépassent souvent les 10000 lignes de code. C'est pourquoi ces projets nécessitent une équipe de développement bien structurée. La gestion de projet se retrouve naturellement intimement liée au génie logiciel.

### 1.1.4 Notion de qualité pour un logiciel

En génie logiciel, divers travaux ont mené à la définition de la qualité du logiciel en termes de facteurs, qui dépendent, entre autres, du domaine de l'application et des outils utilisés. Parmi ces derniers nous pouvons citer :

**Validité :**
aptitude d'un produit logiciel à remplir exactement ses fonctions, définies par le cahier des charges et les spécifications.

**Fiabilité ou robustesse :**
aptitude d'un produit logiciel à fonctionner dans des conditions anormales.

**Extensibilité (maintenance) :**
facilité avec laquelle un logiciel se prête à sa maintenance, c'est-à-dire à une modification ou à une extension des fonctions qui lui sont demandées.

**Réutilisabilité :**
aptitude d'un logiciel à être réutilisé, en tout ou en partie, dans de nouvelles applications.

**Compatibilité :**
facilité avec laquelle un logiciel peut être combiné avec d'autres logiciels.

**Efficacité :**
Utilisation optimales des ressources matérielles.

**Portabilité :**
facilité avec laquelle un logiciel peut être transféré sous différents environnements matériels et logiciels.

**Vérifiabilité :**
facilité de préparation des procédures de test.

**Intégrité :**
aptitude d'un logiciel à protéger son code et ses données contre des accès non autorisés.

**Facilité d'emploi :**
facilité d'apprentissage, d'utilisation, de préparation des données, d'interprétation des erreurs et de rattrapage en cas d'erreur d'utilisation.

Ces facteurs sont parfois contradictoires, le choix des compromis doit s'effectuer en fonction du contexte.

---

[1] Par exemple, aujourd'hui, 90% des nouvelles fonctionnalités des automobiles sont apportées par l'électronique et l'informatique embarquées. Il y a, ou aura à terme, du

logiciel partout : ampoules électriques, four à micro ondes, tissus des vêtements, stylos, livres, etc.

2

Comparativement à sa production, le coût du développement d'un logiciel est extrêmement important. Nous verrons par la suite que la maintenance coûte également très cher.

3

c.f. section 1.2.1 << Maîtrise d'ouvrage et maîtrise d'œuvre >> pour une définition de ce terme.

4

Souvent, les personnes qui doivent opérer des modifications ultérieures dans le code ne sont plus les personnes qui l'ont développé.

## 1.2 Modélisation, cycles de vie et méthodes

### 1.2.1 Pourquoi et comment modéliser ?

#### Qu'est-ce qu'un modèle ?

Un *modèle* est une représentation abstraite et simplifiée (*i.e.* qui exclut certains détails), d'une entité (phénomène, processus, système, etc.) du monde réel en vue de le décrire, de l'expliquer ou de le prévoir. Modèle est synonyme de théorie, mais avec une connotation pratique : un modèle, c'est une théorie orientée vers l'action qu'elle doit servir.

Concrètement, un modèle permet de réduire la complexité d'un phénomène en éliminant les détails qui n'influencent pas son comportement de manière significative. Il reflète ce que le concepteur croit important pour la compréhension et la prédiction du phénomène modélisé. Les limites du phénomène modélisé dépendant des objectifs du modèle.

Voici quelques exemples de modèles :

**Modèle météorologique –**
à partir de données d'observation (satellite, …), il permet de prévoir les conditions climatiques pour les jours à venir.

**Modèle économique –**
peut par exemple permettre de simuler l'évolution de cours boursiers en fonction d'hypothèses macro-économiques (évolution du chômage, taux de croissance, …).

**Modèle démographique –**
définit la composition d'un panel d'une population et son comportement, dans le but de fiabiliser des études statistiques, d'augmenter l'impact de démarches commerciales, etc.

**Plans –**
Les plans sont des modèles qui donnent une vue d'ensemble du système concerné. Par exemple, dans le bâtiment, pour la construction d'un immeuble, il faut préalablement élaborer de nombreux plans :

- plans d'implantation du bâtiment dans son environnement ;
- plans généraux du bâtiment et de sa structure ;
- plans détaillées des différents locaux, bureaux, appartements, …
- plans des câblages électriques ;
- plans d'écoulements des eaux, etc.

Les trois premiers exemples sont des modèles que l'on qualifie de prédictifs. Le dernier, plus conceptuel, possède différents niveaux de vues comme la pluspart des modèles en génie logiciel.

#### Pourquoi modéliser ?

Modéliser un système avant sa réalisation permet de mieux comprendre le fonctionnement du système. C'est également un bon moyen de maîtriser sa complexité et d'assurer sa cohérence. Un modèle est un langage commun, précis, qui est connu par tous les membres de l'équipe et il est donc, à ce titre, un vecteur privilégié pour communiquer. Cette communication est essentielle pour aboutir à une compréhension commune aux différentes parties prenantes (notamment entre la maîtrise d'ouvrage et la maîtrise d'œuvre informatique) et précise d'un problème donné.

Dans le domaine de l'ingénierie du logiciel, le modèle permet de mieux répartir les tâches et d'automatiser certaines d'entre elles. C'est également un facteur de réduction des coûts et des délais. Par exemple, les plateformes de modélisation savent maintenant exploiter les modèles pour faire de la génération de code (au moins au niveau du squelette) voire des aller-retours entre le code et le modèle sans perte d'information. Le modèle est enfin indispensable pour assurer un bon niveau de qualité et une maintenance efficace. En effet, une fois mise en production, l'application va devoir être maintenue, probablement par une autre équipe et, qui plus est, pas nécessairement de la même société que celle ayant créée l'application.

Le choix du modèle a donc une influence capitale sur les solutions obtenues. Les systèmes non-triviaux sont mieux modélisés par un ensemble de modèles indépendants. Selon les modèles employés, la démarche de modélisation n'est pas la même.

#### Qui doit modéliser ?

La modélisation est souvent faite par la maîtrise d'œuvre informatique (MOE). C'est malencontreux, car les priorités de la MOE résident dans le fonctionnement de la plate-forme informatique et non dans les processus de l'entreprise.

Il est préférable que la modélisation soit réalisée par la maîtrise d'ouvrage (MOA) de sorte que le métier soit maître de ses propres concepts. La MOE doit intervenir dans le modèle lorsque, après avoir défini les concepts du métier, on doit introduire les contraintes propres à la plate-forme informatique.

Il est vrai que certains métiers, dont les priorités sont opérationnelles, ne disposent pas toujours de la capacité d'abstraction et de la rigueur conceptuelle nécessaires à la formalisation. La professionnalisation de la MOA a pour but de les doter de ces compétences. Cette professionnalisation réside essentiellement dans l'aptitude à modéliser le système d'information du métier : le maître mot est *modélisation*. Lorsque le modèle du système d'information est de bonne qualité, sobre, clair, stable, la maîtrise d'œuvre peut travailler dans de bonnes conditions. Lorsque cette professionnalisation a lieu, elle modifie les rapports avec l'informatique et déplace la frontière des responsabilités, ce qui contrarie parfois les informaticiens dans un premier temps, avant qu'ils n'en voient apparaître les bénéfices.

#### Maîtrise d'ouvrage et maîtrise d'œuvre

**Maître d'ouvrage (MOA) :**

Le MOA est une personne morale (entreprise, direction etc.), une entité de l'organisation. Ce n'est jamais une personne.

**Maître d'œuvre (MOE) :**

Le MOE est une personne morale (entreprise, direction etc.) garante de la bonne réalisation technique des solutions. Il a, lors de la conception du SI, un devoir de conseil vis-à-vis du MOA, car le SI doit tirer le meilleur parti des possibilités techniques.

Le MOA est client du MOE à qui il passe commande d'un produit nécessaire à son activité.

Le MOE fournit ce produit ; soit il le réalise lui-même, soit il passe commande à un ou plusieurs fournisseurs (« entreprises ») qui élaborent le produit sous sa direction.

La relation MOA et MOE est définie par un contrat qui précise leurs engagements mutuels.

Lorsque le produit est compliqué, il peut être nécessaire de faire appel à plusieurs fournisseurs. Le MOE assure leur coordination ; il veille à la cohérence des fournitures et à leur compatibilité. Il coordonne l'action des fournisseurs en contrôlant la qualité technique, en assurant le respect des délais fixés par le MOA et en minimisant les risques.

Le MOE est responsable de la qualité technique de la solution. Il doit, avant toute livraison au MOA, procéder aux vérifications nécessaires (« recette usine »).

### 1.2.2 Le cycle de vie d'un logiciel

Le *cycle de vie d'un logiciel* (en anglais *software lifecycle*), désigne toutes les étapes du développement d'un logiciel, de sa conception à sa disparition. L'objectif d'un tel découpage est de permettre de définir des jalons intermédiaires permettant la validation du développement logiciel, c'est-à-dire la conformité du logiciel avec les besoins exprimés, et la vérification du processus de développement, c'est-à-dire l'adéquation des méthodes mises en œuvre.

L'origine de ce découpage provient du constat que les erreurs ont un coût d'autant plus élevé qu'elles sont détectées tardivement dans le processus de réalisation. Le cycle de vie permet de détecter les erreurs au plus tôt et ainsi de maîtriser la qualité du logiciel, les délais de sa réalisation et les coûts associés.

Le cycle de vie du logiciel comprend généralement au minimum les étapes suivantes :

**Définition des objectifs –**

Cet étape consiste à définir la finalité du projet et son inscription dans une stratégie globale.

**Analyse des besoins et faisabilité –**

c'est-à-dire l'expression, le recueil et la formalisation des besoins du demandeur (le client) et de l'ensemble des contraintes, puis l'estimation de la faisabilité de ces besoins.

**Spécifications ou conception générale –**

Il s'agit de l'élaboration des spécifications de l'architecture générale du logiciel.

**Conception détaillée –**

Cette étape consiste à définir précisément chaque sous-ensemble du logiciel.

**Codage (Implémentation ou programmation) –**

c'est la traduction dans un langage de programmation des fonctionnalités définies lors de phases de conception.

**Tests unitaires –**

Ils permettent de vérifier individuellement que chaque sous-ensemble du logiciel est implémenté conformément aux spécifications.

**Intégration –**

L'objectif est de s'assurer de l'interfaçage des différents éléments (modules) du logiciel. Elle fait l'objet de tests d'intégration consignés dans un document.

**Qualification (ou recette) –**

C'est-à-dire la vérification de la conformité du logiciel aux spécifications initiales.

**Documentation –**

Elle vise à produire les informations nécessaires pour l'utilisation du logiciel et pour des développements ultérieurs.

**Mise en production –**

C'est le déploiement sur site du logiciel.

**Maintenance –**

Elle comprend toutes les actions correctives (maintenance corrective) et évolutives (maintenance évolutive) sur le logiciel.

La séquence et la présence de chacune de ces activités dans le cycle de vie dépend du choix d'un modèle de cycle de vie entre le client et l'équipe de développement. Le cycle de vie permet de prendre en compte, en plus des aspects techniques, l'organisation et les aspects humains.

### 1.2.3 Modèles de cycles de vie d'un logiciel

**Modèle de cycle de vie en cascade**

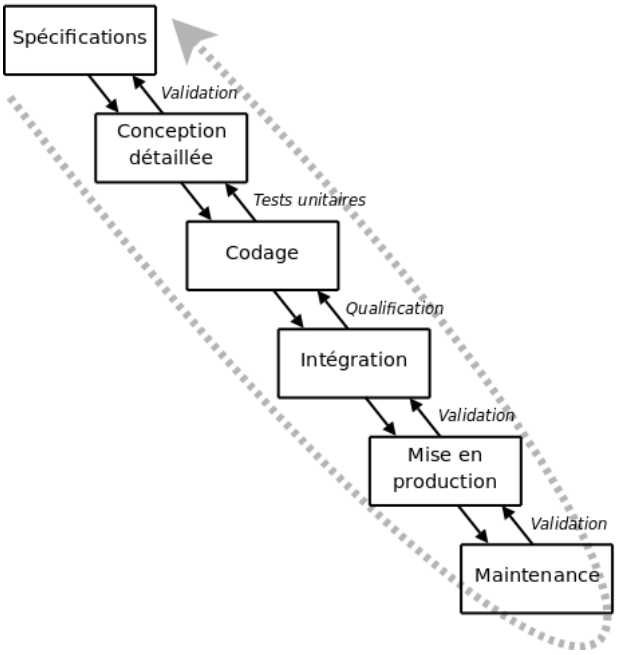



Figure 1.1: Modèle du cycle de vie en cascade

Le modèle de cycle de vie en cascade (cf. figure 1.1) a été mis au point dès 1966, puis formalisé aux alentours de 1970.

Dans ce modèle le principe est très simple : chaque phase se termine à une date précise par la production de certains documents ou logiciels. Les résultats sont définis sur la base des interactions entre étapes, ils sont soumis à une revue approfondie et on ne passe à la phase suivante que s'ils sont jugés satisfaisants.

Le modèle original ne comportait pas de possibilité de retour en arrière. Celle-ci a été rajoutée ultérieurement sur la base qu'une étape ne remet en cause que l'étape précédente, ce qui, dans la pratique, s'avère insuffisant.

L'inconvénient majeur du modèle de cycle de vie en cascade est que la vérification du bon fonctionnement du système est réalisée trop tardivement: lors de la phase d'intégration, ou pire, lors de la mise en production.

**Modèle de cycle de vie en V**

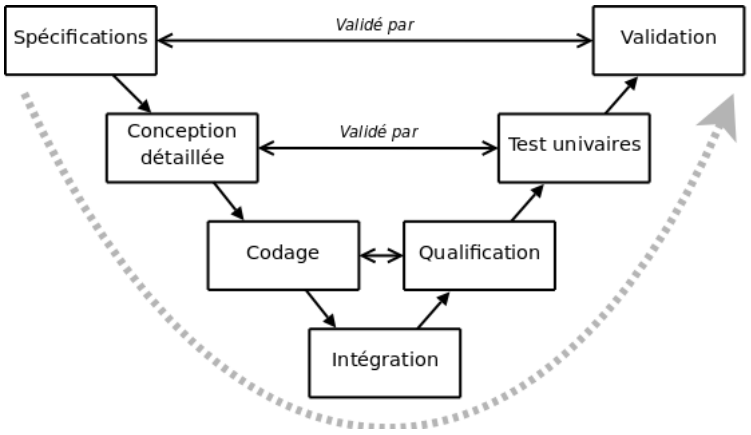



Figure 1.2: Modèle du cycle de vie en V

Le modèle en V (cf. figure 1.2) demeure actuellement le cycle de vie le plus connu et certainement le plus utilisé. Il s'agit d'un modèle en cascade dans lequel le développement des tests et du logiciels sont effectués de manière synchrone.

Le principe de ce modèle est qu'avec toute décomposition doit être décrite la recomposition et que toute description d'un composant est accompagnée de tests qui permettront de s'assurer qu'il correspond à sa description.

Ceci rend explicite la préparation des dernières phases (validation-vérification) par les premières (construction du logiciel), et permet ainsi d'éviter un écueil bien connu de la spécification du logiciel : énoncer une propriété qu'il est impossible de vérifier objectivement après la réalisation.

Cependant, ce modèle souffre toujours du problème de la vérification trop tardive du bon fonctionnement du système.

**Modèle de cycle de vie en spirale**

Proposé par B. Boehm en 1988, ce modèle est beaucoup plus général que le précédent. Il met l'accent sur l'activité d'analyse des risques : chaque cycle de la spirale se déroule en quatre phases :

1. détermination, à partir des résultats des cycles précédents, ou de l'analyse préliminaire des besoins, des objectifs du cycle, des alternatives pour les atteindre et des contraintes ;
2. analyse des risques, évaluation des alternatives et, éventuellement maquettage ;

3. développement et vérification de la solution retenue, un modèle « classique » (cascade ou en V) peut être utilisé ici ;
4. revue des résultats et vérification du cycle suivant.

L'analyse préliminaire est affinée au cours des premiers cycles. Le modèle utilise des maquettes exploratoires pour guider la phase de conception du cycle suivant. Le dernier cycle se termine par un processus de développement classique.

**Modèle par incrément**

Dans les modèles précédents un logiciel est décomposé en composants développés séparément et intégrés à la fin du processus.

Dans les modèles par incrément un seul ensemble de composants est développé à la fois : des incréments viennent s'intégrer à un noyau de logiciel développé au préalable. Chaque incrément est développé selon l'un des modèles précédents.

Les avantages de ce type de modèle sont les suivants :

- chaque développement est moins complexe ;
- les intégrations sont progressives ;
- il est ainsi possible de livrer et de mettre en service chaque incrément ;
- il permet un meilleur lissage du temps et de l'effort de développement grâce à la possibilité de recouvrement (parallélisation) des différentes phases.

Les risques de ce type de modèle sont les suivants :

- remettre en cause les incréments précédents ou pire le noyau ;
- ne pas pouvoir intégrer de nouveaux incréments.

Les noyaux, les incréments ainsi que leurs interactions doivent donc être spécifiés globalement, au début du projet. Les incréments doivent être aussi indépendants que possibles, fonctionnellement mais aussi sur le plan du calendrier du développement.

### 1.2.4 Méthodes d'analyse et de conception

Les méthodes d'analyse et de conception fournissent une méthodologie et des notations standards qui aident à concevoir des logiciels de qualité. Il existe différentes manières pour classer ces méthodes, dont :

**La distinction entre composition et décomposition :**
Elle met en opposition d'une part les méthodes ascendantes qui consistent à construire un logiciel par composition à partir de modules existants et, d'autre part, les méthodes descendantes qui décomposent récursivement le système jusqu'à arriver à des modules programmables simplement.

**La distinction entre fonctionnel (dirigée par le traitement) et orientée objet :**
Dans la stratégie fonctionnelle (également qualifiée de structurée) un système est vu comme un ensemble hiérarchique d'unités en interaction, ayant chacune une fonction clairement définie. Les fonctions disposent d'un état local, mais le système a un état partagé, qui est centralisé et accessible par l'ensemble des fonctions. Les stratégies orientées objet considèrent qu'un système est un ensemble d'objets interagissants. Chaque objet dispose d'un ensemble d'attributs décrivant son état et l'état du système est décrit (de façon décentralisée) par l'état de l'ensemble.

## 1.3 De la programmation structurée à l'approche orientée objet

### 1.3.1 Méthodes fonctionnelles ou structurées

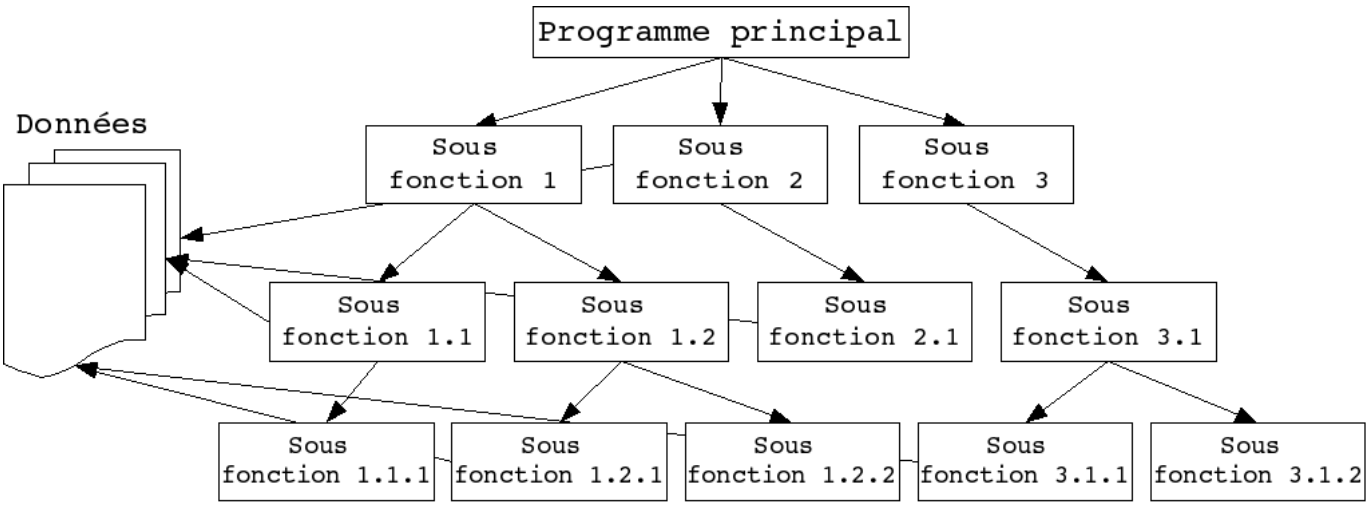



Figure 1.3: Représentation graphique d'une approche fonctionnelle

Les méthodes fonctionnelles (également qualifiées de méthodes structurées) trouvent leur origine dans les langages procéduraux. Elles mettent en évidence les fonctions à assurer et proposent une approche hiérarchique descendante et modulaire.

Ces méthodes utilisent intensivement les raffinements successifs pour produire des spécifications dont l'essentielle est sous forme de notation graphique en diagrammes de flots de données. Le plus haut niveau représente l'ensemble du problème (sous forme d'activité, de données ou de processus, selon la méthode). Chaque niveau est ensuite décomposé en respectant les entrées/sorties du niveau supérieur. La décomposition se poursuit jusqu'à arriver à des composants maîtrisables (cf. figure 1.3).

L'approche fonctionnelle dissocie le problème de la représentation des données, du problème du traitement de ces données. Sur la figure 1.3, les données du problème sont représentées sur la gauche. Des flèches transversalles matérialisent la manipulation de ces données par des sous-fonctions. Cet accès peut-être direct (c'est parfois le cas quand

les données sont regroupées dans une base de données), ou peut être réalisé par le passage de paramètre depuis le programme principal.

La SADT (*Structured Analysis Design Technique*) est probablement la méthode d'analyse fonctionnelle et de gestion de projets la plus connue. Elle permet non seulement de décrire les tâches du projet et leurs interactions, mais aussi de décrire le système que le projet vise à étudier, créer ou modifier, en mettant notamment en évidence les parties qui constituent le système, la finalité et le fonctionnement de chacune, ainsi que les interfaces entre ces diverses parties. Le système ainsi modélisé n'est pas une simple collection d'éléments indépendants, mais une organisation structurée de ceux-ci dans une finalité précise.

**En résumé, l'architecture du système est dictée par la réponse au problème (*i.e.* la fonction du système).**

### 1.3.2 L'approche orientée objet

L'approche orientée objet considère le logiciel comme une collection d'objets dissociés, identifiés et possédant des caractéristiques. Une caractéristique est soit un attribut (*i.e.* une donnée caractérisant l'état de l'objet), soit une entité comportementale de l'objet (*i.e.* une fonction). La fonctionnalité du logiciel émerge alors de l'interaction entre les différents objets qui le constituent. L'une des particularités de cette approche est qu'elle rapproche les données et leurs traitements associés au sein d'un unique objet.

Comme nous venons de le dire, un objet est caractérisé par plusieurs notions :

**L'identité –**
L'objet possède une identité, qui permet de le distinguer des autres objets, indépendamment de son état. On construit généralement cette identité grâce à un identifiant découlant naturellement du problème (par exemple un produit pourra être repéré par un code, une voiture par un numéro de série, etc.)

**Les attributs –**
Il s'agit des données caractérisant l'objet. Ce sont des variables stockant des informations sur l'état de l'objet.

**Les méthodes –**
Les méthodes d'un objet caractérisent son comportement, c'est-à-dire l'ensemble des actions (appelées opérations) que l'objet est à même de réaliser. Ces opérations permettent de faire réagir l'objet aux sollicitations extérieures (ou d'agir sur les autres objets). De plus, les opérations sont étroitement liées aux attributs, car leurs actions peuvent dépendre des valeurs des attributs, ou bien les modifier.

La difficulté de cette modélisation consiste à créer une représentation abstraite, sous forme d'objets, d'entités ayant une existence matérielle (chien, voiture, ampoule, personne, …) ou bien virtuelle (client, temps, …).

La Conception Orientée Objet (COO) est la méthode qui conduit à des architectures logicielles fondées sur les objets du système, plutôt que sur la fonction qu'il est censé réaliser.

**En résumé, l'architecture du système est dictée par la structure du problème.**

### 1.3.3 Approche fonctionnelle vs. approche objet

Selon la thèse de Church-Turing, tout langage de programmation non trivial équivaut à une machine de Turing. Il en résulte que tout programme qu'il est possible d'écrire dans un langage pourrait également être écrit dans n'importe quel autre langage. Ainsi, tout ce que l'on fait avec un langage de programmation par objets pourrait être fait en programmation impérative. La différence entre une approche fonctionnelle et une approche objet n'est donc pas d'ordre logique, mais pratique.

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

En approche objet, l'évolution des besoins aura le plus souvent tendance à se présenter comme un changement de l'interaction des objets. S'il faut apporter une modification aux données, seul l'objet incriminé (encapsulant cette donnée) sera modifié. Toutes les fonctions à modifier sont bien identifiées : elles se trouvent dans ce même objet : ce sont ses méthodes. Dans une approche structurée, l'évolution des besoins entraîne souvent une dégénérescence, ou une profonde remise en question, de la topologie typique de la figure 1.3 car la décomposition des unités de traitement (du programme principal aux sous-fonctions) est directement dictée par ces besoins. D'autre part, une modification des données entraîne généralement une modification d'un nombre important de fonctions éparpillées et difficiles à identifier dans la hiérarchie de cette décomposition.

En fait, la modularité n'est pas antinomique de l'approche structurée. Les modules résultant de la décomposition objet sont tout simplement différents de ceux émanant de l'approche structurée. Les unités de traitement, et surtout leur dépendance dans la topologie de la figure 1.3 sont initialement bons. C'est leur résistance au temps, contrairement aux modules objet, qui est source de problème. La structure d'un logiciel issue d'une approche structurée est beaucoup moins malléable, adaptable, que celle issue d'une approche objet.

Ainsi la technologie objet est la conséquence ultime de la modularisation du logiciel, démarche qui vise à maîtriser sa production et son évolution. Mais malgré cette continuité logique les langages objet ont apporté en pratique un profond changement dans l'art de la programmation : ils impliquent en effet un changement de l'attitude mentale du programmeur.

### 1.3.4 Concepts importants de l'approche objet

Dans la section 1.3.2, nous avons dit que l'approche objet rapproche les données et leurs traitements. Mais cette approche ne fait pas que ça, d'autres concepts importants sont spécifiques à cette approche et participent à la qualité du logiciel.

#### Notion de classe

Tout d'abord, introduisons la notion de classe. Une classe est un type de données abstrait qui précise des caractéristiques (attributs et méthodes) communes à toute une famille d'objets et qui permet de créer (instancier) des objets possédant ces caractéristiques. Les autres concepts importants qu'il nous faut maintenant introduire sont l'encapsulation, l'héritage et l'agrégation.

#### Encapsulation

L'encapsulation consiste à masquer les détails d'implémentation d'un objet, en définissant une interface. L'interface est la vue externe d'un objet, elle définit les services accessibles (offerts) aux utilisateurs de l'objet.

L'encapsulation facilite l'évolution d'une application car elle stabilise l'utilisation des objets : on peut modifier l'implémentation des attributs d'un objet sans modifier son

interface, et donc la façon dont l'objet est utilisé.

L'encapsulation garantit l'intégrité des données, car elle permet d'interdire, ou de restreindre, l'accès direct aux attributs des objets.

**Héritage, Spécialisation, Généralisation et Polymorphisme**

L'héritage est un mécanisme de transmission des caractéristiques d'une classe (ses attributs et méthodes) vers une sous-classe. Une classe peut être spécialisée en d'autres classes, afin d'y ajouter des caractéristiques spécifiques ou d'en adapter certaines. Plusieurs classes peuvent être généralisées en une classe qui les factorise, afin de regrouper les caractéristiques communes d'un ensemble de classes.

Ainsi, la spécialisation et la généralisation permettent de construire des hiérarchies de classes. L'héritage peut être simple ou multiple. L'héritage évite la duplication et encourage la réutilisation.

Le polymorphisme représente la faculté d'une méthode à pouvoir s'appliquer à des objets de classes différentes. Le polymorphisme augmente la généricité, et donc la qualité, du code.

**Agrégation**

Il s'agit d'une relation entre deux classes, spécifiant que les objets d'une classe sont des composants de l'autre classe. Une relation d'agrégation permet donc de définir des objets composés d'autres objets. L'agrégation permet donc d'assembler des objets de base, afin de construire des objets plus complexes.

### 1.3.5 Historique la programmation par objets

Les premiers langages de programmation qui ont utilisé des objets sont Simula I (1961-64) et Simula 67 (1967), conçus par les informaticiens norvégiens Ole-Johan Dahl et Kristan Nygaard. Simula 67 contenait déjà les objets, les classes, l'héritage, l'encapsulation, etc.

Alan Kay, du PARC de Xerox, avait utilisé Simula dans les années 1960. Il réalisa en 1976 Smalltalk qui reste, aux yeux de certains programmeurs, le meilleur langage de programmation par objets.

Bjarne Stroustrup a mis au point C++, une extension du langage C permettant la programmation orientée objets, aux Bell Labs d'AT&T en 1982. C++ deviendra le langage le plus utilisé par les programmeurs professionnels. Il arrivera à maturation en 1986, sa standardisation ANSI / ISO date de 1997.

Java est lancé par Sun en 1995. Comme il présente plus de sécurité que C++, il deviendra le langage favori de certains programmeurs professionnels.

## 1.4 UML

### 1.4.1 Introduction

La description de la programmation par objets a fait ressortir l'étendue du travail conceptuel nécessaire : définition des classes, de leurs relations, des attributs et méthodes, des interfaces etc.

Pour programmer une application, il ne convient pas de se lancer tête baissée dans l'écriture du code : il faut d'abord organiser ses idées, les documenter, puis organiser la réalisation en définissant les modules et étapes de la réalisation. C'est cette démarche antérieure à l'écriture que l'on appelle *modélisation* ; son produit est un *modèle*.

Les spécifications fournies par la maîtrise d'ouvrage en programmation impérative étaient souvent floues : les articulations conceptuelles (structures de données, algorithmes de traitement) s'exprimant dans le vocabulaire de l'informatique, le modèle devait souvent être élaboré par celle-ci. L'approche objet permet en principe à la maîtrise d'ouvrage de s'exprimer de façon précise selon un vocabulaire qui, tout en transcrivant les besoins du métier, pourra être immédiatement compris par les informaticiens. En principe seulement, car la modélisation demande aux maîtrises d'ouvrage une compétence et un professionnalisme qui ne sont pas aujourd'hui répandus.

### 1.4.2 Histoire des modélisations par objets

Les méthodes utilisées dans les années 1980 pour organiser la programmation impérative (notamment Merise) étaient fondées sur la modélisation séparée des données et des traitements. Lorsque la programmation par objets prend de l'importance au début des années 1990, la nécessité d'une méthode qui lui soit adaptée devient évidente. Plus de cinquante méthodes apparaissent entre 1990 et 1995 (Booch, Classe-Relation, Fusion, HOOD, OMT, OOA, OOD, OOM, OOSE, etc.) mais aucune ne parvient à s'imposer. En 1994, le consensus se fait autour de trois méthodes :

- OMT de James Rumbaugh (*General Electric*) fournit une représentation graphique des aspects statique, dynamique et fonctionnel d'un système ;
- OOD de Grady Booch, définie pour le *Department of Defense*, introduit le concept de paquetage (*package*) ;
- OOSE d'Ivar Jacobson (Ericsson) fonde l'analyse sur la description des besoins des utilisateurs (cas d'utilisation, ou *use cases*).

Chaque méthode avait ses avantages et ses partisans. Le nombre de méthodes en compétition s'était réduit, mais le risque d'un éclatement subsistait : la profession pouvait se diviser entre ces trois méthodes, créant autant de continents intellectuels qui auraient du mal à communiquer.

Événement considérable et presque miraculeux, les trois gourous qui régnaient chacun sur l'une des trois méthodes se mirent d'accord pour définir une méthode commune qui fédérerait leurs apports respectifs (on les surnomme depuis « the Amigos »). UML (*Unified Modeling Language*) est né de cet effort de convergence. L'adjectif *unified* est là pour marquer qu'UML unifie, et donc remplace.

En fait, et comme son nom l'indique, UML n'a pas l'ambition d'être exactement une méthode : c'est un langage.

L'unification a progressé par étapes. En 1995, Booch et Rumbaugh (et quelques autres) se sont mis d'accord pour construire une méthode unifiée, *Unified Method 0.8* ; en 1996, Jacobson les a rejoints pour produire UML 0.9 (notez le remplacement du mot *méthode* par le mot *langage*, plus modeste). Les acteurs les plus importants dans le monde du logiciel s'associent alors à l'effort (IBM, Microsoft, Oracle, DEC, HP, Rational, Unisys etc.) et UML 1.0 est soumis à l'OMG[5]. L'OMG adopte en novembre 1997 UML 1.1 comme langage de modélisation des systèmes d'information à objets. La version d'UML en cours en 2008 est UML 2.1.1 et les travaux d'amélioration se poursuivent.

UML est donc non seulement un outil intéressant mais une norme qui s'impose en technologie à objets et à laquelle se sont rangés tous les grands acteurs du domaine, acteurs qui ont d'ailleurs contribué à son élaboration.

### 1.4.3 UML en œuvre

UML n'est pas une méthode (*i.e.* une description normative des étapes de la modélisation) : ses auteurs ont en effet estimé qu'il n'était pas opportun de définir une méthode en raison de la diversité des cas particuliers. Ils ont préféré se borner à définir un langage graphique qui permet de représenter et de communiquer les divers aspects d'un système d'information. Aux graphiques sont bien sûr associés des textes qui expliquent leur contenu. UML est donc un métalangage car il fournit les éléments permettant de construire le modèle qui, lui, sera le langage du projet.

Il est impossible de donner une représentation graphique complète d'un logiciel, ou de tout autre système complexe, de même qu'il est impossible de représenter entièrement une statue (à trois dimensions) par des photographies (à deux dimensions). Mais il est possible de donner sur un tel système des *vues* partielles, analogues chacune à une photographie d'une statue, et dont la conjonction donnera une idée utilisable en pratique sans risque d'erreur grave.

UML 2.0 comporte ainsi treize types de diagrammes représentant autant de *vues* distinctes pour représenter des concepts particuliers du système d'information. Ils se répartissent en deux grands groupes :

**Diagrammes structurels ou diagrammes statiques (*UML Structure*)**

- diagramme de classes (*Class diagram*)
- diagramme d'objets (*Object diagram*)
- diagramme de composants (*Component diagram*)
- diagramme de déploiement (*Deployment diagram*)
- diagramme de paquetages (*Package diagram*)
- diagramme de structures composites (*Composite structure diagram*)

**Diagrammes comportementaux ou diagrammes dynamiques (*UML Behavior*)**

- diagramme de cas d'utilisation (*Use case diagram*)
- diagramme d'activités (*Activity diagram*)
- diagramme d'états-transitions (*State machine diagram*)
- **Diagrammes d'interaction (*Interaction diagram*)**
  - diagramme de séquence (*Sequence diagram*)
  - diagramme de communication (*Communication diagram*)
  - diagramme global d'interaction (*Interaction overview diagram*)
  - diagramme de temps (*Timing diagram*)

Ces diagrammes, d'une utilité variable selon les cas, ne sont pas nécessairement tous produits à l'occasion d'une modélisation. Les plus utiles pour la maîtrise d'ouvrage sont les diagrammes d'activités, de cas d'utilisation, de classes, d'objets, de séquence et d'états-transitions. Les diagrammes de composants, de déploiement et de communication sont surtout utiles pour la maîtrise d'œuvre à qui ils permettent de formaliser les contraintes de la réalisation et la solution technique.

**Diagramme de cas d'utilisation**

Le diagramme de cas d'utilisation (cf. section 2) représente la structure des grandes fonctionnalités nécessaires aux utilisateurs du système. C'est le premier diagramme du modèle UML, celui où s'assure la relation entre l'utilisateur et les objets que le système met en œuvre.

**Diagramme de classes**

Le diagramme de classes (cf. section 3) est généralement considéré comme le plus important dans un développement orienté objet. Il représente l'architecture conceptuelle du système : il décrit les classes que le système utilise, ainsi que leurs liens, que ceux-ci représentent un emboîtage conceptuel (héritage) ou une relation organique (agrégation).

**Diagramme d'objets**

Le diagramme d'objets (cf. section 3.5) permet d'éclairer un diagramme de classes en l'illustrant par des exemples. Il est, par exemple, utilisé pour vérifier l'adéquation d'un diagramme de classes à différents cas possibles.

**Diagramme d'états-transitions**

Le diagramme d'états-transitions (cf. section 5) représente la façon dont évoluent (*i.e.* cycle de vie) les objets appartenant à une même classe. La modélisation du cycle de vie est essentielle pour représenter et mettre en forme la dynamique du système.

**Diagramme d'activités**

Le diagramme d'activités (cf. section 6) n'est autre que la transcription dans UML de la représentation du processus telle qu'elle a été élaborée lors du travail qui a préparé la modélisation : il montre l'enchaînement des activités qui concourent au processus.

**Diagramme de séquence et de communication**

Le diagramme de séquence (cf. section 7.3) représente la succession chronologique des opérations réalisées par un acteur. Il indique les objets que l'acteur va manipuler et les opérations qui font passer d'un objet à l'autre. On peut représenter les mêmes opérations par un diagramme de communication (cf. section 7.2), graphe dont les nœuds sont des objets et les arcs (numérotés selon la chronologie) les échanges entre objets. En fait, diagramme de séquence et diagramme de communication sont deux vues différentes mais logiquement équivalentes (on peut construire l'une à partir de l'autre) d'une même chronologie. Ce sont des diagrammes d'interaction (section 7).

### 1.4.4 Comment présenter un modèle UML ?

La présentation d'un modèle UML se compose de plusieurs documents écrits en langage courant et d'un document formalisé : elle ne doit pas se limiter au seul document formalisé car celui-ci est pratiquement incompréhensible si on le présente seul. Un expert en UML sera capable dans certains cas de reconstituer les intentions initiales en lisant le

modèle, mais pas toujours ; et les experts en UML sont rares. Voici la liste des documents qui paraissent nécessaires :

**Présentation stratégique :**
elle décrit pourquoi l'entreprise a voulu se doter de l'outil considéré, les buts qu'elle cherche à atteindre, le calendrier de réalisation prévu, etc.

**Présentation des processus de travail par lesquels la stratégie entend se réaliser :**
pour permettre au lecteur de voir comment l'application va fonctionner en pratique, elle doit être illustrée par une esquisse des écrans qui seront affichés devant les utilisateurs de terrain.

**Explication des choix qui ont guidé la modélisation formelle :**
il s'agit de synthétiser, sous les yeux du lecteur, les discussions qui ont présidé à ces choix.

**Modèle formel :**
c'est le document le plus épais et le plus difficile à lire. Il est préférable de le présenter sur l'Intranet de l'entreprise. En effet, les diagrammes peuvent être alors équipés de liens hypertextes permettant l'ouverture de diagrammes plus détaillés ou de commentaires.

On doit présenter en premier le diagramme de cas d'utilisation qui montre l'enchaînement des cas d'utilisation au sein du processus, enchaînement immédiatement compréhensible ; puis le diagramme d'activités, qui montre le contenu de chaque cas d'utilisation ; puis le diagramme de séquence, qui montre l'enchaînement chronologique des opérations à l'intérieur de chaque cas d'utilisation. Enfin, le diagramme de classes, qui est le plus précis conceptuellement mais aussi le plus difficile à lire car il présente chacune des classes et leurs relations (agrégation, héritage, association, etc.).

---

[5] L'OMG (*Object Management Group*) est une association américaine à but non-lucratif créée en 1989 dont l'objectif est de standardiser et promouvoir le modèle objet sous toutes ses formes. L'OMG est notamment à la base des spécifications UML, MOF, CORBA et IDL. L'OMG est aussi à l'origine de la recommandation MDA

# Chapitre 2 Diagramme de cas d'utilisation (*Use Case Diagram*)

## 2.1 Introduction

Bien souvent, la maîtrise d'ouvrage et les utilisateurs ne sont pas des informaticiens. Il leur faut donc un moyen simple d'exprimer leurs besoins. C'est précisément le rôle des diagrammes de cas d'utilisation qui permettent de recueillir, d'analyser et d'organiser les besoins, et de recenser les grandes fonctionnalités d'un système. Il s'agit donc de la première étape UML d'analyse d'un système.

Un diagramme de cas d'utilisation capture le comportement d'un système, d'un sous-système, d'une classe ou d'un composant tel qu'un utilisateur extérieur le voit. Il scinde la fonctionnalité du système en unités cohérentes, les cas d'utilisation, ayant un sens pour les acteurs. Les cas d'utilisation permettent d'exprimer le besoin des utilisateurs d'un système, ils sont donc une vision orientée utilisateur de ce besoin au contraire d'une vision informatique.

Il ne faut pas négliger cette première étape pour produire un logiciel conforme aux attentes des utilisateurs. Pour élaborer les cas d'utilisation, il faut se fonder sur des entretiens avec les utilisateurs.

## 2.2 Éléments des diagrammes de cas d'utilisation

### 2.2.1 Acteur

Un acteur est l'idéalisation d'un rôle joué par une personne externe, un processus ou une chose qui interagit avec un système.

Il se représente par un petit bonhomme (figure 2.1) avec son nom (*i.e.* son rôle) inscrit dessous.


Figure 2.1: Exemple de représentation d'un acteur

Il est également possible de représenter un acteur sous la forme d'un classeur (cf. section 2.4.3) stéréotypé (cf. section 2.4.4) << *actor* >> (figure 2.2).


Figure 2.2: Exemple de représentation d'un acteur sous la forme d'un classeur

### 2.2.2 Cas d'utilisation

Un cas d'utilisation est une unité cohérente représentant une fonctionnalité visible de l'extérieur. Il réalise un service de bout en bout, avec un déclenchement, un déroulement et une fin, pour l'acteur qui l'initie. Un cas d'utilisation modélise donc un service rendu par le système, sans imposer le mode de réalisation de ce service.

Un cas d'utilisation se représente par une ellipse (figure 2.3) contenant le nom du cas (un verbe à l'infinitif), et optionnellement, au-dessus du nom, un stéréotype (cf. section 2.4.4).

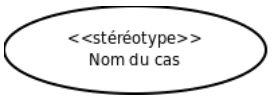


Figure 2.3: Exemple de représentation d'un cas d'utilisation

Dans le cas où l'on désire présenter les attributs ou les opérations du cas d'utilisation, il est préférable de le représenter sous la forme d'un classeur stéréotypé << *use case* >> (figure 2.4). Nous reviendrons sur les notions d'attributs ou d'opération lorsque nous aborderons les diagrammes de classes et d'objets (section 3).

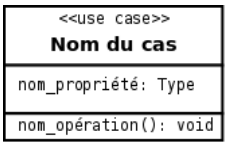



Figure 2.4: Exemple de représentation d'un cas d'utilisation sous la forme d'un classeur

### 2.2.3 Représentation d'un diagramme de cas d'utilisation

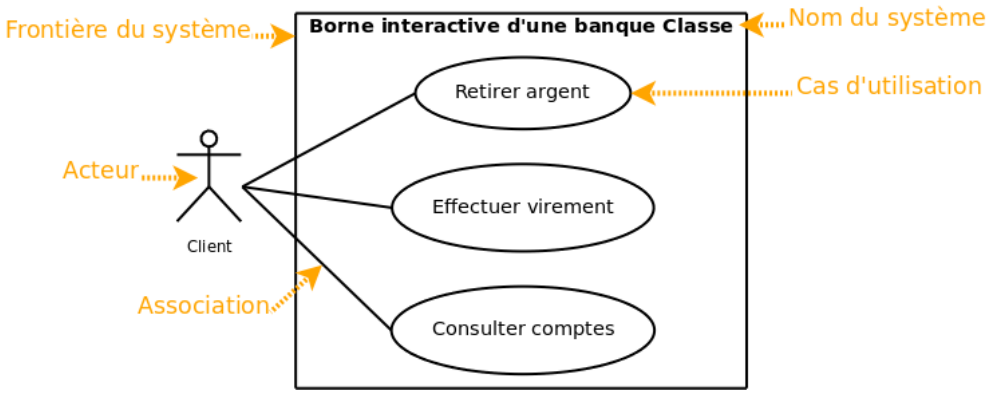



Figure 2.5: Exemple simplifié de diagramme de cas d'utilisation modélisant une borne d'accès à une banque.

Comme le montre la figure 2.5, la frontière du système est représentée par un cadre. Le nom du système figure à l'intérieur du cadre, en haut. Les acteurs sont à l'extérieur et les cas d'utilisation à l'intérieur.

## 2.3 Relations dans les diagrammes de cas d'utilisation

### 2.3.1 Relations entre acteurs et cas d'utilisation

#### Relation d'association

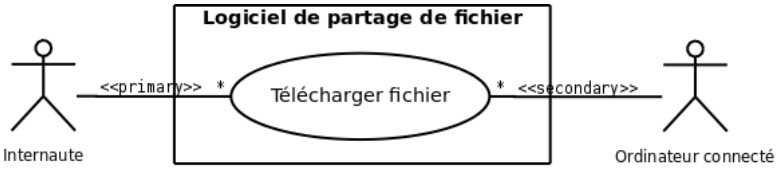



Figure 2.6: Diagramme de cas d'utilisation représentant un logiciel de partage de fichiers

Une relation d'association est chemin de communication entre un acteur et un cas d'utilisation et est représenté un trait continu (cf. figure 2.5 ou 2.6).

#### Multiplicité

Lorsqu'un acteur peut interagir plusieur fois avec un cas d'utilisation, il est possible d'ajouter une multiplicité sur l'association du côté du cas d'utilisation. Le symbole * signifie *plusieurs* (figure 2.6), exactement *n* s'écrit tout simplement *n*, *n..m* signifie entre *n* et *m*, etc. Préciser une multiplicité sur une relation n'implique pas nécessairement que les cas sont utilisés en même temps.

La notion de multiplicité n'est pas propre au diagramme de cas d'utilisation. Nous en reparlerons dans le chapitre consacré au diagramme de classes section 3.3.4.

#### Acteurs principaux et secondaires

Un acteur est qualifié de *principal* pour un cas d'utilisation lorsque ce cas rend service à cet acteur. Les autres acteurs sont alors qualifiés de *secondaires*. Un cas d'utilisation a au plus un acteur principal. Un acteur principal obtient un résultat observable du système tandis qu'un acteur secondaire est sollicité pour des informations complémentaires. En général, l'acteur principal initie le cas d'utilisation par ses sollicitations. Le stéréotype << *primary* >> vient orner l'association reliant un cas d'utilisation à son acteur principal, le stéréotype << *secondary* >> est utilisé pour les acteurs secondaires (figure 2.6).

#### Cas d'utilisation interne

Quand un cas n'est pas directement relié à un acteur, il est qualifié de *cas d'utilisation interne*.

### 2.3.2 Relations entre cas d'utilisation

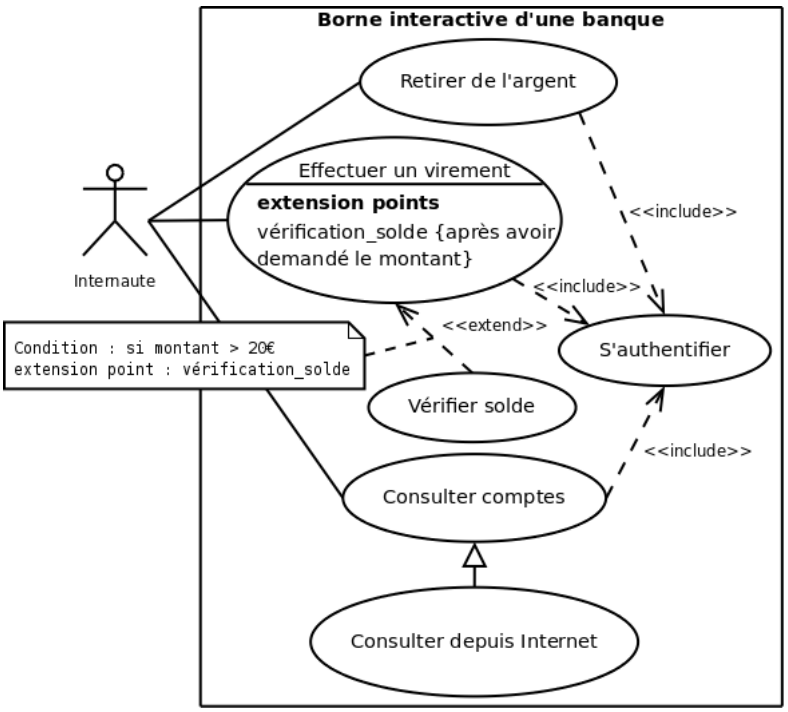



Figure 2.7: Exemple de diagramme de cas d'utilisation

**Types et représentations**

Il existe principalement deux types de relations :

- les dépendances stéréotypées, qui sont explicitées par un stéréotype (les plus utilisés sont l'inclusion et l'extension),
- et la généralisation/spécialisation.

Une dépendance se représente par une flèche avec un trait pointillé (figure 2.7). Si le cas A inclut ou étend le cas B, la flèche est dirigée de A vers B.

Le symbole utilisé pour la généralisation est un flèche avec un trait pleins dont la pointe est un triangle fermé désignant le cas le plus général (figure 2.7).

**Relation d'inclusion**

Un cas A inclut un cas B si le comportement décrit par le cas A inclut le comportement du cas B : le cas A dépend de B. Lorsque A est sollicité, B l'est obligatoirement, comme une partie de A. Cette dépendance est symbolisée par le stéréotype << *include* >> (figure 2.7). Par exemple, l'accès aux informations d'un compte bancaire inclut nécessairement une phase d'authentification avec un identifiant et un mot de passe (figure 2.7).

Les inclusions permettent essentiellement de factoriser une partie de la description d'un cas d'utilisation qui serait commune à d'autres cas d'utilisation (cf. le cas *S'authentifier* de la figure 2.7).

Les inclusions permettent également de décomposer un cas complexe en sous-cas plus simples (figure 2.8). Cependant, il ne faut surtout pas abuser de ce type de décomposition : il faut éviter de réaliser du découpage fonctionnel d'un cas d'utilisation en plusieurs *sous-cas d'utilisation* pour ne pas retomber dans le travers de la décomposition fonctionnelle.

Attention également au fait que, les cas d'utilisation ne s'enchaînent pas, puisqu'il n'y a aucune représentation temporelle dans un diagramme de cas d'utilisation.

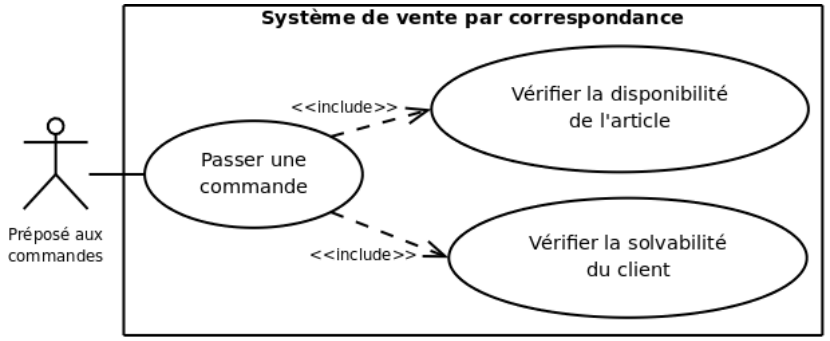



Figure 2.8: Relations entre cas pour décomposer un cas complexe

**Relation d'extension**

La relation d'extension est probablement la plus utile car elle a une sémantique qui a un sens du point de vue métier au contraire des deux autres qui sont plus des artifices d'informaticiens.

On dit qu'un cas d'utilisation A étend un cas d'utilisation B lorsque le cas d'utilisation A peut être appelé au cours de l'exécution du cas d'utilisation B. Exécuter B peut éventuellement entraîner l'exécution de A : contrairement à l'inclusion, l'extension est optionnelle. Cette dépendance est symbolisée par le stéréotype << *extend* >> (figure

2.7).

L'extension peut intervenir à un point précis du cas étendu. Ce point s'appelle le point d'extension. Il porte un nom, qui figure dans un compartiment du cas étendu sous la rubrique *point d'extension*, et est éventuellement associé à une contrainte indiquant le moment où l'extension intervient. Une extension est souvent soumise à condition. Graphiquement, la condition est exprimée sous la forme d'une note. La figure 2.7 présente l'exemple d'une banque où la vérification du solde du compte n'intervient que si la demande de retrait dépasse 20 euros.

**Relation de généralisation**

Un cas A est une généralisation d'un cas B si B est un cas particulier de A. Dans la figure 2.7, la consultation d'un compte *via* Internet est un cas particulier de la consultation. Cette relation de généralisation/spécialisation est présente dans la plupart des diagrammes UML et se traduit par le concept d'héritage dans les langages orientés objet.

### 2.3.3 Relations entre acteurs

La seule relation possible entre deux acteurs est la généralisation : un acteur A est une généralisation d'un acteur B si l'acteur A peut être substitué par l'acteur B. Dans ce cas, tous les cas d'utilisation accessibles à A le sont aussi à B, mais l'inverse n'est pas vrai.

Le symbole utilisé pour la généralisation entre acteurs est une flèche avec un trait plein dont la pointe est un triangle fermé désignant l'acteur le plus général (comme nous l'avons déjà vu pour la relation de généralisation entre cas d'utilisation).

Par exemple, la figure 2.9 montre que le directeur des ventes est un préposé aux commandes avec un pouvoir supplémentaire : en plus de pouvoir passer et suivre une commande, il peut gérer le stock. Par contre, le préposé aux commandes ne peut pas gérer le stock.

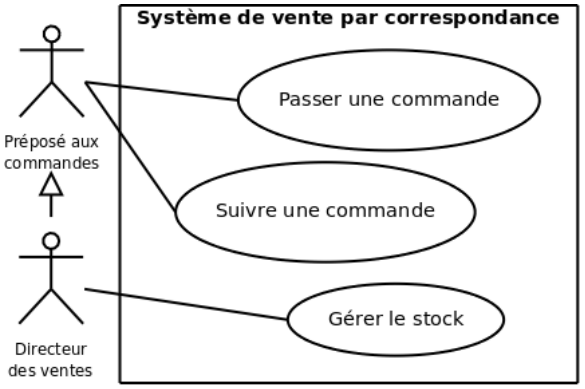



Figure 2.9: Relations entre acteurs

## 2.4 Notions générales du langage UML

Les éléments du langage UML que nous abordons ici ne sont pas spécifiques au diagramme de cas d'utilisation mais sont généraux. Nous avons déjà utilisé certains de ces éléments dans ce chapitre et nous utiliserons les autres dans les chapitres qui suivent, notamment dans le chapitre sur les diagrammes de classes (section 3).

### 2.4.1 Paquetage

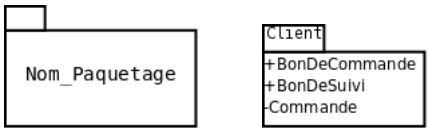



Figure 2.10: Représentations d'un paquetage

Un paquetage est un regroupement d'éléments de modèle et de diagrammes. Il permet ainsi d'organiser des éléments de modélisation en groupes. Il peut contenir tout type d'élément de modèle : des classes, des cas d'utilisation, des interfaces, des diagrammes, … et même des paquetages imbriqués (décomposition hiérarchique).

Un paquetage se représente comme un dossier avec son nom inscrit dedans (figure 2.10, diagramme de gauche). Il est possible de représenter explicitement le contenu d'un paquetage. Dans ce cas, le nom du paquetage est placé dans l'onglet (figure 2.10, diagramme de droite).

Les éléments contenus dans un paquetage doivent représenter un ensemble fortement cohérent et sont généralement de même nature et de même niveau sémantique. Tout élément n'appartient qu'à un seul paquetage. Les paquetage constituent un mécanisme de gestion important des problèmes de grande taille. Ils permettent d'éviter les grands modèles plats et de cloisonner des éléments constitutifs d'un système évoluant à des rythmes différents ou développés par des équipes différentes.

Il existe un paquetage racine unique, éventuellement anonyme, qui contient la totalité des modèles d'un système.

### 2.4.2 Espace de noms

Les espaces de noms sont des paquetages, des classeurs, etc. On peut déterminer un élément nommé de façon unique par son nom qualifié, qui est constitué de la série des noms des paquetages ou des autres espaces de noms depuis la racine jusqu'à l'élément en question. Dans un nom qualifié, chaque espace de nom est séparé par deux doubles points (`::`).

Par exemple, si un paquetage B est inclus dans un paquetage A et contient une classe X, il faut écrire `A::B::X` pour pouvoir utiliser la classe X en dehors du contexte du paquetage B.

### 2.4.3 Classeur

Les paquetages et les relations de généralisation ne peuvent avoir d'instance. D'une manière générale, les éléments de modélisation pouvant en avoir sont représentés dans des classeurs[1]. Plus important encore, un classeur est un élément de modèle qui décrit une unité structurelle ou comportementale.

Un classeur modélise un concept discret qui décrit un élément (*i.e.* objet) doté d'une identité (*i.e.* un nom), d'une structure ou d'un état (*i.e.* des attributs), d'un comportement (*i.e.* des opérations), de relations et d'une structure interne facultative. Il peut participer à des relations d'association, de généralisation, de dépendance et de contrainte. On le déclare dans un espace de noms, comme un paquetage ou une autre classe. Un classeur se représente par un rectangle, en traits pleins, contenant éventuellement des compartiments.

Les acteurs et les cas d'utilisation sont des classeurs. Tout au long de ce cours, nous retrouverons le terme de *classeur* car cette notion englobe aussi les classes, les interfaces, les signaux, les nœuds, les composants, les sous-systèmes, etc. Le type de classeur le plus important étant, bien évidemment, la classe (cf. section 3).

### 2.4.4 Stéréotype

Un stéréotype est une annotation s'appliquant sur un élément de modèle. Il n'a pas de définition formelle, mais permet de mieux caractériser des *variétés* d'un même concept. Il permet donc d'adapter le langage à des situations particulières. Il est représenté par une chaînes de caractères entre guillemets (<< >>) dans, ou à proximité du symbole de l'élément de modèle de base.

Par exemple, la figure 2.4 représente un cas d'utilisation par un rectangle. UML utilise aussi les rectangles pour représenter les classes (cf. section 3). La notation n'est cependant pas ambiguë grâce à la présence du stéréotype << *use case* >>.

### 2.4.5 Note

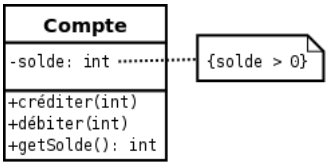



Figure 2.11: Exemple d'utilisation d'une note pour préciser que le solde d'un compte doit toujours être positif.

Une note contient une information textuelle comme un commentaire, un corps de méthode ou une contrainte. Graphiquement, elle est représentée par un rectangle dont l'angle supérieur droit est plié. Le texte contenu dans le rectangle n'est pas contraint par UML. Une note n'indique pas explicitement le type d'élément qu'elle contient, toute l'intelligibilité d'une note doit être contenu dans le texte même. On peut relier une note à l'élément qu'elle décrit grâce à une ligne en pointillés. Si elle décrit plusieurs éléments, on dessine une ligne vers chacun d'entre eux.

L'exemple de la figure 2.11 montre une note exprimant une contrainte (cf. section 4.1) sur un attribut.

[1] Certains éléments, comme les associations, peuvent avoir des instances bien qu'ils ne soient pas représentés dans des classeurs.

## 2.5 Modélisation des besoins avec UML

### 2.5.1 Comment identifier les acteurs ?

UML n'emploie pas le terme *d'utilisateur* mais d'*acteur*. Les acteurs d'un système sont les entités externes à ce système qui interagissent (saisie de données, réception d'information, …) avec lui. Les acteurs sont donc à l'extérieur du système et dialoguent avec lui. Ces acteurs permettent de cerner l'interface que le système va devoir offrir à son environnement. Oublier des acteurs ou en identifier de faux conduit donc nécessairement à se tromper sur l'interface et donc la définition du système à produire.

Il faut faire attention à ne pas confondre acteurs et utilisateurs (utilisateur avec le sens de la personne physique qui va appuyer sur un bouton) d'un système. D'une part parce que les acteurs inclus les utilisateurs humains mais aussi les autres systèmes informatiques ou hardware qui vont communiquer avec le système. D'autre part parce que un acteur englobe tout une classe d'utilisateur. Ainsi, plusieurs utilisateurs peuvent avoir le même rôle, et donc correspondre à un même acteur, et une même personne physique peut jouer des rôles différents vis-à-vis du système, et donc correspondre à plusieurs acteurs.

Chaque acteur doit être nommé. Ce nom doit refléter sont rôle car un acteur représente un ensemble cohérent de rôles joués vis-à-vis du système.

Pour trouver les acteurs d'un système, il faut identifier quels sont les différents rôles que vont devoir jouer ses utilisateurs (ex : responsable clientèle, responsable d'agence, administrateur, approbateur, …). Il faut également s'intéresser aux autres systèmes avec lesquels le système va devoir communiquer comme :

- les périphériques manipulés par le système (imprimantes, hardware d'un distributeur de billet, …) ;
- des logiciels déjà disponibles à intégrer dans le projet ;
- des systèmes informatiques externes au système mais qui interagissent avec lui, etc.

Pour faciliter la recherche des acteurs, on peut imaginer les frontières du système. Tout ce qui est à l'extérieur et qui interagit avec le système est un acteur, tout ce qui est à l'intérieur est une fonctionnalité à réaliser.

Vérifiez que les acteurs communiquent bien *directement* avec le système par émission ou réception de messages. Une erreur fréquente consiste à répertorier en tant qu'acteur des entités externes qui n'interagissent pas directement avec le système, mais uniquement par le biais d'un des véritables acteurs. Par exemple, l'hôtesse de caisse d'un magasin de grande distribution est un acteur pour la caisse enregistreuse, par contre, les clients du magasins ne correspondent pas à un acteur car ils n'interagissent pas directement avec la caisse.

### 2.5.2 Comment recenser les cas d'utilisation ?

L'ensemble des cas d'utilisation doit décrire exhaustivement les exigences fonctionnelles du système. Chaque cas d'utilisation correspond donc à une fonction métier du système, selon le point de vue d'un de ses acteurs. Aussi, pour identifier les cas d'utilisation, il faut se placer du point de vue de chaque acteur et déterminer comment et surtout pourquoi il se sert du système. Il faut éviter les redondances et limiter le nombre de cas en se situant à un bon niveau d'abstraction. Trouver le bon niveau de détail pour les cas d'utilisation est un problème difficile qui nécessite de l'expérience.

Nommez les cas d'utilisation avec un verbe à l'infinitif suivi d'un complément en vous plaçant du point de vue de l'acteur et non pas de celui du système. Par exemple, un distributeur de billets aura probablement un cas d'utilisation *Retirer de l'argent* et non pas *Distribuer de l'argent*.

De par la nature fonctionnelle, et non objet, des cas d'utilisation, et en raison de la difficulté de trouver le bon niveau de détail, il faut être très vigilant pour ne pas retomber dans une décomposition fonctionnelle descendante hiérarchique. Un nombre trop important de cas d'utilisation est en général le symptôme de ce type d'erreur.

Dans tous les cas, il faut bien garder à l'esprit qu'il n'y a pas de notion temporelle dans un diagramme de cas d'utilisation.

### 2.5.3 Description textuelle des cas d'utilisation

Le diagramme de cas d'utilisation décrit les grandes fonctions d'un système du point de vue des acteurs, mais n'expose pas de façon détaillée le dialogue entre les acteurs et les cas d'utilisation. Bien que de nombreux diagrammes d'UML permettent de décrire un cas, il est recommandé de rédiger une description textuelle car c'est une forme souple qui convient dans bien des situations.

Une description textuelle couramment utilisée se compose de trois parties.

1. La première partie permet d'identifier le cas, elle doit contenir les informations qui suivent.

   **Nom :**
   Utiliser une tournure à l'infinitif (ex : Réceptionner un colis).

   **Objectif :**
   Une description résumée permettant de comprendre l'intention principale du cas d'utilisation. Cette partie est souvent renseignée au début du projet dans la phase de découverte des cas d'utilisation.

   **Acteurs principaux :**
   Ceux qui vont réaliser le cas d'utilisation (la relation avec le cas d'utilisation est illustrée par le trait liant le cas d'utilisation et l'acteur dans un diagramme de cas d'utilisation)

   **Acteurs secondaires :**
   Ceux qui ne font que recevoir des informations à l'issue de la réalisation du cas d'utilisation

   **Dates :**
   Les dates de créations et de mise à jour de la description courante.

   **Responsable :**
   Le nom des responsables.

   **Version :**
   Le numéro de version.

2. La deuxième partie contient la description du fonctionnement du cas sous la forme d'une séquence de messages échangés entre les acteurs et le système. Elle contient toujours une séquence nominale qui décrit de déroulement normal du cas. À la séquence nominale s'ajoutent fréquemment des séquences alternatives (des embranchement dans la séquence nominale) et des séquences d'exceptions (qui interviennent quand une erreur se produit).

   **Les préconditions :**
   elles décrivent dans quel état doit être le système (l'application) avant que ce cas d'utilisation puisse être déclenché.

   **Des scénarii :**
   Ces scénarii sont décrits sous la forme d'échanges d'évènements entre l'acteur et le système. On distingue le scénario nominal, qui se déroule quand il n'y a pas d'erreur, des scénarii alternatifs qui sont les variantes du scénario nominal et enfin les scénarii d'exception qui décrivent les cas d'erreurs.

   **Des postconditions :**
   Elle décrivent l'état du système à l'issue des différents scénarii.

3. La troisième partie de la description d'un cas d'utilisation est une rubrique optionnelle. Elle contient généralement des spécifications non fonctionnelles (spécifications techniques, …). Elle peut éventuellement contenir une description des besoins en termes d'interface graphique.

### 2.5.4 Remarques

**Concernant les relations dans les cas d'utilisation**

Il est important de noter que l'utilisation des relations n'est pas primordiale dans la rédaction des cas d'utilisation et donc dans l'expression du besoin. Ces relations peuvent être utiles dans certains cas mais une trop forte focalisation sur leur usage conduit souvent à une perte de temps ou à un usage faussé, pour une valeur ajoutée, au final, relativement faible.

**Concernant les cas d'utilisation**

Unanimement reconnus comme cantonnés à l'ingénierie des besoins, les diagrammes de cas d'utilisation ne peuvent être qualifiés de modélisation à proprement parler. D'ailleur, de nombreux éléments descriptifs sont en langage naturel. De plus, ils ne correspondent pas *stricto sensu* à une approche objet. En effet, capturer les besoins, les découvrir, les réfuter, les consolider, etc., correspond plus à une analyse fonctionnelle classique.

**Les *Use case Realization***

UML ne mentionne que le fait que la réalisation d'un cas d'utilisation est décrit par une suite de collaborations entre éléments de modélisation mais ne parle par de l'élément de modélisation *use case realization*. Les *use case realization* ne sont pas un formalisme d'UML mais du RUP (*Rational Unified Process*).

Après avoir rédigé les cas d'utilisation, il faut identifier des objets, des classes, des données et des traitements qui vont permettre au système de supporter ces cas d'utilisation. Pour documenter la manière dont sont mis en œuvre les cas d'utilisation du système, on peut utiliser le mécanisme des *use case realization*. Ils permettent de regrouper un diagramme de classes et des diagrammes d'interaction. On retrouvera dans le diagramme de classes les classes qui mettent en œuvre le cas d'utilisation associé au *use case realization*. On retrouvera dans les différents diagrammes d'interaction une documentation des différents évènements échangés entre les objets afin de réaliser les différents scénarii décrit dans le cas d'utilisation.

Au final on aura un *use case realization* par cas d'utilisation et dans chaque *use case realization* on aura autant de diagrammes d'interaction que nécessaire pour illustrer les scénarii décrits dans le cas d'utilisation (scénario nominal, scénarii alternatifs et scénarii d'exception).

Les *use case realization* permettent donc, dans la pratique, d'apporter un élément de réponse à la question : Comment structurer mon modèle UML ?

# Chapitre 3 Diagramme de classes (*Class Diagram*)

## 3.1 Introduction

Le diagramme de classes est considéré comme le plus important de la modélisation orientée objet, il est le seul obligatoire lors d'une telle modélisation.

Alors que le diagramme de cas d'utilisation montre un système du point de vue des acteurs, le diagramme de classes en montre la structure interne. Il permet de fournir une représentation abstraite des objets du système qui vont interagir ensemble pour réaliser les cas d'utilisation. Il est important de noter qu'un même objet peut très bien intervenir dans la réalisation de plusieurs cas d'utilisation. Les cas d'utilisation ne réalisent donc pas une partition[1] des classes du diagramme de classes. Un diagramme de classes n'est donc pas adapté (sauf cas particulier) pour détailler, décomposer, ou illustrer la réalisation d'un cas d'utilisation particulier.

Il s'agit d'une vue statique car on ne tient pas compte du facteur temporel dans le comportement du système. Le diagramme de classes modélise les conceps du domaine d'application ainsi que les concepts internes créés de toutes pièces dans le cadre de l'implémentation d'une application. Chaque langage de Programmation Orienté Objets donne un moyen spécifique d'implémenter le paradigme objet (pointeurs ou pas, héritage multiple ou pas, etc.), mais le diagramme de classes permet de modéliser les classes du système et leurs relations indépendamment d'un langage de programmation particulier.

Les principaux éléments de cette vue statique sont les classes et leurs relations : association, généralisation et plusieurs types de dépendances, telles que la réalisation et l'utilisation.

---

[1] Une partition d'un ensemble est un ensemble de parties non vides de cet ensemble, deux à deux disjointes et dont la réunion est égale à l'ensemble.

## 3.2 Les classes

### 3.2.1 Notions de classe et d'instance de classe

Une *instance* est une concrétisation d'un concept abstrait. Par exemple :

- la Ferrari *Enzo* qui se trouve dans votre garage est une instance du concept abstrait *Automobile* ;
- l'amitié qui lie Jean et Marie est une instance du concept abstrait *Amitié* ;

Une classe est un concept abstrait représentant des éléments variés comme :

- des éléments concrets (ex : des avions),
- des éléments abstraits ( ex : des commandes de marchandises ou services),
- des composants d'une application (ex : les boutons des boîtes de dialogue),
- des structures informatiques (ex : des tables de hachage),
- des éléments comportementaux (ex : des tâches), etc.

Tout système orienté objet est organisé autour des classes.

Une classe est la description formelle d'un ensemble d'objets ayant une sémantique et des caractéristiques communes.

Un objet est une instance d'une classe. C'est une entité discrète dotée d'une identité, d'un état et d'un comportement que l'on peut invoquer. Les objets sont des éléments individuels d'un système en cours d'exécution.

Par exemple, si l'on considère que *Homme* (au sens être humain) est un concept abstrait, on peut dire que la personne Marie-Cécile est une instance de Homme. Si *Homme* était une classe, Marie-Cécile en serait une instance : un objet.

### 3.2.2 Caractéristiques d'une classe

Une classe définit un jeu d'objets dotés de caractéristiques communes. Les caractéristiques d'un objet permettent de spécifier son *état* et son *comportement*. Dans les sections 1.3.2 et 1.3.4, nous avons dit que les caractéristiques d'un objet étaient soit des attributs, soit des opérations. Ce n'est pas exact dans un diagramme de classe car *les terminaisons d'associations sont des propriétés qui peuvent faire partie des caractéristiques d'un objet au même titre que les attributs et les opérations* (cf. section 3.3.2).

**État d'un objet :**

Ce sont les attributs et généralement les *terminaisons d'associations*, tous deux réunis sous le terme de *propriétés structurelles*, ou tout simplement *propriétés*[2], qui décrivent l'état d'un objet. Les attributs sont utilisés pour des valeurs de données pures, dépourvues d'identité, telles que les nombres et les chaînes de caractères. Les associations sont utilisées pour connecter les classes du diagramme de classe ; dans ce cas, la terminaison de l'association (du côté de la classe cible) est généralement une propriété de la classe de base (cf. section 3.3.1 et 3.3.2).

Les propriétés décrites par les attributs prennent des valeurs lorsque la classe est instanciée. L'instance d'une association est appelée un lien.

**Comportement d'un objet :**

Les opérations décrivent les éléments individuels d'un comportement que l'on peut invoquer. Ce sont des fonctions qui peuvent prendre des valeurs en entrée et modifier les attributs ou produire des résultats.

Une opération est la spécification (*i.e.* déclaration) d'une méthode. L'implémentation (*i.e.* définition) d'une méthode est également appelée méthode. Il y a donc une ambiguïté sur le terme *méthode*.

Les attributs, les terminaisons d'association et les méthodes constituent donc les caractéristiques d'une classe (et de ses instances).

### 3.2.3 Représentation graphique

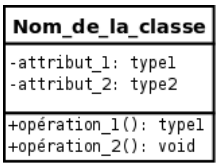



Figure 3.1: Représentation UML d'une classe

Une classe est un classeur [3]. Elle est représentée par un rectangle divisé en trois à cinq compartiments (figure 3.1).

Le premier indique le nom de la classe (cf. section 3.2.5), le deuxième ses attributs (cf. section 3.2.6) et le troisième ses opérations (cf. section 3.2.7). Un compartiment des responsabilités peut être ajouté pour énumérer l'ensemble de tâches devant être assurées par la classe mais pour lesquelles on ne dispose pas encore assez d'informations. Un compartiment des exceptions peut également être ajouté pour énumérer les situations exceptionnelles devant être gérées par la classe.

### 3.2.4 Encapsulation, visibilité, interface

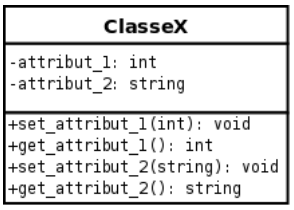



Figure 3.2: Bonnes pratiques concernant la manipulation des attributs.

Nous avons déjà abordé cette problématique section 1.3.4. L'encapsulation est un mécanisme consistant à rassembler les données et les méthodes au sein d'une structure en cachant l'implémentation de l'objet, c'est-à-dire en empêchant l'accès aux données par un autre moyen que les services proposés. Ces services accessibles (offerts) aux utilisateurs de l'objet définissent ce que l'on appel l'interface de l'objet (sa vue externe). L'encapsulation permet donc de garantir l'intégrité des données contenues dans l'objet.

L'encapsulation permet de définir des niveaux de visibilité des éléments d'un conteneur. La visibilité déclare la possibilité pour un élément de modélisation de référencer un élément qui se trouve dans un espace de noms différents de celui de l'élément qui établit la référence. Elle fait partie de la relation entre un élément et le conteneur qui l'héberge, ce dernier pouvant être un paquetage, une classe ou un autre espace de noms. Il existe quatre visibilités prédéfinies.

***Public* ou + :**
tout élément qui peut voir le conteneur peut également voir l'élément indiqué.

***Protected* ou # :**
seul un élément situé dans le conteneur ou un de ses descendants peut voir l'élément indiqué.

***Private* ou - :**
seul un élément situé dans le conteneur peut voir l'élément.

***Package* ou ~ ou rien :**
seul un élément déclaré dans le même paquetage peut voir l'élément.

Par ailleur, UML 2.0 donne la possibilité d'utiliser n'importe quel langage de programmation pour la spécification de la visibilité.

Dans une classe, le marqueur de visibilité se situe au niveau de chacune de ses caractéristiques (attributs, terminaisons d'association et opération). Il permet d'indiquer si une autre classe peut y accéder.

Dans un paquetage, le marqueur de visibilité se situe sur des éléments contenus directement dans le paquetage, comme les classes, les paquetages imbriqués, etc. Il indique si un autre paquetage susceptible d'accéder au premier paquetage peut voir les éléments.

Dans la pratique, lorsque des attributs doivent être accessibles de l'extérieur, il est préférable que cet accès ne soit pas direct mais se fasse par l'intermédiaire d'opérations (figure 3.2).

### 3.2.5 Nom d'une classe

Le nom de la classe doit évoquer le concept décrit par la classe. Il commence par une majuscule. On peut ajouter des informations subsidiaires comme le nom de l'auteur de la modélisation, la date, etc. Pour indiquer qu'une classe est abstraite, il faut ajouter le mot-clef *abstract*.

La syntaxe de base de la déclaration d'un nom d'une classe est la suivante :

```
[ <Nom_du_paquetage_1>::...::<Nom_du_paquetage_N> ]
<Nom_de_la_classe> [ { [abstract], [<auteur>], [<date>], ... } ]
```

**Méta-langage des syntaxes –**

Nous aurons régulièrement recours à ce méta-langage pour décrire des syntaxes de déclaration. Ce méta-langage contient certains méta-caractère :

**`[ ]` :**
les crochets indiquent que ce qui est à l'intérieur est optionnel ;

**`< >` :**
les signes inférieur et supérieur indiquent que ce qui est à l'intérieur est plus ou moins libre ; par exemple, la syntaxe de déclaration d'une variable comme `compteur : int` est `<nom_variable> : <type>` ;

**`' '` :**

les cotes sont utiles quand on veut utiliser un méta-caractère comme un caractère ; par exemple, pour désigner un crochet (`[`) il faut écrire `'['` car `[` est un méta-caractère ayant une signification spéciale ;

**`...`** :
permet de désigner une suite de séquence de longueur non définie, le contexte permettant de comprendre de quelle suite il s'agit.

### 3.2.6 Les attributs

#### Attributs de la classe

Les attributs définissent des informations qu'une classe ou un objet doivent connaître. Ils représentent les données encapsulées dans les objets de cette classe. Chacune de ces informations est définie par un nom, un type de données, une visibilité et peut être initialisé. Le nom de l'attribut doit être unique dans la classe. La syntaxe de la déclaration d'un attribut est la suivante :

```
<visibilité> [/] <nom_attribut> :
<type> [ '['<multiplicité>']' [{<contrainte>}] ] [ = <valeur_par_défaut> ]
```

Le type de l'attribut (`<type>`) peut être un nom de classe, un nom d'interface ou un type de donné prédéfini. La multiplicité (`<multiplicité>`) d'un attribut précise le nombre de valeurs que l'attribut peut contenir. Lorsqu'une multiplicité supérieure à 1 est précisée, il est possible d'ajouter une contrainte (`{<contrainte>}`) pour préciser si les valeurs sont ordonnées (*{ordered}*) ou pas (*{list}*).

#### Attributs de classe

Par défaut, chaque instance d'une classe possède sa propre copie des attributs de la classe. Les valeurs des attributs peuvent donc différer d'un objet à un autre. Cependant, il est parfois nécessaire de définir un *attribut de classe* (*static* en Java ou en C++) qui garde une valeur unique et partagée par toutes les instances de la classe. Les instances ont accès à cet attribut mais n'en possèdent pas une copie. Un attribut de classe n'est donc pas une propriété d'une instance mais une propriété de la classe et l'accès à cet attribut ne nécessite pas l'existence d'une instance.

Graphiquement, un attribut de classe est souligné.

#### Attributs dérivés

Les attributs dérivés peuvent être calculés à partir d'autres attributs et de formules de calcul. Lors de la conception, un attribut dérivé peut être utilisé comme marqueur jusqu'à ce que vous puissiez déterminer les règles à lui appliquer.

Les attributs dérivés sont symbolisés par l'ajout d'un « / » devant leur nom.

### 3.2.7 Les méthodes

#### Méthode de la classe

Dans une classe, une opération (même nom et même types de paramètres) doit être unique. Quand le nom d'une opération apparaît plusieurs fois avec des paramètres différents, on dit que l'opération est surchargée. En revanche, il est impossible que deux opérations ne se distinguent que par leur valeur retournée.

La déclaration d'une opération contient les types des paramètres et le type de la valeur de retour, sa syntaxe est la suivante :

```
<visibilité> <nom_méthode> ([<paramètre_1>, ... , <paramètre_N>]) :
[<type_renvoyé>] [{<propriétés>}]
```

La syntaxe de définition d'un paramètre (`<paramètre>`) est la suivante :

```
[<direction>] <nom_paramètre>:<type> ['['<multiplicité>']'] [=<valeur_par_défaut>]
```

La direction peut prendre l'une des valeurs suivante :

**in**
: Paramètre d'entrée passé par valeur. Les modifications du paramètre ne sont pas disponibles pour l'appelant. C'est le comportement par défaut.

**out**
: Paramètre de sortie uniquement. Il n'y a pas de valeur d'entrée et la valeur finale est disponible pour l'appelant.

**inout**
: Paramètre d'entrée/sortie. La valeur finale est disponible pour l'appelant.

Le type du paramètre (`<type>`) peut être un nom de classe, un nom d'interface ou un type de donné prédéfini.

Les propriétés (`<propriétés>`) correspondent à des contraintes ou à des informations complémentaires comme les exceptions, les préconditions, les postconditions ou encore l'indication qu'une méthode est abstraite (mot-clef *abstract*), etc.

#### Méthode de classe

Comme pour les attributs de classe, il est possible de déclarer des méthodes de classe. Une méthode de classe ne peut manipuler que des attributs *de classe* et ses propres paramètres. Cette méthode n'a pas accès aux attributs *de la classe* (*i.e.* des instances de la classe). L'accès à une méthode de classe ne nécessite pas l'existence d'une instance de cette classe.

Graphiquement, une méthode de classe est soulignée.

#### Méthodes et classes abstraites

Une méthode est dite abstraite lorsqu'on connaît son entête mais pas la manière dont elle peut être réalisée (*i.e.* on connaît sa déclaration mais pas sa définition).

Une classe est dite abstraite lorsqu'elle définit au moins une méthode abstraite ou lorsqu'une classe parent (cf. section 3.3.9) contient une méthode abstraite non encore réalisée.

On ne peut instancier une classe abstraite : elle est vouée à se spécialiser (cf. section 3.3.9). Une classe abstraite peut très bien contenir des méthodes concrètes.

Une classe abstraite pure ne comporte que des méthodes abstraites. En programmation orientée objet, une telle classe est appelée une interface.

Pour indiquer qu'une classe est abstraite, il faut ajouter le mot-clef *abstract* derrière son nom.

### 3.2.8 Classe active

Une classe est passive par défaut, elle sauvegarde les données et offre des services aux autres. Une classe active initie et contrôle le flux d'activités.

Graphiquement, une classe active est représentée comme une classe standard dont les lignes verticales du cadre, sur les côtés droit et gauche, sont doublées.

---

[2] Il faut ici aborder un petit problème de terminologie autour du mot *propriété*. En effet, dans la littérature, le mot *propriété* est parfois utilisé pour désigner toutes les caractéristiques d'une classe (*i.e.* les attributs comme les méthodes). Dans ce cas, les attributs et les terminaisons d'association sont rassemblés sous le terme de *propriétés structurelles*, le qualificatif *structurelle* prenant ici toute son importance. D'un autre côté, le mot *propriété* est souvent utilisé dans l'acception du terme anglais *property* (dans la norme UML Superstructure version 2.1.1), qui, lui, ne désigne que les attributs et les terminaisons d'association, c'est-à-dire les *propriétés structurelles*. Pour englober les méthodes, il faut alors utiliser le terme plus générique de *caractéristiques*, qui prend ainsi le rôle de traduction du terme anglais *feature* dans la norme. Dans le présent cours, je m'efforce de me conformer à cette deuxième solution où *propriété* et *propriété structurelle* désignent finalement la même chose.

[3] De manière générale, toute boîte non stéréotypée dans un diagramme de classes est implicitement une classe. Ainsi, le stéréotype *class* est le stéréotype par défaut.

## 3.3 Relations entre classes

### 3.3.1 Notion d'association

Une association est une relation entre deux classes (association binaire) ou plus (association n-aire), qui décrit les connexions structurelles entre leurs instances. Une association indique donc qu'il peut y avoir des liens entre des instances des classes associées.

Comment une association doit-elle être modélisée ? Plus précisément, quelle différence existe-t-il entre les deux diagrammes de la figure 3.3 ?

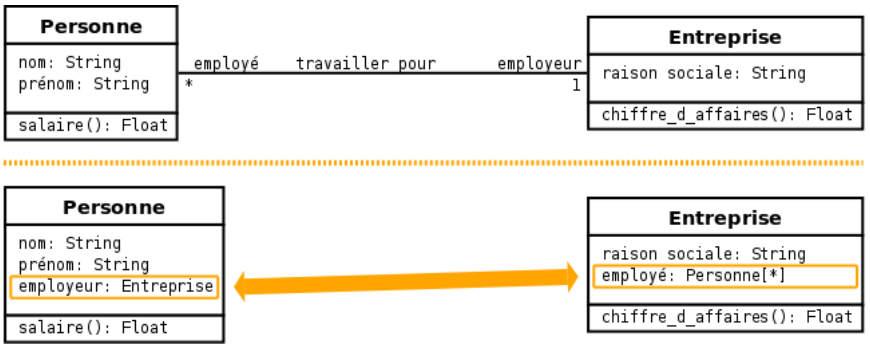



Figure 3.3: Deux façons de modéliser une association.

Dans la première version, l'association apparaît clairement et constitue une entité distincte. Dans la seconde, l'association se manifeste par la présence de deux attributs dans chacune des classes en relation. C'est en fait la manière dont une association est généralement implémentée dans un langage objet quelconque (cf. section 3.6.2), mais pas dans tout langage de représentation (cf. section 3.6.3).

La question de savoir s'il faut modéliser les associations en tant que tel a longtemps fait débat. UML a tranché pour la première version car elle se situe plus à un niveau conceptuel (par opposition au niveau d'implémentation) et simplifie grandement la modélisation d'associations complexes (comme les associations plusieurs à plusieurs par exemple).

Un attribut peut alors être considéré comme une association dégénérée dans laquelle une terminaison d'association[4] est détenue par un classeur (généralement une classe). Le classeur détenant cette terminaison d'association devrait théoriquement se trouver à l'autre extrémité, non modélisée, de l'association. Un attribut n'est donc rien d'autre qu'une terminaison d'un cas particulier d'association (cf. figure 3.9 section 3.3.5).

Ainsi, les terminaisons d'associations et les attributs sont deux éléments conceptuellement très proches que l'on regroupe sous le terme de *propriété*.

### 3.3.2 Terminaison d'association

#### Propriétaire d'une terminaison d'association

La possession d'une terminaison d'association par le classeur situé à l'autre extrémité de l'association peut être spécifié graphiquement par l'adjonction d'un petit cercle plein (point) entre la terminaison d'association et la classe (cf. figure 3.4). Il n'est pas indispensable de préciser la possession des terminaisons d'associations. Dans ce cas, la possession est ambiguë. Par contre, si l'on indique des possessions de terminaisons d'associations, toutes les terminaisons d'associations sont non ambiguë : la présence d'un point implique que la terminaison d'association appartient à la classe située à l'autre extrémité, l'absence du point implique que la terminaison d'association appartient à l'association.

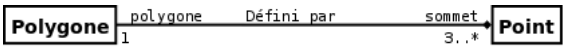



Figure 3.4: Utilisation d'un petit cercle plein pour préciser le propriétaire d'une terminaison d'association.

Par exemple, le diagramme de la figure 3.4 précise que la terminaison d'association *sommet* (*i.e.* la propriété *sommet*) appartient à la classe *Polygone* tandis que la terminaison d'association *polygone* (*i.e.* la propriété *polygone*) appartient à l'association *Défini par*.

**Une terminaison d'association est une propriété**

Une propriété est une caractéristique structurelle. Dans le cas d'une classe, les propriétés sont constituées par les attributs et les éventuelles terminaisons d'association que possède la classe. Dans le cas d'une association, les propriétés sont constituées par les terminaisons d'association que possède l'association. Attention, une association ne possède pas forcément toutes ses terminaisons d'association !

Une propriété peut être paramétrée par les éléments suivant (on s'intéresse ici essentiellement aux terminaisons d'associations puisque les attributs ont été largement traités section 3.2) :

**nom :**
Comme un attribut, une terminaison d'association peut être nommée. Le nom est situé à proximité de la terminaison, mais contrairement à un attribut, ce nom est facultatif. Le nom d'une terminaison d'association est appelée *nom du rôle*. Une association peut donc posséder autant de noms de rôle que de terminaisons (deux pour une association binaire et *n* pour une association n-aire).

**visibilité :**
Comme un attribut, une terminaison d'association possède une visibilité (cf. section 3.2.4). La visibilité est mentionnée à proximité de la terminaison, et plus précisément, le cas échéant, devant le nom de la terminaison.

**multiplicité :**
Comme un attribut, une terminaison d'association peut posséder une multiplicité. Elle est mentionnée à proximité de la terminaison. Il n'est pas impératif de la préciser, mais, contrairement à un attribut dont la multiplicité par défaut est 1, la multiplicité par défaut d'une terminaison d'association est *non spécifiée*. L'interprétation de la multiplicité pour une terminaison d'association est moins évidente que pour un attributs (cf. section 3.3.4).

**navigabilité :**
Pour un attribut, la navigabilité est implicite, navigable, et toujours depuis la classe vers l'attribut. Pour une terminaison d'association, la navigabilité peut être précisée (cf. section 3.3.5).

### 3.3.3 Association binaire et n-aire

**Association binaire**

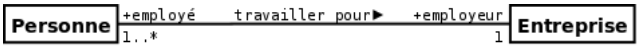



Figure 3.5: Exemple d'association binaire.

Une association binaire est matérialisée par un trait plein entre les classes associées (cf. figure 3.5). Elle peut être ornée d'un nom, avec éventuellement une précision du sens de lecture (▸ ou ◂).

Quand les deux extrémités de l'association pointent vers la même classe, l'association est dite *réflexive*.

**Association n-aire**

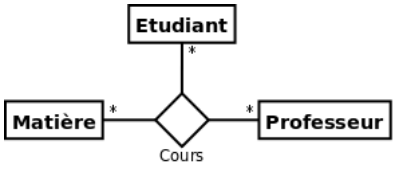



Figure 3.6: Exemple d'association n-aire.

Une association n-aire lie plus de deux classes. La section 3.3.4 détaille comment interpréter les multiplicités d'une association n-aire. La ligne pointillé d'une classe-association (cf. section 3.3.7) peut être reliée au losange par une ligne discontinue pour représenter une association n-aire dotée d'attributs, d'opérations ou d'associations.

On représente une association n-aire par un grand losange avec un chemin partant vers chaque classe participante (cf. figure 3.6). Le nom de l'association, le cas échéant, apparaît à proximité du losange.

### 3.3.4 Multiplicité ou cardinalité

La multiplicité associée à une terminaison d'association, d'agrégation ou de composition déclare le nombre d'objets susceptibles d'occuper la position définie par la terminaison d'association. Voici quelques exemples de multiplicité :

- exactement un : 1 ou 1..1
- plusieurs : * ou 0..*
- au moins un : 1..*
- de un à six : 1..6

Dans une association binaire (cf. figure 3.5), la multiplicité sur la terminaison cible contraint le nombre d'objets de la classe cible pouvant être associés à un seul objet donné de la classe source (la classe de l'autre terminaison de l'association).

Dans une association n-aire, la multiplicité apparaissant sur le lien de chaque classe s'applique sur une instance de chacune des classes, à l'exclusion de la classe-association et de la classe considérée. Par exemple, si on prend une association ternaire entre les classes (A, B, C), la multiplicité de la terminaison C indique le nombre d'objets C qui peuvent apparaître dans l'association (définie section 3.3.7) avec une paire particulière d'objets A et B.

**Remarque 1 :**

Pour une association n-aire, la multiplicité minimale doit en principe, mais pas nécessairement, être 0. En effet, une multiplicité minimale de 1 (ou plus) sur une extrémité implique qu'il doit exister un lien (ou plus) pour TOUTES les combinaisons possibles des instances des classes situées aux autres extrémités de l'association n-aire !

**Remarque 2 :**

Il faut noter que, pour les habitués du modèle entité/relation, les multiplicités sont en UML « à l'envers » (par référence à Merise) pour les associations binaires et « à l'endroit » pour les n-aires avec $n>2$.

### 3.3.5 Navigabilité


Figure 3.7: Navigabilité.

La navigabilité indique s'il est possible de traverser une association. On représente graphiquement la navigabilité par une flèche du côté de la terminaison navigable et on empêche la navigabilité par une croix du côté de la terminaison non navigable (cf. figure 3.7). Par défaut, une association est navigable dans les deux sens.

Par exemple, sur la figure 3.7, la terminaison du côté de la classe *Commande* n'est pas navigable : cela signifie que les instances de la classe *Produit* ne stockent pas de liste d'objets du type *Commande*. Inversement, la terminaison du côté de la classe *Produit* est navigable : chaque objet commande contient une liste de produits.

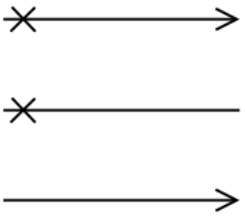


Figure 3.8: Implicitement, ces trois notations ont la même sémantique.

Lorsque l'on représente la navigabilité uniquement sur l'une des extrémités d'une association, il faut remarquer que, implicitement, les trois associations représentées sur la figure 3.8 ont la même signification : l'association ne peut être traversée que dans un sens.

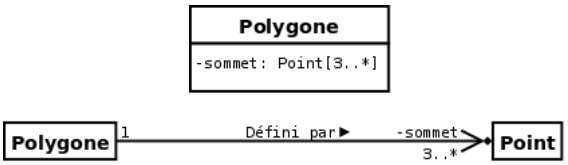



Figure 3.9: Deux modélisations équivalentes.

Dans la section 3.3.1, nous avons dit que :

> « Un attribut est une association dégénérée dans laquelle une terminaison d'association est détenue par un classeur (généralement une classe). Le classeur détenant cette terminaison d'association devrait théoriquement se trouver à l'autre terminaison, non modélisée, de l'association. Un attribut n'est donc rien d'autre qu'une terminaison d'un cas particulier d'association. »

La figure 3.9 illustre parfaitement cette situation. Attention toutefois, si vous avez une classe *Point* dans votre diagramme de classe, il est extrêmement maladroit de représenter des classes (comme la classe *Polygone*) avec un ou plusieurs attributs de type *Point*. Il faut, dans ce cas, matérialiser cette propriété de la classe en question par une ou plusieurs associations avec la classe *Point*.

### 3.3.6 Qualification

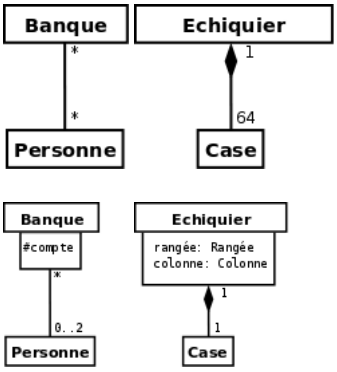



Figure 3.10: En haut, un diagramme représentant l'association entre une banque et ses clients (à gauche), et un diagramme représentant l'association entre un échiquier et les cases qui le composent (à droite). En bas, les diagrammes équivalents utilisant des associations qualifiées.

Généralement, une classe peut être décomposée en sous-classes ou posséder plusieurs propriétés. Une telle classe rassemble un ensemble d'éléments (d'objets). Quand une classe est liée à une autre classe par une association, il est parfois préférable de restreindre la portée de l'association à quelques éléments ciblés (comme un ou plusieurs attributs) de la classe. Ces éléments ciblés sont appelés un *qualificatif*. Un qualificatif permet donc de sélectionner un ou des objets dans le jeu des objets d'un objet (appelé *objet qualifié*) relié par une association à un autre objet. L'objet sélectionné par la valeur du qualificatif est appelé *objet cible*. L'association est appelée *association qualifiée*. Un qualificatif agit toujours sur une association dont la multiplicité est *plusieurs* (avant que l'association ne soit qualifiée) du côté *cible*.

Un objet qualifié et une valeur de qualificatif génèrent un objet cible lié unique. En considérant un objet qualifié, chaque valeur de qualificatif désigne un objet cible unique.

Par exemple, le diagramme de gauche de la figure 3.10 nous dit que :

- Un compte dans une banque appartient à au plus deux personnes. Autrement dit, une instance du couple {*Banque , compte*} est en association avec zéro à deux instances de la classe *Personne*.
- Mais une personne peut posséder plusieurs comptes dans plusieurs banques. C'est-à-dire qu'une instance de la classe *Personne* peut être associée à plusieurs (zéro compris) instances du couple {*Banque , compte*}.
- Bien entendu, et dans tous les cas, un instance du couple {*Personne , compte*} est en association avec une instance unique de la classe *Banque*.

Le diagramme de droite de cette même figure nous dit que :

- Une instance du triplet {*Echiquier, rangée, colonne*} est en association avec une instance unique de la classe *Case*.
- Inversement, une instance de la classe *Case* est en association avec une instance unique du triplet {*Echiquier, rangée, colonne*}.

### 3.3.7 Classe-association

#### Définition et représentation

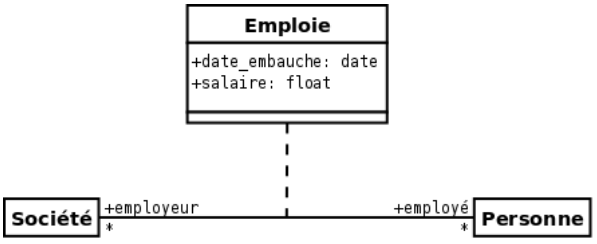



Figure 3.11: Exemple de classe-association.

Parfois, une association doit posséder des propriétés. Par exemple, l'association *Emploie* entre une société et une personne possède comme propriétés le salaire et la date d'embauche. En effet, ces deux propriétés n'appartiennent ni à la société, qui peut employer plusieurs personnes, ni aux personnes, qui peuvent avoir plusieurs emplois. Il s'agit donc bien de propriétés de l'association *Emploie*. Les associations ne pouvant posséder de propriété, il faut introduire un nouveau concept pour modéliser cette situation : celui de *classe-association*.

Une classe-association possède les caractéristiques des associations et des classes : elle se connecte à deux ou plusieurs classes et possède également des attributs et des opérations.

Une classe-association est caractérisée par un trait discontinu entre la classe et l'association qu'elle représente (figure 3.11).

#### Classe-association pour plusieurs associations

Il n'est pas possible de rattacher une classe-association à plus d'une association puisque la classe-association constitue elle-même l'association. Dans le cas où plusieurs classe-associations doivent disposer des mêmes caractéristiques, elles doivent hériter d'une même classe possédant ces caractéristiques, ou l'utiliser en tant qu'attribut.

De même, il n'est pas possible de rattacher une instance de la classe d'une classe-association à plusieurs instances de l'association de la classe-association. En effet, la représentation graphique (association reliée par une ligne pointillé à une classe déportée) est trompeuse : une classe-associaiton est une entité sémantique atomique et non composite qui s'intancie donc en bloc. Ce problème et à nouveau abordé et illustré section 3.5.2.

#### Auto-association sur classe-association

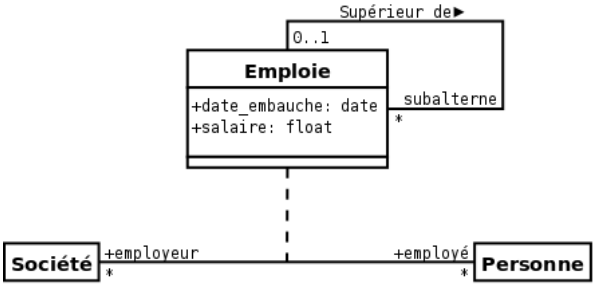



Figure 3.12: Exemple d'auto-association sur classe-association.

Imaginons que nous voulions ajouter une association *Supérieur de* dans le diagramme 3.11 pour préciser qu'une personne est le supérieur d'une autre personne. On ne peut simplement ajouter une association réflexive sur la classe *Personne*. En effet, une personne n'est pas le supérieur d'une autre dans l'absolu. Une personne est, en tant qu'employé d'une entreprise donné, le supérieur d'une autre personne dans le cadre de son emploi pour une entreprise donné (généralement, mais pas nécessairement, la même). Il s'agit donc d'une association réflexive, non pas sur la classe *Personne* mais sur la classe-association *Emploie* (cf. figure 3.12).

#### Liens multiples

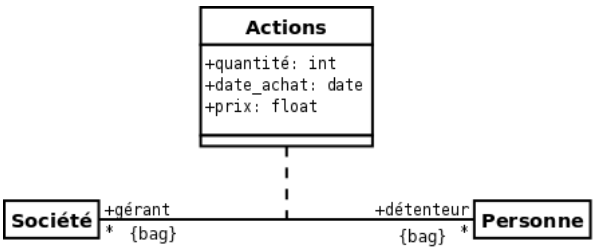



Figure 3.13: Exemple de classe-association avec liens multiples.

Plusieurs instances d'une même association ne peuvent lier les mêmes objets. Cependant, si l'on s'intéresse à l'exemple de la figure 3.13, on voit bien qu'il doit pouvoir y avoir plusieurs instances de la classe-association *Actions* liant une même personne à une même société : une même personne peut acheter à des moments différents des actions d'une même société.

C'est justement la raison de la contrainte *{bag}* qui, placée sur les terminaisons d'association de la classe-association *Actions*, indique qu'il peut y avoir des liens multiples impliquant les mêmes paires d'objets.

**Équivalences**

Parfois, l'information véhiculée par une classe-association peut être véhiculée sans perte d'une autre manière (cf. figure 3.14 et 3.15).

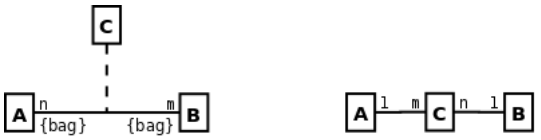



Figure 3.14: Deux modélisations modélisant la même information.

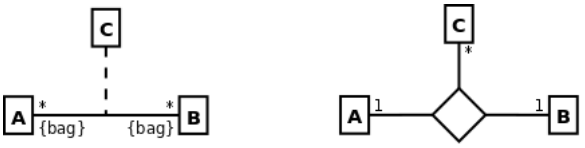



Figure 3.15: Deux modélisations modélisant la même information.

**Classe-association, association n-aire ou association qualifiée ?**

Il n'est souvent pas simple trancher entre l'utilisation d'une classe-association, d'une association n-aire ou encore d'une association qualifiée. Lorsque l'on utilise l'un de ces trois types d'association, il peut être utile ou instructif de se demander si l'un des deux autres types ne serait pas plus pertinent. Dans tous les cas, il faut garder à l'esprit qu'une classe-association est d'abord et avant tout une association et que, dans une classe-association, la classe est indissociable de l'association.

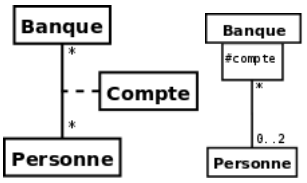



Figure 3.16: Pour couvrir le cas des comptes joints, il faut utiliser le modèle de droite.

Ainsi, le cas d'un compte joint ne peut être représenté correctement par le diagramme de gauche dans figure 3.16 : mieux vaut utiliser une association qualifiée (diagramme de droite dans la figure 3.16). Ce problème et à nouveau abordé et illustré section 3.5.2.

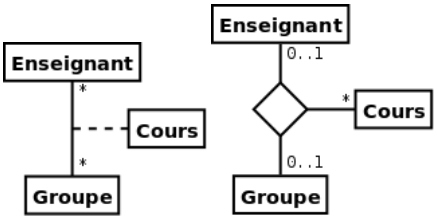



Figure 3.17: Si un cours doit pouvoir exister indépendamment d'un lien entre un enseignant et un groupe, il faut utiliser le modèle de droite.

Dans le diagramme de gauche de la figure 3.17, un cours ne peut exister que s'il existe un lien entre un objet *Enseignant* et un objet *Groupe*. Quand le lien est rompu (effacé), le cours l'est également. Si un cours doit pouvoir exister indépendamment de l'existence d'un lien (on a pas encore trouvé d'enseignant pour ce cours, le cours n'est pas enseigné cette année mais le sera probablement l'année prochaine, …) il faut opter pour une association ternaire (modèle de droite dans figure 3.17).

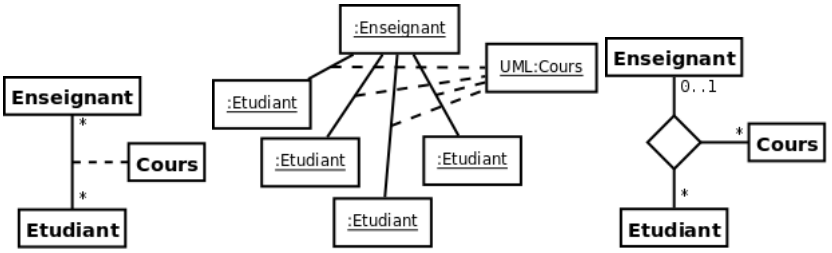



Figure 3.18: Si un même cours doit concerner plusieurs couples *Enseignant*/*Etudiant*, il ne faut pas utiliser une classe-association, mais une association ternaire comme sur le modèle de droite.

Le cas de figure est encore pire si l'on remplace *Groupe* par *Etudiant* (cf. modèle à gauche sur la figure 3.18). En effet, le cas général décrit par ce modèle ne correspond pas du tout au diagramme d'objet (cf. section 3.5) situé au centre. Nous reviendrons sur ce problème dans la section 3.5.2 avec l'illustration 3.24. Dans cette situation, il faut opter pour une association ternaire comme sur le modèle de droite.

### 3.3.8 Agrégation et composition

**Agrégation**


Figure 3.19: Exemple de relation d'agrégation et de composition.

Une association simple entre deux classes représente une relation structurelle entre pairs, c'est à dire entre deux classes de même niveau conceptuel : aucune des deux n'est plus importante que l'autre. Lorsque l'on souhaite modéliser une relation *tout/partie* où une classe constitue un élément plus grand (*tout*) composé d'éléments plus petit (*partie*), il faut utiliser une agrégation.

Une agrégation est une association qui représente une relation d'inclusion structurelle ou comportementale d'un élément dans un ensemble. Graphiquement, on ajoute un losange vide (♦) du côté de l'agrégat (cf. figure 3.19). Contrairement à une association simple, l'agrégation est transitive.

La signification de cette forme simple d'agrégation est uniquement conceptuelle. Elle ne contraint pas la navigabilité ou les multiplicités de l'association. Elle n'entraîne pas non plus de contrainte sur la durée de vie des parties par rapport au tout.

**Composition**

La composition, également appelée agrégation composite, décrit une contenance structurelle entre instances. Ainsi, la destruction de l'objet composite implique la destruction de ses composants. Une instance de la partie appartient toujours à au plus une instance de l'élément composite : la multiplicité du côté composite ne doit pas être supérieure à 1 (*i.e.* 1 ou 0..1). Graphiquement, on ajoute un losange plein (✦) du côté de l'agrégat (cf. figure 3.19).

**Remarque**

Les notions d'agrégation et surtout de composition posent de nombreux problèmes en modélisation et sont souvent le sujet de querelles d'experts et sources de confusions. Ce que dit la norme *UML Superstructure version 2.1.1* reflète d'ailleur très bien le flou qui entoure ces notions :

> *Precise semantics of shared aggregation varies by application area and modeler. The order and way in which part instances are created is not defined.*

### 3.3.9 Généralisation et Héritage

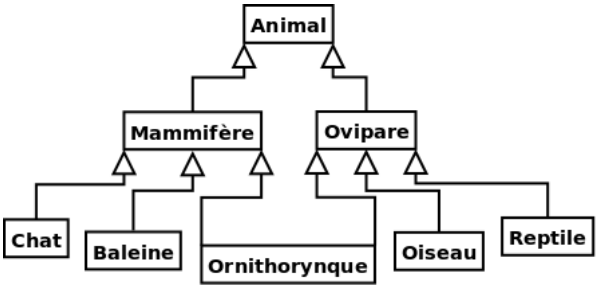



Figure 3.20: Partie du règne animal décrit avec l'héritage multiple.

La généralisation décrit une relation entre une classe générale (classe de base ou classe parent) et une classe spécialisée (sous-classe). La classe spécialisée est intégralement cohérente avec la classe de base, mais comporte des informations supplémentaires (attributs, opérations, associations). Un objet de la classe spécialisée peut être utilisé partout où un objet de la classe de base est autorisé.

Dans le langage UML, ainsi que dans la plupart des langages objet, cette relation de généralisation se traduit par le concept d'héritage. On parle également de relation d'héritage. Ainsi, l'héritage permet la classification des objets (cf. figure 3.20).

Le symbole utilisé pour la relation d'héritage ou de généralisation est une flèche avec un trait plein dont la pointe est un triangle fermé désignant le cas le plus général (cf. figure 3.20).

Les propriétés principales de l'héritage sont :

- La classe enfant possède toutes les caractéristiques des ses classes parents, mais elle ne peut accéder aux caractéristiques privées de cette dernière.

- Une classe enfant peut redéfinir (même signature) une ou plusieurs méthodes de la classe parent. Sauf indication contraire, un objet utilise les opérations les plus spécialisées dans la hiérarchie des classes.
- Toutes les associations de la classe parent s'appliquent aux classes dérivées.
- Une instance d'une classe peut être utilisée partout où une instance de sa classe parent est attendue. Par exemple, en se basant sur le diagramme de la figure 3.20, toute opération acceptant un objet d'une classe *Animal* doit accepter un objet de la classe *Chat*.
- Une classe peut avoir plusieurs parents, on parle alors d'héritage multiple (cf. la classe *Ornithorynque* de la figure 3.20). Le langage C++ est un des langages objet permettant son implémentation effective, le langage Java ne le permet pas.

En UML, la relation d'héritage n'est pas propre aux classes. Elle s'applique à d'autre éléments du langage comme les paquetages, les acteurs (cf. section 2.3.3) ou les cas d'utilisation (cf. section 2.3.2).

### 3.3.10 Dépendance

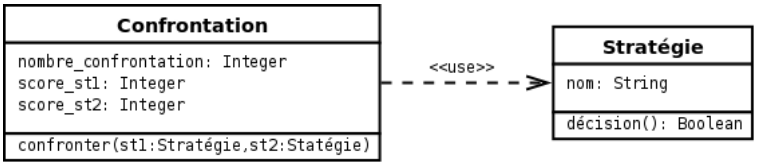



Figure 3.21: Exemple de relation de dépendance.

Une dépendance est une relation unidirectionnelle exprimant une dépendance sémantique entre des éléments du modèle. Elle est représentée par un trait discontinu orienté (cf. figure 3.21). Elle indique que la modification de la cible peut impliquer une modification de la source. La dépendance est souvent stéréotypée pour mieux expliciter le lien sémantique entre les éléments du modèle (cf. figure 3.21 ou 3.25).

On utilise souvent une dépendance quand une classe en utilise une autre comme argument dans la signature d'une opération. Par exemple, le diagramme de la figure 3.21 montre que la classe *Confrontation* utilise la classe *Stratégie* car la classe *Confrontation* possède une méthode *confronter* dont deux paramètre sont du type *Stratégie*. Si la classe *Stratégie*, notamment son interface, change, alors des modifications devront également être apportées à la classe *Confrontation*.

[4] Une terminaison d'associations est une extrémité de l'association. Une association binaire en possède deux, une association n-aire en possède *n*.

## 3.4 Interfaces

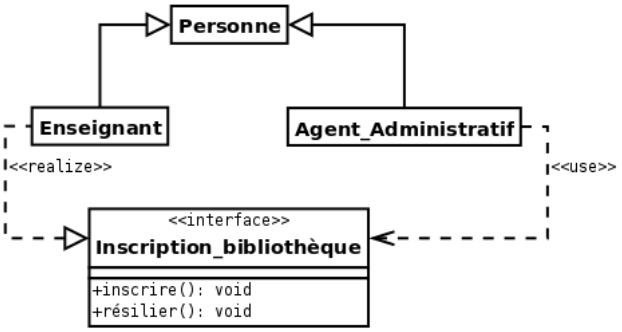



Figure 3.22: Exemple de diagramme mettant en œuvre une interface

Nous avons déjà abordé la notion d'interface dans les sections 1.3.4 et 3.2.4. En effet, les classes permettent de définir en même temps un objet et son interface. Le classeur, que nous décrivons dans cette section, ne permet de définir que des éléments d'interface. Il peut s'agir de l'interface complète d'un objet, ou simplement d'une partie d'interface qui sera commune à plusieurs objets.

Le rôle de ce classeur, stéréotypé << *interface* >>, est de regrouper un ensemble de propriétés et d'opérations assurant un service cohérent. L'objectif est de diminuer le couplage entre deux classeurs. La notion d'interface en UML est très proche de la notion d'interface en Java.

Une interface est représentée comme une classe excepté l'absence du mot-clef *abstract* (car l'interface et toutes ses méthodes sont, par définition, abstraites) et l'ajout du stéréotype << *interface* >> (cf. figure 3.22).

Une interface doit être réalisée par au moins une classe et peut l'être par plusieurs. Graphiquement, cela est représenté par un trait discontinu terminé par une flèche triangulaire et le stéréotype *« realize »*. Une classe peut très bien réaliser plusieurs interfaces. Une classe (classe cliente de l'interface) peut dépendre d'une interface (interface requise). On représente cela par une relation de dépendance et le stéréotype *« use »*. Attention aux problèmes de conflits si une classe dépend d'une interface réalisée par plusieurs autres classes.

La notion d'interface est assez proche de la notion de classe abstraite avec une capacité de découplage plus grand. En C++ (le C++ ne connaît pas la notion d'interface), la notion d'interface est généralement implémentée par une classe abstraite.

## 3.5 Diagramme d'objets (*object diagram*)

### 3.5.1 Présentation

Un diagramme d'objets représente des objets (*i.e.* instances de classes) et leurs liens (*i.e.* instances de relations) pour donner une vue figée de l'état d'un système à un instant donné. Un diagramme d'objets peut être utilisé pour :

- illustrer le modèle de classes en montrant un exemple qui explique le modèle ;

- préciser certains aspects du système en mettant en évidence des détails imperceptibles dans le diagramme de classes ;
- exprimer une exception en modélisant des cas particuliers ou des connaissances non généralisables qui ne sont pas modélisés dans un diagramme de classe ;
- prendre une image (*snapshot*) d'un système à un moment donné.

Le diagramme de classes modélise les règles et le diagramme d'objets modélise des faits.

Par exemple, le diagramme de classes de la figure 3.23 montre qu'une entreprise emploie au moins deux personnes et qu'une personne travaille dans au plus deux entreprises. Le diagramme d'objets modélise lui une entreprise particulière (*PERTNE*) qui emploie trois personnes.

Un diagramme d'objets ne montre pas l'évolution du système dans le temps. Pour représenter une interaction, il faut utiliser un diagramme de communication (cf. section 7.2) ou de séquence (cf. section 7.3).

### 3.5.2 Représentation

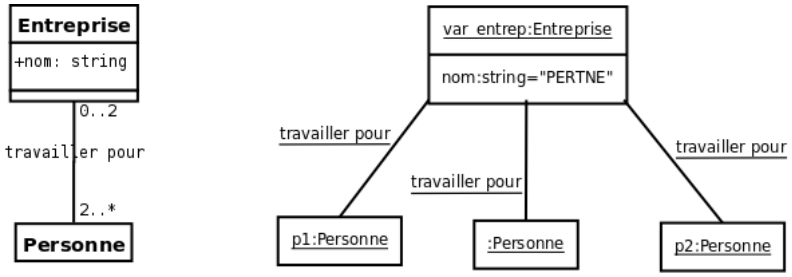



Figure 3.23: Exemple de diagramme de classes et de diagramme d'objets associé.

Graphiquement, un objet se représente comme une classe. Cependant, le compartiment des opérations n'est pas utile. De plus, le nom de la classe dont l'objet est une instance est précédé d'un << : >> et est souligné. Pour différencier les objets d'une même classe, leur identifiant peut être ajouté devant le nom de la classe. Enfin les attributs reçoivent des valeurs. Quand certaines valeurs d'attribut d'un objet ne sont pas renseignées, on dit que l'objet est partiellement défini.

Dans un diagrammes d'objets, les relations du diagramme de classes deviennent des liens. La relation de généralisation ne possède pas d'instance, elle n'est donc jamais représentée dans un diagramme d'objets. Graphiquement, un lien se représente comme une relation, mais, s'il y a un nom, il est souligné. Naturellement, on ne représente pas les multiplicités.

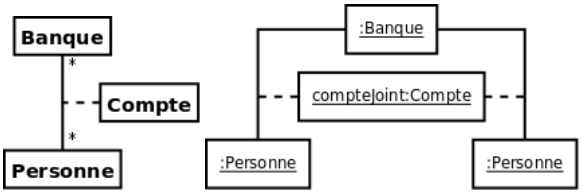



Figure 3.24: Le diagramme d'objets de droite, illustrant le cas de figure d'un compte joint, n'est pas une instance *normale* du diagramme de classe de gauche mais peut préciser une situation exceptionnelle.

La norme UML 2.1.1 précise qu'une instance de classe-association ne peut être associée qu'à une instance de chacunes des classes associées[5] ce qui interdit d'instancier le diagramme de classe à gauche dans la figure 3.24 par le diagramme d'objet à droite dans cette même figure. Cependant, un diagramme d'objet peut être utilisé pour exprimer une exception. Sur la figure 3.24, le diagramme d'objets à droite peut être légitime s'il vient préciser une situation exceptionnelle non prise en compte par le diagramme de classe représenté à gauche. Néanmoins, le cas des comptes joints n'étant pas si exceptionnel, mieux vaut revoir la modélisation comme préconisé par la figure 3.16.

### 3.5.3 Relation de dépendance d'instanciation

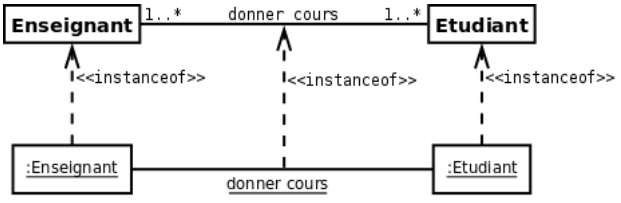



Figure 3.25: Dépendance d'instanciation entre les classeurs et leurs instances.

La relation de dépendance d'instanciation (stéréotypée << *instanceof* >>) décrit la relation entre un classeur et ses instances. Elle relie, en particulier, les liens aux associations et les objets aux classes.

[5] UML Superstructure Specification, v2.1.1 ; p.49 : << *It should be noted that in an instance of an association class, there is only one instance of the associated classifiers at each end, i.e., from the instance point of view, the multiplicity of the associations ends are '1'* >>

## 3.6 Élaboration et implémentation d'un diagramme de classes

### 3.6.1 Élaboration d'un diagramme de classes

Une démarche couramment utilisée pour bâtir un diagramme de classes consiste à :

**Trouver les classes du domaine étudié.**
Cette étape empirique se fait généralement en collaboration avec un expert du domaine. Les classes correspondent généralement à des concepts ou des substantifs du domaine.

**Trouver les associations entre classes.**
Les associations correspondent souvent à des verbes, ou des constructions verbales, mettant en relation plusieurs classes, comme << est composé de >>, << pilote >>, << travaille pour >>. *Attention, méfiez vous de certains attributs qui sont en réalité des relations entre classes.*

**Trouver les attributs des classes.**
Les attributs correspondent souvent à des substantifs, ou des groupes nominaux, tels que << la masse d'une voiture >> ou << le montant d'une transaction >>. Les adjectifs et les valeurs correspondent souvent à des valeurs d'attributs. Vous pouvez ajouter des attributs à toutes les étapes du cycle de vie d'un projet (implémentation comprise). N'espérez pas trouver tous les attributs dès la construction du diagramme de classes.

**Organiser et simplifier le modèle**
en éliminant les classes redondantes et en utilisant l'héritage.

**Itérer et raffiner le modèle.**
Un modèle est rarement correct dès sa première construction. La modélisation objet est un processus non pas linéaire mais itératif.

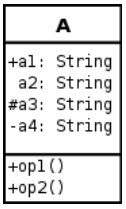
### 3.6.2 Implémentation en Java

#### Classe

Parfois, la génération automatique de code produit, pour chaque classe, un constructeur et une méthode finalize comme ci-dessous. Rappelons que cette méthode est invoquée par le *ramasse miettes* lorsque celui-ci constate que l'objet n'est plus référencé. Pour des raisons de simplification, nous ne ferons plus figurer ces opérations dans les sections suivantes.

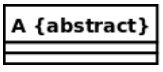

```
public class A {
  public A() {
    ...
  }
  protected void finalize() throws Throwable {
    super.finalize();
    ...
  }
}
```

#### Classe avec attributs et opérations



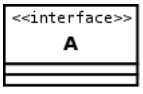

```
public class A {
  public    String a1;
  package   String a2;
  protected String a3;
  private   String a4;
  public void op1() {
    ...
  }
  public void op2() {
    ...
  }
}
```

#### Classe abstraite



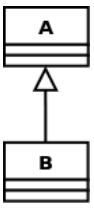

```
public abstract class A {
  ...
}
```

#### Interface




```
public interface A {
  ...
}
```

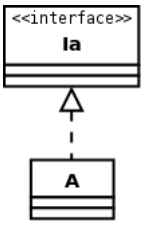
#### Héritage simple




```
public class A {
  ...
}

public class B extends A {
  ...
}
```

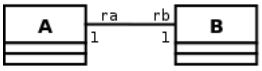
**Réalisation d'une interface par une classe**




```
public interface Ia {
  ...
}

public class A implements Ia {
  ...
}
```

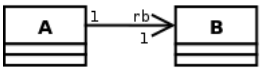
**Association bidirectionnelle 1 vers 1**




```
public class A {
  private B rb;
  public void addB( B b ) {
    if( b != null ){
      if ( b.getA() != null ) {   // si b est déjà connecté à un autre A
        b.getA().setB(null);    // cet autre A doit se déconnecter
      }
      this.setB( b );
      b.setA( this );
    }
  }
  public B getB() { return( rb ); }
  public void setB( B b ) { this.rb=b; }
}

public class B {
  private A ra;
  public void addA( A a ) {
    if( a != null ) {
      if (a.getB() != null) {   // si a est déjà connecté à un autre B
        a.getB().setA( null );  // cet autre B doit se déconnecter
      }
      this.setA( a );
      a.setB( this );
    }
  }
  public void setA(A a){ this.ra=a; }
  public A getA(){ return(ra); }
}
```

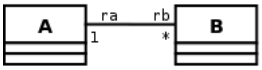
**Association unidirectionnelle 1 vers 1**




```
public class A {
  private B rb;
  public void addB( B b ) {
    if( b != null ) {
      this.rb=b;
    }
  }
}

public class B {
 ... // La classe B ne connaît pas l'existence de la classe A
}
```

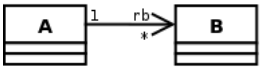
**Association bidirectionnelle 1 vers N**




```
public class A {
  private ArrayList <B> rb;
  public A() { rb = new  ArrayList<B>(); }
  public ArrayList <B> getArray() {return(rb);}
  public void remove(B b){rb.remove(b);}
  public void addB(B b){
    if( !rb.contains(b) ){
      if (b.getA()!=null) b.getA().remove(b);
      b.setA(this);
      rb.add(b);
    }
  }
}

public class B {
  private  A ra;
  public B() {}
  public A getA() { return (ra); }
  public void setA(A a){ this.ra=a; }
  public void addA(A a){
    if( a != null ) {
      if( !a.getArray().contains(this)) {
        if (ra != null) ra.remove(this);
        this.setA(a);
        ra.getArray().add(this);
      }
    }
  }
}
```

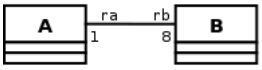
**Association unidirectionnelle 1 vers plusieurs**




```
public class A {
  private ArrayList <B> rb;
  public A() { rb = new  ArrayList<B>(); }
  public void addB(B b){
    if( !rb.contains( b ) ) {
      rb.add(b);
    }
  }
}

public class B {
 ... // B ne connaît pas l'existence de A
}
```

**Association 1 vers N**

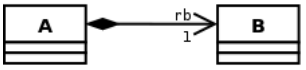



Dans ce cas, il faut utiliser un tableau plutôt qu'un vecteur. La dimension du tableau étant données par la cardinalité de la terminaison d'association.

**Agrégations**


Les agrégations s'implémentent comme les associations.

**Composition**




Une composition peut s'implémenter comme une association unidirectionnelle.

### 3.6.3 Implémentation en SQL

**Introduction**

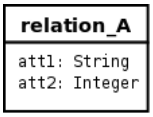
Il est possible de traduire un diagramme de classe en modèle relationnel. Bien entendu, les méthodes des classes ne sont pas traduites. Aujourd'hui, lors de la conception de base de données, il devient de plus en plus courant d'utiliser la modélisation UML plutôt que le traditionnel modèle entités-associations.

Cependant, à moins d'avoir respecté une méthodologie adaptée, la correspondance entre le modèle objet et le modèle relationnel n'est pas une tâche facile. En effet, elle ne peut que rarement être complète puisque l'expressivité d'un diagramme de classes est bien plus grande que celle d'un schéma relationnel. Par exemple, comment représenter dans un schéma relationnel des notions comme la navigabilité ou la composition ? Toutefois, de nombreux AGL (Atelier de Génie Logiciel) comportent maintenant des fonctionnalités de traduction en SQL qui peuvent aider le développeur dans cette tâche.

Dans la section 9.3.1, nous présentons un type de diagramme de classes, appelé *modèle du domaine*, tout à fait adapté à une implémentation sous forme de base de données.

**Classe avec attributs**

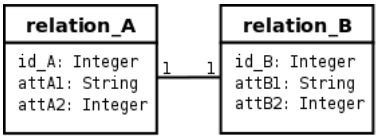
Chaque classe devient une relation. Les attributs de la classe deviennent des attributs de la relation. Si la classe possède un identifiant, il devient la clé primaire de la relation, sinon, il faut ajouter une clé primaire arbitraire.




```
create table relation_A (
    num_relation_A integer primary key,
    att1 text,
    att2 integer);
```

**Association 1 vers 1**

Pour représenter une association 1 vers 1 entre deux relation, la clé primaire de l'une des relations doit figurer comme clé étrangère dans l'autre relation.

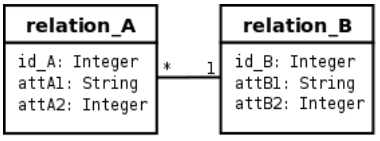



```
create table relation_A (
    id_A integer primary key,
    attA1 text,
    attA2 integer);

create table relation_B (
    id_B integer primary key,
    num_A integer references relation_A,
    attB1 text,
    attB2 integer);
```

**Association 1 vers plusieurs**

Pour représenter une association 1 vers plusieurs, on procède comme pour une association 1 vers 1, excepté que c'est forcément la relation du côté plusieurs qui reçoit comme clé étrangère la clé primaire de la relation du côté 1.

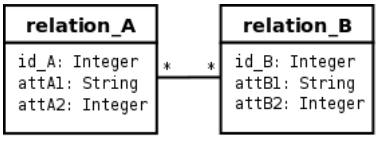



```
create table relation_A (
    id_A integer primary key,
    num_B integer references relation_B,
    attA1 text,
    attA2 integer);

create table relation_B (
    id_B integer primary key,
    attB1 text,
    attB2 integer);
```

**Association plusieurs vers plusieurs**

Pour représenter une association du type plusieurs vers plusieurs, il faut introduire une nouvelle relation dont les attributs sont les clés primaires des relations en association et dont la clé primaire est la concaténation de ces deux attributs.



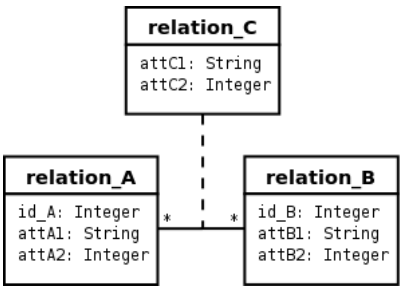

```
create table relation_A (
    id_A integer primary key,
    attA1 text,
    attA2 integer);

create table relation_B (
    id_B integer primary key,
    attB1 text,
    attB2 integer);

create table relation_A_B (
    num_A integer references relation_A,
    num_B integer references relation_B,
    primary key (num_A, num_B));
```

**Classe-association plusieurs vers plusieurs**

Le cas est proche de celui d'une association plusieurs vers plusieurs, les attributs de la classe-association étant ajoutés à la troisième relation qui représente, cette fois ci, la classe-association elle-même.



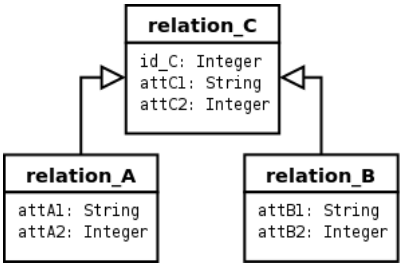

```
create table relation_A (
    id_A integer primary key,
    attA1 text,
    attA2 integer);

create table relation_B (
    id_B integer primary key,
    attB1 text,
    attB2 integer);

create table relation_C (
    num_A integer references relation_A,
    num_B integer references relation_B,
    attC1 text,
    attC2 integer,
    primary key (num_A, num_B));
```

**Héritage**

Les relations correspondant aux sous-classes ont comme clé étrangère et primaire la clé de la relation correspondant à la classe parente. Un attribut type est ajouté dans la relation correspondant à la classe parente. Cet attribut permet de savoir si les informations d'un tuple de la relation correspondant à la classe parente peuvent être complétées par un tuple de l'une des relations correspondant à une sous-classe, et, le cas échéant, de quelle relation il s'agit. Ainsi, dans cette solution, un objet peut avoir ses attributs répartis dans plusieurs relations. Il faut donc opérer des jointures pour reconstituer un objet. L'attribut type de la relation correspondant à la classe parente doit indiquer quelles jointures faire.




```
create table relation_C (
    id_C integer primary key,
    attC1 text,
    attC2 integer,
    type text);

create table relation_A (
    id_A references relation_C,
    attA1 text,
    attA2 integer,
    primary key (id_A));

create table relation_B (
    id_B references relation_C,
    attB1 text,
    attB2 integer,
    primary key (id_B));
```

Une alternative à cette représentation est de ne créer qu'une seule table par arborescence d'héritage. Cette table doit contenir tous les attributs de toutes les classes de l'arborescence plus l'attribut type dont nous avons parlé ci-dessus. L'inconvénient de cette solution est qu'elle implique que les tuples contiennent de nombreuses valeurs nulles.

```
create table relation_ABC (
    id_C integer primary key,
    attC1 text, attC2 integer,
    attA1 text, attA2 integer,
    attB1 text, attB2 integer,
    type text);
```

# Chapitre 4 Langage de contraintes objet *(Object Constraint Langage : OCL)*

## 4.1 Expression des contraintes en UML

### 4.1.1 Introduction

Nous avons déjà vu comment exprimer certaines formes de contraintes avec UML :

**Contraintes structurelles :**
les attributs dans les classes, les différents types de relations entre classes (généralisation, association, agrégation, composition, dépendance), la cardinalité et la navigabilité des propriétés structurelles, etc.

**Contraintes de type :**
typage des propriétés, etc.

**Contraintes diverses :**
les contraintes de visibilité, les méthodes et classes abstraites (contrainte *abstract*), etc.

Dans la pratique, toutes ces contraintes sont très utiles mais se révèlent insuffisantes. Toutefois, UML permet de spécifier explicitement des contraintes particulières sur des éléments de modèle.

### 4.1.2 Écriture des contraintes

Une contrainte constitue une condition ou une restriction sémantique exprimée sous forme d'instruction dans un langage textuel qui peut être naturel ou formel. En général, une contrainte peut être attachée à n'importe quel élément de modèle ou liste d'éléments de modèle. Une contrainte désigne une restriction qui doit être appliquée par une implémentation correcte du système.

On représente une contrainte sous la forme d'une chaîne de texte placée entre accolades ({}). La chaîne constitue le corps écrit dans un langage de contrainte qui peut être :

- naturel ;
- dédié, comme OCL ;
- ou encore directement issu d'un langage de programmation.

Si une contrainte possède un nom, on présente celui-ci sous forme d'une chaîne suivie d'un double point (:), le tout précédant le texte de la contrainte.

### 4.1.3 Représentation des contraintes et contraintes prédéfinies

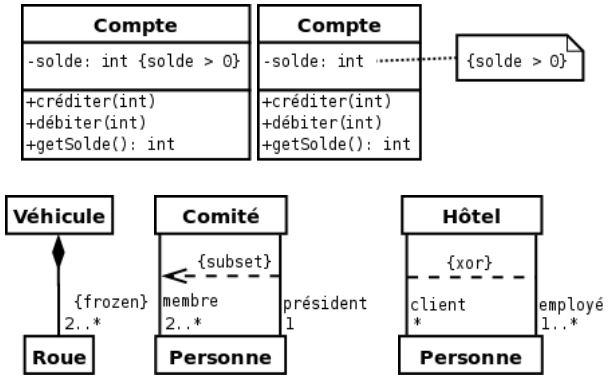



Figure 4.1: UML permet d'associer une contrainte à un élément de modèle de plusieurs façons. Sur les deux diagrammes du haut, la contrainte porte sur un attribut qui doit être positif. En bas à gauche, la contrainte *{frozen}* précise que le nombre de roues d'un véhicule ne peut pas varier. Au milieu, la contrainte *{subset}* précise que le président est également un membre du comité. Enfin, en bas à droite, la contrainte *{xor}* (*ou* exclusif) précise que les employés de l'hôtel n'ont pas le droit de prendre une chambre dans ce même hôtel.


Figure 4.2: Ce diagramme exprime que : une personne est née dans un pays, et que cette association ne peut être modifiée ; une personne a visité un certain nombre de pays, dans un ordre donné, et que le nombre de pays visités ne peut que croître ; une personne aimerait encore visiter tout une liste de pays, et que cette liste est ordonnée (probablement par ordre de préférence).

UML permet d'associer une contrainte à un, ou plusieurs, élément(s) de modèle de différentes façons (cf. figure 4.1) :

- en plaçant directement la contrainte à côté d'une propriété ou d'une opération dans un classeur ;
- en ajoutant une note associée à l'élément à contraindre ;
- en plaçant la contrainte à proximité de l'élément à contraindre, comme une extrémité d'association par exemple ;
- en plaçant la contrainte sur une flèche en pointillés joignant les deux éléments de modèle à contraindre ensemble, la direction de la flèche constituant une information pertinente au sein de la contrainte ;
- en plaçant la contrainte sur un trait en pointillés joignant les deux éléments de modèle à contraindre ensemble dans le cas où la contrainte est bijective ;
- en utilisant une note reliée, par des traits en pointillés, à chacun des éléments de modèle, subissant la contrainte commune, quand cette contrainte s'applique sur plus de deux éléments de modèle.

Nous venons de voir, au travers des exemples de la figure 4.1, quelques contraintes prédéfinies (*{frozen}*, *{subset}* et *{xor}*). Le diagramme de la figure 4.2 en introduit deux nouvelles : *{ordered}* et *{addOnly}*. La liste est encore longue, mais le pouvoir expressif de ces contraintes reste insuffisant comme nous le verrons dans la section 4.2.2. Le langage de contraintes objet OCL apporte une solution élégante à cette insuffisance.

## 4.2 Intérêt d'un langage de contraintes objet comme OCL

### 4.2.1 OCL – Introduction

**QuesacOCL ?**

C'est avec OCL (*Object Constraint Language*) qu'UML formalise l'expression des contraintes . Il s'agit donc d'un langage formel d'expression de contraintes bien adapté aux diagrammes d'UML, et en particulier au diagramme de classes.

OCL existe depuis la version 1.1 d'UML et est une contribution d'IBM. OCL fait partie intégrante de la norme UML depuis la version 1.3 d'UML. Dans le cadre d'UML 2.0, les spécifications du langage OCL figurent dans un document indépendant de la norme d'UML, décrivant en détail la syntaxe formelle et la façon d'utiliser ce langage.

OCL peut s'appliquer sur la plupart des diagrammes d'UML et permet de spécifier des contraintes sur l'état d'un objet ou d'un ensemble d'objets comme :

- des invariants sur des classes ;
- des préconditions et des postconditions à l'exécution d'opérations :
  - les préconditions doivent être vérifiées avant l'exécution,
  - les postconditions doivent être vérifiées après l'exécution ;
- des gardes sur des transitions de diagrammes d'états-transitions ou des messages de diagrammes d'interaction ;
- des ensembles d'objets destinataires pour un envoi de message ;
- des attributs dérivés, etc.

**Pourquoi OCL ?**

Nous avons dit que les contraintes pouvaient être écrites en langage naturel, alors pourquoi s'embarrasser du langage OCL ? L'intérêt du langage naturel est qu'il est simple à mettre en œuvre et compréhensible par tous. Par contre (et comme toujours), il est ambigu et imprécis, il rend difficile l'expression des contraintes complexes et ne facilite pas les références à d'autres éléments (autres que celui sur lequel porte la contrainte) du modèle.

OCL est un langage formel volontairement simple d'accès. Il possède une grammaire élémentaire (OCL peut être interprété par des outils) que nous décrirons dans les sections 4.3 à 4.6. OCL représente, en fait, un juste milieu entre le langage naturel et un langage très technique (langage mathématique, informatique, …). Il permet ainsi de limiter les ambiguïtés, tout en restant accessible.

### 4.2.2 Illustration par l'exemple

**Mise en situation**

Plaçons-nous dans le contexte d'une application bancaire. Il nous faut donc gérer :

- des comptes bancaires,
- des clients,
- et des banques.

De plus, on aimerait intégrer les contraintes suivantes dans notre modèle :

- un compte doit avoir un solde toujours positif ;
- un client peut posséder plusieurs comptes ;
- une personne peut être cliente de plusieurs banques ;
- un client d'une banque possède au moins un compte dans cette banque ;
- un compte appartient forcément à un client ;
- une banque gère plusieurs comptes ;
- une banque possède plusieurs clients.

**Diagramme de classes**

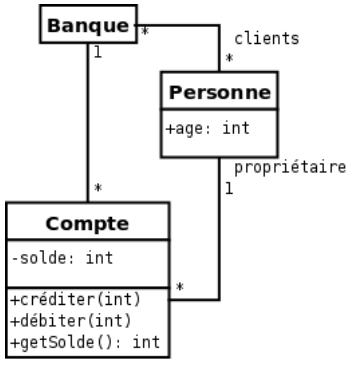



Figure 4.3: Diagramme de classes modélisant une banque, ses clients et leurs comptes.

La figure 4.3 montre un diagramme de classes correspondant à la problématique que nous venons de décrire.

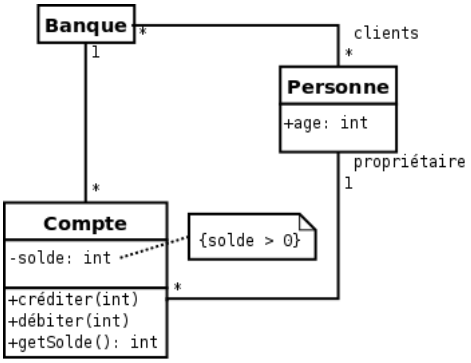



Figure 4.4: Ajout d'une contrainte sur le diagramme de la figure 4.3.

Un premier problème apparaît immédiatement : rien ne spécifie, dans ce diagramme, que le solde du client doit toujours être positif. Pour résoudre le problème, on peut simplement ajouter une note précisant cette contrainte (*{solde > 0}*), comme le montre la figure 4.4.

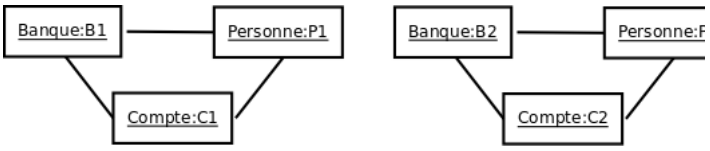



Figure 4.5: Diagramme d'objets cohérent avec le diagramme de classes de la figure 4.4.

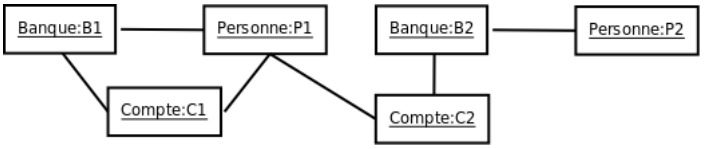
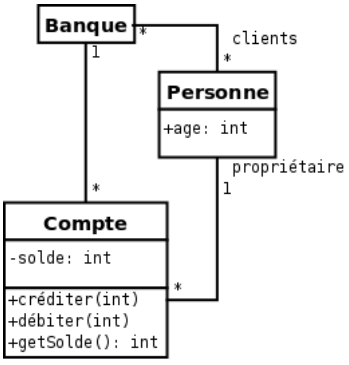



Figure 4.6: Diagramme d'objets cohérent avec le diagramme de classes de la figure 4.4, mais représentant une situation inacceptable.

Cependant, d'autres problèmes subsistent. La figure 4.5 montre un diagramme d'objets valide vis-à-vis du diagramme de classes de la figure 4.4 et également valide vis-à-vis de la spécification du problème. Par contre, la figure 4.6 montre un diagramme d'objets valide vis-à-vis du diagramme de classes de la figure 4.4 mais ne respectant pas la spécification du problème. En effet, ce diagramme d'objets montre une personne (`P1`) ayant un compte dans une banque sans en être client. Ce diagramme montre également un client (`P2`) d'une banque n'y possédant pas de compte.




**context** Compte
**inv** : solde > 0

**context** Compte : : débiter(somme : int)
**pre** : somme > 0
**post** : solde = solde@pre - somme

**context** Compte
**inv** : banque.clients -> includes (propriétaire)

Figure 4.7: Exemple d'utilisation du langage de contrainte OCL sur l'exemple bancaire.

Le langage OCL est particulièrement adapté à la spécification de ce type de contrainte. La figure 4.7 montre le diagramme de classes de notre application bancaire accompagné des contraintes OCL adaptées à la spécification du problème.

**Remarque :**

faites bien attention au fait qu'une expression OCL décrit une contrainte à respecter et ne décrit absolument pas l'implémentation d'une méthode.

## 4.3 Typologie des contraintes OCL

### 4.3.1 Diagramme support des exemples illustratifs

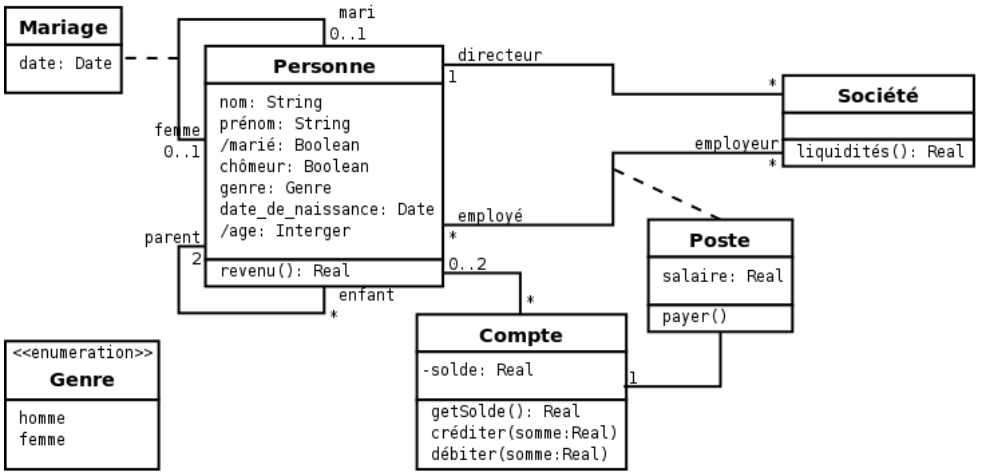



Figure 4.8: Diagramme de classes modélisant une entreprise et des personnes.

Le diagramme de la figure 4.8 modélise des personnes, leurs liens de parenté (enfant/parent et mari/femme) et le poste éventuel de ces personnes dans une société. Ce diagramme nous servira de support aux différents exemples de contraintes que nous donnerons, à titre d'illustration, dans les sections qui suivent (4.3 à 4.7).


Figure 4.9: Définition d'une énumération en utilisant un classeur.

Ce diagramme introduit un nouveau type de classeur, stéréotypé *«enumeration»*, permettant de définir une énumération. Une énumération est un type de donné UML, possédant un nom, et utilisé pour énumérer un ensemble de littéraux correspondant à toutes les valeurs possibles que peut prendre une expression de ce type. Un type énuméré est défini par un classeur possédant le stéréotype *«enumeration»* comme représenté sur la figure 4.9.

### 4.3.2 Contexte (*context*)

Une contrainte est toujours associée à un élément de modèle. C'est cet élément qui constitue le contexte de la contrainte. Il existe deux manières pour spécifier le contexte d'une contrainte OCL :

- En écrivant la contrainte entre accolades ({}) dans une note (comme nous l'avons fait sur la figure 4.4). L'élément pointé par la note est alors le contexte de la contrainte.
- En utilisant le mot-clef `context` dans un document accompagnant le diagramme (comme nous l'avons fait sur la figure 4.7).

**Syntaxe**

```
context <élément>
```

`<élément>` peut être une classe, une opération, etc. Pour faire référence à un élément *op* (comme un opération) d'un classeur *C* (comme une classe), ou d'un paquetage, …, il faut utiliser les `::` comme séparateur (comme *C*`::`*op*).

**Exemple**

Le contexte est la classe *Compte*:

**context** Compte

Le contexte est l'opération *getSolde()* de la classe *Compte*:

**context** Compte::getSolde()

### 4.3.3 Invariants (*inv*)

Un invariant exprime une contrainte prédicative sur un objet, ou un groupe d'objets, qui doit être respectée en permanence.

**Syntaxe**

```
inv : <expression_logique>
```

`<expression_logique>` est une expression logique qui doit toujours être vraie.

**Exemple**

Le solde d'un compte doit toujours être positif.

**context** Compte

**inv :** solde > 0

Les femmes (au sens de l'association) des personnes doivent être des femmes (au sens du genre).

**context** Personne
**inv :** femme->forAll(genre=Genre::femme)

*self* est décrit section 4.5.1 et *forAll()* section 4.6.3.

### 4.3.4 Préconditions et postconditions (*pre*, *post*)

Une précondition (respectivement une postcondition) permet de spécifier une contrainte prédicative qui doit être vérifiée avant (respectivement après) l'appel d'une opération.

Dans l'expression de la contrainte de la postcondition, deux éléments particuliers sont utilisables :

- l'attribut `result` qui désigne la valeur retournée par l'opération,
- et `<nom_attribut>@pre` qui désigne la valeur de l'attribut `<nom_attribut>` avant l'appel de l'opération.

**Syntaxe**

- Précondition :

```
pre : <expression_logique>
```

- Postcondition :

```
post : <expression_logique>
```

`<expression_logique>` est une expression logique qui doit toujours être vraie.

**Exemple**

Concernant la méthode *débiter* de la classe *Compte*, la somme à débiter doit être positive pour que l'appel de l'opération soit valide et, après l'exécution de l'opération, l'attribut *solde* doit avoir pour valeur la différence de sa valeur avant l'appel et de la somme passée en paramètre.

**context** Compte::débiter(somme : Real)
**pre :** somme > 0
**post :** solde = solde@**pre** - somme

Le résultat de l'appel de l'opération *getSolde* doit être égal à l'attribut *solde*.

**context** Compte::getSolde() : Real
**post : result** = solde

**Attention :**

même si cela peut sembler être le cas dans ces exemples, nous n'avons pas décrit comment l'opération est réalisée, mais seulement les contraintes sur l'état avant et après son exécution.

### 4.3.5 Résultat d'une méthode (*body*)

Ce type de contrainte permet de définir directement le résultat d'une opération.

**Syntaxe**

```
body : <requête>
```

`<requête>` est une expression qui retourne un résultat dont le type doit être compatible avec le type du résultat de l'opération désignée par le contexte.

**Exemple**

Voici une autre solution au deuxième exemple de la section 4.3.4 : le résultat de l'appel de l'opération *getSolde* doit être égal à l'attribut *solde*.

**context** Compte::getSolde() : Real
**body :** solde

### 4.3.6 Définition d'attributs et de méthodes (*def* et *let…in*)

Parfois, une sous-expression est utilisée plusieurs fois dans une expression. *let* permet de déclarer et de définir la valeur (*i.e.* initialiser) d'un attribut qui pourra être utilisé dans l'expression qui suit le *in*.

*def* est un type de contrainte qui permet de déclarer et de définir la valeur d'attributs comme la séquence *let…in*. *def* permet également de déclarer et de définir la valeur retournée par une opération interne à la contrainte.

**Syntaxe de *let…in***

```
let <déclaration> = <requête> in <expression>
```

Un nouvel attribut déclaré dans `<déclaration>` aura la valeur retournée par l'expression `<requête>` dans toute l'expression `<expression>`.

Reportez-vous à la section 4.7 pour un exemple d'utilisation.

**Syntaxe de *def***

```
def : <déclaration> = <requête>
```

`<déclaration>` peut correspondre à la déclaration d'un attribut ou d'une méthode. `<requête>` est une expression qui retourne un résultat dont le type doit être compatible avec le type de l'attribut, ou de la méthode, déclaré dans `<déclaration>`. Dans le cas où il s'agit d'une méthode, `<requête>` peut utiliser les paramètres spécifiés dans la déclaration de la méthode.

**Exemple**

Pour imposer qu'une personne majeure doit avoir de l'argent, on peut écrire indifféremment :

- **context** Personne
  **inv : let** argent=compte.solde->sum() **in** age>=18 implies argent>0
- **context** Personne
  **def :** argent : int = compte.solde->sum()
  **context** Personne
  **inv :** age>=18 implies argent>0

*sum()* est décrit section 4.6.2.

### 4.3.7 Initialisation (*init*) et évolution des attributs (*derive*)

Le type de contrainte *init* permet de préciser la valeur initiale d'un attribut ou d'une terminaison d'association.

Les diagrammes d'UML définissent parfois des attributs ou des associations dérivées. La valeur de tels éléments est toujours déterminée en fonctions d'autres éléments du diagramme. Le type de contrainte *derive* permet de préciser comment la valeur de ce type d'élément évolue.

Notez bien la différence entre ces deux types de contraintes. La contrainte *derive* impose une contrainte perpétuelle : l'élément dérivé doit toujours avoir la valeur imposé par l'expression de la contrainte *derive*. D'un autre côté, la contrainte *init* ne s'applique qu'au moment de la création d'une instance précisée par le contexte de la contrainte. Ensuite, la valeur de l'élément peut fluctuer indépendamment de la contrainte *init*.

**Syntaxe**

```
init : <requête>
derive : <requête>
```

**Exemple**

Quand on crée une personne, la valeur initiale de l'attribut *marié* est faux et la personne ne possède pas d'employeur :

**context** Personne::marié : Boolean
**init :** false
**context** Personne::employeur : Set(Société)
**init :** Set{}

Les collections (dont *Set* est une instance) sont décrites section 4.4.4. *Set{}* correspond à un ensemble vide.

L'âge d'une personne est la différence entre la date courante et la date de naissance de la personne :

**context** Personne::age : Integer
**derive :** date_de_naissance - Date::current()

On suppose ici que le type *Date* possède une méthode de classe permettant de connaître la date courante et que l'opération moins (-) entre deux dates est bien définie et retourne un nombre d'années.

## 4.4 Types et opérations utilisables dans les expressions OCL

### 4.4.1 Types et opérateurs prédéfinis

Le langage OCL possède un certain nombre de types prédéfinis et d'opérations prédéfinies sur ces types. Ces types et ces opérations sont utilisables dans n'importe quelle contrainte et sont indépendants du modèle auquel sont rattachées ces contraintes.

Le tableau 4.1 donne un aperçu des types et opérations prédéfinis dans les contraintes OCL. Les tableaux 4.2 et 4.3 rappellent les conventions d'interprétation des opérateurs logiques.

L'opérateur logique *if-then-else-endif* est un peu particulier. Sa syntaxe est la suivante :

```
if <expression_logique_0>
then <expression_logique_1>
else <expression_logique_2>
endif
```

Cet opérateur s'interprète de la façon suivante : si <*expression_logique_0*> est vrai, alors la valeur de vérité de l'expression est celle de <*expression_logique_1*>, sinon, c'est celle de <*expression_logique_2*>.

| Type | Exemples de valeurs | Opérateurs |
|---|---|---|
| Boolean | *true* ; *false* | *and* ; *or* ; *xor* ; *not* ; *implies* ; *if-then-else-endif* ; … |
| Integer | 1 ; −5 ; 2 ; 34 ; 26524 ; … | * ; + ; − ; / ; *abs*() ; … |
| Real | 1,5 ; 3,14 ; … | * ; + ; − ; / ; *abs*() ; *floor*() ; … |

| String | "To be or not to be …" | *concat*() ; *size*() ; *substring*() ; … |
|---|---|---|

Tableau 4.1: Types et opérateurs prédéfinis dans les contraintes OCL.

| *E*1 | *E*2 | *P*1 *and P*2 | *P*1 *or P*2 | *P*1 *xor P*2 | *P*1 *implies P*2 |
|---|---|---|---|---|---|
| vrai | vrai | vrai | vrai | faux | vrai |
| vrai | faux | faux | vrai | vrai | faux |
| faux | vrai | faux | vrai | vrai | vrai |
| faux | faux | faux | faux | faux | vrai |

Tableau 4.2: Interprétation des quatre connecteurs, *E*1 et *E*2 sont deux expressions logiques.

| *expression* | *not expression* |
|---|---|
| vrai | faux |
| faux | vrai |

Tableau 4.3: Convention d'interprétation de la négation.

### 4.4.2 Types du modèle UML

Toute expression OCL est écrite dans le contexte d'un modèle UML donné. Bien entendu, tous les classeurs de ce modèle sont des types dans les expressions OCL attachées à ce modèle.

Dans la section 4.3.1, nous avons introduit le type énuméré. Une contrainte OCL peut référencer une valeur de ce type de la manière suivante :

```
<nom_type_enuméré>::valeur
```

Par exemple, la classe *Personne* possède un attribut *genre* de type *Genre*. On peut donc écrire la contrainte :

**context** Personne
**inv :** genre = Genre::femme

Dans ce cas, toutes les personnes doivent être des femmes.

### 4.4.3 OCL est un langage typé

OCL est un langage typé dont les types sont organisés sous forme de hiérarchie. Cette hiérarchie détermine comment différents types peuvent être combinés. Par exemple, il est impossible de comparer un booléen (*Boolean*) avec un entier (*Integer*) ou une chaîne de caractères (*String*). Par contre, il est possible de comparer un entier (*Integer*) et un réel (*Real*) car le type entier est un *sous-type* du type réel dans la hiérarchie des types OCL. Bien entendu, la hiérarchie des types du modèle UML est donnée par la relation de généralisation entre les classeurs du modèle UML.

### 4.4.4 Collections

OCL définit également la notion d'ensemble sous le terme générique de collection (*collection* en anglais). Il existe plusieurs sous-types du type abstrait *Collection* :

**Ensemble (*Set*) :**
collection non ordonnée d'éléments uniques (*i.e.* pas d'élément en double).

**Ensemble ordonné (*OrderedSet*) :**
collection ordonnée d'éléments uniques.

**Sac (*Bag*) :**
collection non ordonnée d'éléments identifiables (*i.e.* comme un ensemble, mais pouvant comporter des doublons).

**Séquence (*Sequence*) :**
collection ordonnée d'éléments identifiables.

Jusqu'à UML 2.0 exclu, les collections étaient toujours plates : une collection ne pouvait pas posséder des collections comme éléments. Cette restriction n'existe plus à partir d'UML 2.0.

## 4.5 Accès aux caractéristiques et aux objets dans les contraintes OCL

Dans une contrainte OCL associée à un objet, il est possible d'accéder aux caractéristiques (attributs, opérations et terminaison d'association) de cet objet, et donc, d'accéder de manière transitive à tous les objets (et leurs caractéristiques) avec qui il est en relation.

### 4.5.1 Accès aux attributs et aux opérations (*self*)

Pour faire référence à un attribut ou une opération de l'objet désigné par le contexte, il suffit d'utiliser le nom de cet élément. L'objet désigné par le contexte est également accessible par l'expression *self*. On peut donc également utiliser la notation pointée : `self.<propriété>`.

Une opération peut avoir des paramètres, il faut alors les préciser entre les parenthèses de l'opération.

Lorsque la multiplicité d'un attribut, de type *T*, n'est pas 1 (donc s'il s'agit d'un tableau), la référence à cet attribut est du type *ensemble* (*i.e. Set*(*T*)).

Par exemple, dans le contexte de la classe *Compte*, on peut utiliser les expressions suivantes :

- solde
- **self.**solde

- getSolde()
- **self.**getSolde()
- débiter(1000)
- **self.**débiter(1000)

Dans l'exemple précédent, le résultat de l'expression ***self.****débiter(1000)* est un singleton du type *Real*. Mais une opération peut comporter des paramètres définis en sortie ou en entrée/sortie. Dans ce cas, le résultat sera un tuple contenant tous les paramètres définis en sortie ou en entrée/sortie. Par exemple, imaginons une opération dont la déclaration serait *operation(out param_out : Integer):Real* possédant un paramètre défini en sortie *param_out*. Dans ce cas, le résultat de l'expression *operation(paramètre)* est un tuple de la forme *(param_out : Integer, result : Real)*. On peut accéder aux valeurs de ce tuple de la façon suivante :

- operation(paramètre).param_out
- operation(paramètre).result

### 4.5.2 Navigation via une association

Pour faire référence à un objet, ou un groupe d'objets, en association avec l'objet désigné par le contexte, il suffit d'utiliser le nom de la classe associée (en minuscule) ou le nom du rôle d'association du coté de cette classe. Quand c'est possible, il est préférable d'utiliser le nom de rôle de l'association du côté de l'objet auquel on désire faire référence. C'est indispensable s'il existe plusieurs associations entre l'objet désigné par le contexte et l'objet auquel on désire accéder, ou si l'association empruntée est réflexive.

Le type du résultat dépend de la propriété structurelle empruntée pour accéder à l'objet référencé, et plus précisément de la multiplicité du côté de l'objet référencé, et du type de l'objet référencé proprement dit. Si on appelle *X* la classe de l'objet référencé, dans le cas d'une multiplicité de :

- 1, le type du résultat est *X* (ex : 0..1 — X) ;
- * ou 0..*n*, …, le type du résultat est *Set*(*X*) (ex : * — X) ;
- * ou 0..*n*, …, et s'il y a en plus une contrainte *{ordered}*, le type du résultat est *OrderedSet*(*X*) (ex : {ordered} * — X).

Emprunter une seule propriété structurelle peut produire un résultat du type *Set* (ou *OrderedSet*). Emprunter plusieurs propriétés structurelles peut produire un résultat du type *Bag* (ou *Sequence*).

Par exemple, dans le contexte de la classe *Société* (**context** Société) :

- `directeur` désigne le directeur de la société (résultat de type *Personne*) ;
- `employé` désigne l'ensemble des employés de la société (résultat de type *Set*(*Personne*)) ;
- `employé.compte` désigne l'ensemble des comptes de tous les employés de la société (résultat de type *Bag*(*Compte*)) ;
- `employé.date_de_naissance` désigne l'ensemble des dates de naissance des employés de la société (résultat de type *Bag*(*Date*)).

### 4.5.3 Navigation via une association qualifiée

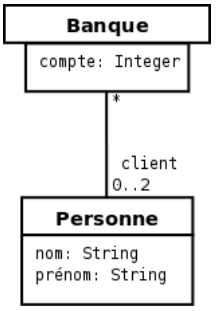



Figure 4.10: Diagramme illustrant une association qualifiée entre une classe *Banque* et une classe *Personne*.

Une association qualifiée (cf. section3.3.6) utilise un ou plusieurs qualificatifs pour sélectionner des instances de la classe cible de l'association. Pour emprunter une telle association, il est possible de spécifier les valeurs, ou les instances, des qualificatifs en utilisant des crochets ([]).

Plaçons-nous dans le cadre du diagramme de la figure 4.10. Dans le contexte de banque (**context** Banque), pour faire référence au nom des clients dont le compte porte le numéro 19503800 il faut écrire :

```
self.client[19503800].nom
```

Dans le cas où il y a plusieurs qualificatifs, il faut séparer chacune des valeurs par une virgule en respectant l'ordre des qualificatifs du diagramme UML. Il n'est pas possible de ne préciser la valeur que de certains qualificatifs en en laissant d'autres non définis. Par contre, il est possible de ne préciser aucune valeur de qualificatif :

```
self.client.nom
```

Dans ce cas, le résultat sera l'ensemble des noms de tous les clients de la banque.

Ainsi, si on ne précise pas la valeur des qualificatifs en empruntant une association qualifiée, tout se passe comme si l'association n'était pas qualifiée. Dans ce cas, faites attention à la cardinalité de la cible qui change quand l'association n'est plus qualifiée (cf. section3.3.6).

### 4.5.4 Navigation vers une classe association

Pour naviguer vers une classe association, il faut utiliser la notation pointée classique en précisant le nom de la classe association en minuscule. Par exemple, dans le contexte de la classe *Société* (**context** Société), pour accéder au salaire de tous les employés, il faut écrire :

self.poste.salaire

Cependant, dans le cas où l'association est réflexive (c'est le cas de la classe association *Mariage*), il faut en plus préciser par quelle extrémité il faut emprunter l'association. Pour cela, on précise le nom de rôle de l'une des extrémités de l'association entre crochets ([]) derrière le nom de la classe association. Par exemple, dans le contexte de la classe *Personne* (**context** Personne), pour accéder à la date de mariage de toutes les femmes, il faut écrire :

self.mariage[femme].date

### 4.5.5 Navigation depuis une classe association

Il est tout-à-fait possible de naviguer directement depuis une classe association vers une classe participante.

Exemple :

**context** Poste
**inv :** self.employé.age > 21

Par définition même d'une classe association, naviguer depuis une classe association vers une classe participante produit toujours comme résultat un objet unique. Par exemple, l'expression *self.employé.age* de l'exemple précédant produit bien un singleton.

### 4.5.6 Accéder à une caractéristique redéfinie (*oclAsType()*)

Quand une caractéristique définie dans une classe parente est redéfinie dans une sous-classe associée, la caractéristique de la classe parente reste accessible dans la sous-classe en utilisant l'expression *oclAsType()*.

Supposons une classe *B* héritant d'une classe *A* et une propriété *p1* définie dans les deux classes. Dans le contexte de la classe *B* (**context** B), pour accéder à la propriété *p1* de *B*, on écrit simplement :

self.p1

et pour accéder à la propriété *p1* de *A* (toujours dans le contexte de *B*), il faut écrire :

self.oclAsType(A).p1

### 4.5.7 Opérations prédéfinies sur tous les objets

L'opération *oclAsType*, que nous venons de décrire (section 4.5.6), est une opération prédéfinie dans le langage OCL qui peut être appliquée à tout objet. Le langage OCL en propose plusieurs :

- oclIsTypeOf (t : OclType) : Boolean
- oclIsKindOf (t : OclType) : Boolean
- oclInState (s : OclState) : Boolean
- oclIsNew () : Boolean
- oclAsType (t : OclType) : instance of OclType

#### Opération *oclIsTypeOf*

*oclIsTypeOf* retourne *vrai* si le type de l'objet au titre duquel cette opération est invoqué est exactement le même que le type *t* passé en paramètre.

Par exemple, dans le contexte de *Société*, l'expression *directeur.oclIsTypeOf(Personne)* est vraie tandis que l'expression *self.oclIsTypeOf(Personne)* est fausse.

#### Opération *oclIsKindOf*

*oclIsKindOf* permet de déterminer si le type *t* passé en paramètre correspond exactement au type ou à un type parent du type de l'objet au titre duquel cette opération est invoqué.

Par exemple, supposons une classe *B* héritant d'une classe *A* :

- dans le contexte de *B*, l'expression *self.oclIsKindOf(B)* est vraie ;
- toujours dans le contexte de *B*, l'expression *self.oclIsKindOf(A)* est vraie ;
- mais dans le contexte de *A*, l'expression *self.oclIsKindOf(B)* est fausse.

#### Opération *oclIsNew*

L'opération *oclIsNew* doit être utilisée dans une postcondition. Elle est vraie quand l'objet au titre duquel elle est invoqué est créé pendant l'opération (*i.e.* l'objet n'existait pas au moment des préconditions).

#### Opération *oclInState*

Cette opération est utilisée dans un diagramme d'états-transitions (cf. section 5). Elle est vraie si l'objet décrit par le diagramme d'états-transitions est dans l'état *s* passé en paramètre. Les valeurs possibles du paramètre *s* sont les noms des états du diagramme d'états-transitions. On peut faire référence à un état imbriqué en utilisant des «: :» (par exemple, pour faire référence à un état B imbriqué dans un état A, on écrit : A: : B).

### 4.5.8 Opération sur les classes

Toutes les opérations que nous avons décrites jusqu'ici s'appliquaient sur des instances de classe. Cependant, OCL permet également d'accéder à des caractéristiques de classe (celles qui sont soulignées dans un diagramme de classes). Pour cela, on utilise le nom qualifié de la classe suivi d'un point puis du nom de la propriété ou de l'opération : *<nom_qualifié>.<propriété>*.

Le langage OCL dispose également d'une opération prédéfinie sur les classes, les interfaces et les énumérations (*allInstances*) qui retourne l'ensemble (*Set*) de toutes les instances du type au titre duquel elle est invoquée, au moment où l'expression est évaluée. Par exemple, pour désigner l'ensemble des instances de la classe personne (type *set*(*Personne*)) on écrit :

```
Personne.allInstances()
```

# 4.6 Opérations sur les collections

## 4.6.1 Introduction : «.», «->», «::» et *self*

Comme nous l'avons vu dans la section précédente (4.5), pour accéder aux caractéristiques (attributs, terminaisons d'associations, opérations) d'un objet, OCL utilise la notation pointée : <*objet*>.<*propriété*>. Cependant, de nombreuses expressions ne produisent pas comme résultat un objet, mais une collection. Le langage OCL propose plusieurs opérations de base sur les collections. Pour accéder ce type d'opération, il faut, utiliser non pas un point mais une flèche : <*collection*>-><*opération*>. Enfin, rappelons que pour désigner un élément dans un élément englobant on utilise les «: :» (cf. section 4.3.2 et 4.5.7 par exemple). En résumé :

**«::» –**
permet de désigner un élément (comme une opération) dans un élément englobant (comme un classeur ou un paquetage) ;

**«.» –**
permet d'accéder à une caractéristique (attributs, terminaisons d'associations, opérations) d'un objet ;

**«->» –**
permet d'accéder à une caractéristiques d'une collection.

Nous avons dit dans la section 4.5.1 que l'objet désigné par le contexte est également accessible par l'expression *self*. *self* n'est pas uniquement utilisé pour désigner le contexte d'une contrainte dans une expression mais également pour désigner le contexte d'une sous-expression dans le texte (en langage naturel). Ainsi, lorsque l'on utilise *self* pour une opération <*opération*>, c'est pour désigner l'objet (comme une collection par exemple) sur lequel porte l'opération. Cette objet peut être le résultat d'une opération intermédiaire comme l'évaluation de l'expression <*expression*> précédant l'opération <*opération*> dans l'expression complète : <*expression*>.<*opération*>.

## 4.6.2 Opérations de base sur les collections

Nous ne décrirons pas toutes les opérations sur les collections et ses sous-types (ensemble, …) dans cette section. Référez vous à la documentation officielle [19] pour plus d'exhaustivité.

#### Opérations de base sur les collections

Nous décrivons ici quelques opérations de base sur les collections que propose le langage OCL.

**size():Integer –**
retourne le nombre d'éléments (la cardinalité) de *self*.

**includes(objet:T):Boolean –**
vrai si *self* contient l'objet *objet*.

**excludes(objet:T):Boolean –**
vrai si *self* ne contient pas l'objet *objet*.

**count(objet:T):Integer –**
retourne le nombre d'occurrences de *objet* dans *self*.

**includesAll(c:Collection(T)):Boolean –**
vrai si *self* contient tous les éléments de la collection *c*.

**excludesAll(c:Collection(T)):Boolean –**
vrai si *self* ne contient aucun élément de la collection *c*.

**isEmpty() –**
vrai si *self* est vide.

**notEmpty() –**
vrai si *self* n'est pas vide.

**sum():T**
retourne la somme des éléments de *self*. Les éléments de *self* doivent supporter l'opérateur *somme* (+) et le type du résultat dépend du type des éléments.

**product(c2:Collection(T2)):Set(Tuple(first:T,second:T2)) –**
le résultat est la collection de Tuple correspondant au produit cartésien de *self* (de type Collection(T)) par *c2*.

#### Opérations de base sur les ensembles (*Set*)

Nous décrivons ici quelques opérations de base sur les ensembles (type *Set*) que propose le langage OCL.

**union(set:Set(T)):Set(T) –**
retourne l'union de *self* et *set*.

**union(bag:Bag(T)):Bag(T) –**
retourne l'union de *self* et *bag*.

**=(set:Set(T)):Boolean –**
vrai si *self* et *set* contiennent les mêmes éléments.

**intersection(set:Set(T)):Set(T) –**
intersection entre *self* et *set*.

**intersection(bag:Bag(T)):Set(T) –**
intersection entre *self* et *bag*. [1]

**including(objet:T):Set(T) –**

Le résultat contient tous les éléments de *self* plus l'objet *objet*.

**excluding(objet:T):Set(T) –**
Le résultat contient tous les éléments de *self* sans l'objet *objet*.

**-(set:Set(T)):Set(T) –**
Le résultat contient tous les éléments de *self* sans ceux de *set*.

**asOrderedSet():OrderedSet(T) –**
permet de convertir *self* du type *Set(T)* en *OrderedSet(T)*.

**asSequence():Sequence(T) –**
permet de convertir *self* du type *Set(T)* en *Sequence(T)*.

**asBag():Bag(T) –**
permet de convertir *self* du type *Set(T)* en *Bag(T)*.

**Remarque :**

les sacs (type *Bag*) disposent d'opérations analogues.

**Exemples**

1. Une société a au moins un employé :

   **context** Société **inv :** self.employé->notEmpty()

2. Une société possède exactement un directeur :

   **context** Société **inv :** self.directeur->size()=1

3. Le directeur est également un employé :

   **context** Société **inv :** self.employé->includes(self.directeur)

### 4.6.3 Opération sur les éléments d'une collection

**Syntaxe générale**

La syntaxe d'une opération portant sur les éléments d'une collection est la suivante :

```
<collection> -> <opération>( <expression> )
```

Dans tous les cas, l'expression <*expression*> est évaluée pour chacun des éléments de la collection <*collection*>. L'expression <*expression*> porte sur les caractéristiques des éléments en les citant directement par leur nom. Le résultat dépend de l'opération <*opération*>.

Parfois, dans l'expression <*expression*>, il est préférable de faire référence aux caractéristiques de l'élément courant en utilisant la notation pointée : <*élément*>.<*propriété*>. Pour cela, on doit utiliser la syntaxe suivante :

```
<collection> -> <opération>( <élément> | <expression> )
```

<*élément*> joue alors un rôle d'itérateur et sert de référence à l'élément courant dans l'expression <*expression*>.

Il est également possible, afin d'être plus explicite, de préciser le type de cet élément :

```
<collection> -> <opération>( <élément> : <Type> | <expression> )
```

La syntaxe générale d'une opération portant sur les éléments d'une collection est donc la suivante :

```
<collection> -> <opération>( [ <élément> [ : <Type> ] | ] <expression> )
```

**Opération *select* et *reject***

Ces deux opérations permettent de générer une sous-collection en filtrant les éléments de la collection *self*. Leur syntaxe est la suivante :

```
select( [ <élément> [ : <Type> ] | ] <expression_logique> )
reject( [ <élément> [ : <Type> ] | ] <expression_logique> )
```

**select**
permet de générer une sous-collection de *self* ne contenant que des éléments qui satisfont l'expression logique <*expression_logique*>.

**reject**
permet de générer une sous-collection contenant tous les éléments de *self* excepté ceux qui satisfont l'expression logique <*expression_logique*>.

Par exemple, pour écrire une contrainte imposant que toute société doit posséder, parmi ses employés, au moins une personne de plus de 50 ans, on peut écrire indifféremment :

1. **context** Société
   **inv:** self.employé->select(age > 50)->notEmpty()
2. **context** Société
   **inv:** self.employé->select(individu | individu.age > 50)->notEmpty()
3. **context** Société
   **inv:** self.employé->select(individu : Personne | individu.age > 50)->notEmpty()

**Opération *forAll* et *exists***

Ces deux opérations permettent de représenter le quantificateur universel (∀) et le quantificateur existentiel (∃). Le résultat de ces opérations est donc du type *Boolean*. Leur syntaxe est la suivante :

```
forAll( [ <élément> [ : <Type> ] | ] <expression_logique> )
exists( [ <élément> [ : <Type> ] | ] <expression_logique> )
```

**forAll**
permet d'écrire une expression logique vraie si l'expression <*expression_logique*> est vraie pour tous les éléments de *self*.

**exists**
permet d'écrire une expression logique vraie si l'expression <*expression_logique*> est vraie pour au moins un élément de *self*.

Par exemple, pour écrire une contrainte imposant que toute société doit posséder, parmi ses employés, au moins une personne de plus de 50 ans, on peut écrire :

**context** Société
**inv:** self.employé->exists(age > 50)

L'opération *forAll* possède une variante étendue possédant plus d'un itérateur. Dans ce cas, chacun des itérateurs parcourra l'ensemble de la collection. Concrètement, une opération *forAll* comportant deux itérateurs est équivalente à une opération *forAll* n'en comportant qu'un, mais réalisée sur le produit cartésien de *self* par lui-même.

Par exemple, imposer qu'il n'existe pas deux instances de la classe *Personne* pour lesquelles l'attribut *nom* a la même valeur, c'est à dire pour imposer que deux personnes différentes ont un nom différent, on peut écrire indifféremment :

1. **context** Personne
   **inv:** Personne.allInstances()->forAll(p1, p2 | p1 <> p2 implies p1.nom <> p2.nom)
2. **context** Personne
   **inv:** (Personne.allInstances().product(Personne.allInstances()))
   ->forAll(tuple | tuple.first <> tuple.second implies tuple.first.nom <> tuple.second.nom)

**Opération *collect***

Cette opération permet de construire une nouvelle collection en utilisant la collection *self*. La nouvelle collection construite possède le même nombre d'éléments que la collection *self*, mais le type de ces éléments est généralement différent. La syntaxe de l'opérateur *collect* est la suivante :

```
collect( [ <élément> [ : <Type> ] | ] <expression> )
```

Pour chaque élément de la collection *self*, l'opérateur *collect* évalue l'expression <*expression*> sur cet élément et ajoute le résultat dans la collection générée.

Par exemple, pour définir la collection des dates de naissance des employés d'une société, il faut écrire, dans le contexte de la classe *Société* :

self.employé->collect(date_de_naissance)

Puisque, toujours dans le contexte de la classe *Société*, l'expression *self.employé->collect(date_de_naissance)->size() = self.employé->size()* est toujours vraie, il faut en conclure que le résultat d'une opération *collect* sur une collection du type *Set* n'est pas du type *Set* mais du type *Bag*. En effet, dans le cadre de notre exemple, il y aura certainement des doublons dans les dates de naissance.

### 4.6.4 Règles de précédence des opérateurs

Ordre de précédence pour les opérateurs par ordre de priorité décroissante :

1. *@pre*
2. «.» et «->»
3. *not* et «-» (opérateur unaire)
4. «*» et «/»
5. «+» et «-»(opérateur binaire)
6. *if-then-else-endif*
7. «<», «>», «<=» et «>=»
8. «=» et «<>»
9. *and*, *or* et *xor*
10. *implies*

Les parenthèses, «(» et «)», permettent de changer cet ordre.

---

[1] Non, il n'y a pas d'erreur de copier/coller : réfléchissez !

## 4.7 Exemples de contraintes

---

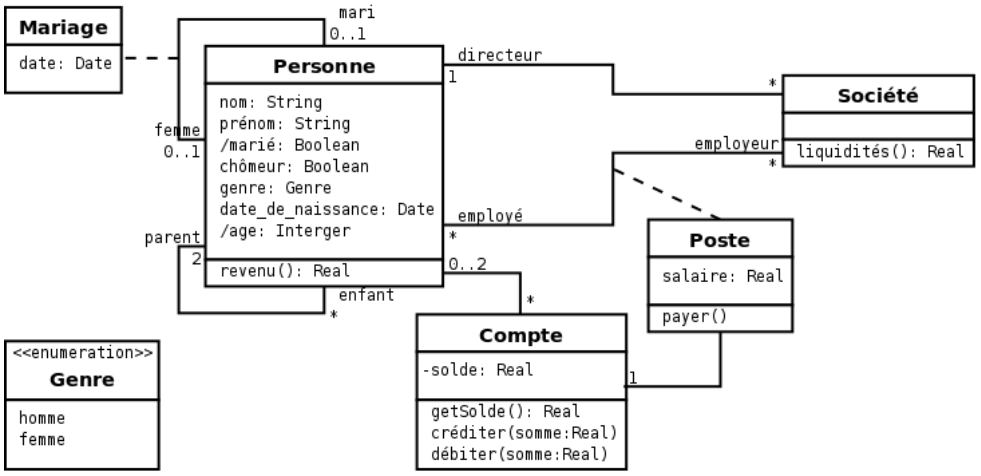



Figure 4.11: Reprise du diagramme de la figure 4.8.

---

Dans cette section, nous allons illustrer par quelques exemples l'utilisation du langage OCL. Nous restons toujours sur le diagramme de classes de la figure 4.8 représenté à nouveau sur la figure 4.11 pour des raisons de proximité.

1. Dans une société, le directeur est un employé, n'est pas un chômeur et doit avoir plus de 40 ans. De plus, une société possède exactement un directeur et au moins un employé.

```
context Société
inv :
  self.directeur->size()=1 and
  not(self.directeur.chômeur) and
  self.directeur.age > 40 and
  self.employé->includes(self.directeur)
```

2. Une personne considérée comme au chômage ne doit pas avoir des revenus supérieurs à 100 €.

```
context Personne
inv :
  let revenus : Real = self.poste.salaire->sum() in
    if chômeur then
      revenus < 100
    else
      revenus >= 100
    endif
```

3. Une personne possède au plus 2 parents (référencés).

```
context Personne
inv : parent->size()<=2
```

4. Si une personne possède deux parents, l'un est une femme et l'autre un homme.

```
context Personne
inv :
  parent->size()=2 implies
     ( parent->exists(genre=Genre::homme) and
       parent->exists(genre=Genre::femme) )
```

5. Tous les enfants d'une personne ont bien cette personne comme parent et inversement.

```
context Personne
inv :
  enfant->notEmpty() implies
    enfant->forAll( p : Personne | p.parents->includes(self))

context Personne
inv :
  parent->notEmpty() implies
    parent->forAll ( p : Personne | p.enfant->includes (self))
```

6. Pour être marié, il faut avoir une femme ou un mari.

```
context Personne::marié
derive : self.femme->notEmpty() or self.mari->notEmpty()
```

7. Pour être marié, il faut avoir plus de 18 ans. Un homme est marié avec exactement une femme et une femme avec exactement un homme.

```
context Personne
inv :
  self.marié implies
    self.genre=Genre::homme implies (
      self.femme->size()=1 and
      self.femme.genre=Genre::femme)
    and self.genre=Genre::femme implies (
      self.mari->size()=1 and
      self.mari.genre=Genre::homme)
    and self.age >=18
```

# Chapitre 5 Diagramme d'états-transitions

# *(State machine diagram)*

## 5.1 Introduction au formalisme

### 5.1.1 Présentation

Les diagrammes d'états-transitions d'UML décrivent le comportement interne d'un objet à l'aide d'un automate à états finis. Ils présentent les séquences possibles d'états et d'actions qu'une instance de classe peut traiter au cours de son cycle de vie en réaction à des événements discrets (de type signaux, invocations de méthode).

Ils spécifient habituellement le comportement d'une instance de classeur (classe ou composant), mais parfois aussi le comportement interne d'autres éléments tels que les cas d'utilisation, les sous-systèmes, les méthodes.

Le diagramme d'états-transitions est le seul diagramme, de la norme UML, à offrir une vision complète et non ambiguë de l'ensemble des comportements de l'élément auquel il est attaché. En effet, un diagramme d'interaction n'offre qu'une vue partielle correspondant à un scénario sans spécifier comment les différents scénarii interagissent entre eux.

La vision globale du système n'apparaît pas sur ce type de diagramme puisqu'ils ne s'intéressent qu'à un seul élément du système indépendamment de son environnement.

Concrètement, un diagramme d'états-transitions est un graphe qui représente un *automate à états finis*, c'est-à-dire une machine dont le comportement des sorties ne dépend pas seulement de l'état de ses entrées, mais aussi d'un historique des sollicitations passées.

### 5.1.2 Notion et exemple d'automate à états finis

Comme nous venons de le dire, un automate à états finis est un automate dont le comportement des sorties ne dépend pas seulement de l'état de ses entrées, mais aussi d'un historique des sollicitations passées. Cet historique est caractérisé par un *état global*.

Un état global est un jeu de valeurs d'objet, pour une classe donnée, produisant la même réponse face aux événements. Toutes les instances d'une même classe ayant le même état global réagissent de la même manière à un événement. Il ne faut pas confondre les notions d'état global et d'état. La section 5.2.1 donne plus d'information sur ces deux acceptions du terme *état*.

Un automate à états finis est graphiquement représenté par un graphe comportant des états, matérialisés par des rectangles aux coins arrondis, et des transitions, matérialisées par des arcs orientés liant les états entre eux.

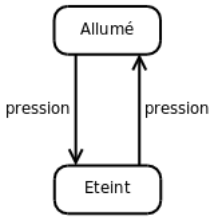



Figure 5.1: Un diagramme d'états-transitions simple.

La figure 5.1 montre un exemple simple d'automate à états finis. Cet automate possède deux états (*Allumé* et *Eteint*) et deux transitions correspondant au même évènement : la pression sur un bouton d'éclairrage domestique. Cet automate à états finis illustre en fait le fonctionnement d'un télérupteur dans une maison. Lorsque l'on appuie sur un bouton d'éclairrage, la réaction de l'éclairage associé dépendra de son état courant (de son historique) : s'il la lumière est allumée, elle s'éteindra, si elle est éteinte, elle s'allumera.

### 5.1.3 Diagrammes d'états-transitions

Un diagramme d'états-transitions rassemble et organise les états et les transitions d'un classeur donné. Bien entendu, le modèle dynamique du système comprend plusieurs diagrammes d'états-transitions. Il est souhaitable de construire un diagramme d'états-transitions pour chaque classeur (qui, le plus souvent, est une classe) possédant un comportement dynamique important. Un diagramme d'états-transitions ne peut être associé qu'à un seul classeur. Tous les automates à états finis des diagrammes d'états-transitions d'un système s'exécutent concurremment et peuvent donc changer d'état de façon indépendante.

## 5.2 État

### 5.2.1 Les deux acceptions du terme *état*

#### État dans un diagrammes d'états-transitions

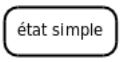



Figure 5.2: Exemple d'état simple.

Comme nous l'avons déjà dit, un état, que l'on peut qualifier informellement d'*élémentaire*, se représente graphiquement dans un diagrammes d'états-transitions par un rectangles aux coins arrondis (figure 5.2).

Certains états, dits *composites* (cf. section 5.6), peuvent contenir (*i.e.* envelopper) des sous-états.

Le nom de l'état peut être spécifié dans le rectangles et doit être unique dans le diagrammes d'états-transitions, ou dans l'état enveloppant. On peut l'omettre, ce qui produit un état anonyme. Il peut y avoir un nombre quelconque d'états anonymes distincts. Un état imbriqué peut être identifié par son nom qualifié (cf. section 2.4.2) si tous les états enveloppant ont des noms.

Un état peut être partitionné en plusieurs compartiments séparés par une ligne horizontale. Le premier compartiment contient le nom de l'état et les autres peuvent recevoir des transitions interne (cf. section 5.4.6), ou des sous-états (cf. section 5.6), quand il s'agit d'un état composite. Dans le cas d'un état simple (*i.e.* sans transitions interne ou sous-état), on peut omettre toute barre de séparation (figure 5.2).

**État d'un objet, ou du diagrammes d'états-transitions (*i.e. état global*)**

Un objet peut passer par une série d'états pendant sa durée de vie. Un état représente une période dans la vie d'un objet pendant laquelle ce dernier attend un événement ou accomplit une activité. La configuration de l'état global de l'objet est le jeu des états (élémentaires) qui sont actifs à un instant donné.

Dans le cas d'un diagramme d'états-transitions simple (sans transition concurrente), il ne peut y avoir qu'un seul état actif à la fois. Dans ce cas, les notions d'état actif et d'état global se rejoignent.

Cependant, la configuration de l'état global peut contenir plusieurs états actifs à un instant donné. On parle d'états concurrents (cf. section 5.6.5) quand plusieurs états sont actifs en même temps et on dit qu'il y a concurrence au sein de l'objet. Le nombre d'états actifs peut changer pendant la durée de vie d'un objet du fait d'embranchements ou de jointures appelées transitions concurrentes (cf. section 5.6.5).

### 5.2.2 État initial et final

**État initial**

Figure 5.3: Représentation graphique de l'état initial.

L'état initial est un pseudo état qui indique l'état de départ, par défaut, lorsque le diagramme d'états-transitions, ou l'état enveloppant, est invoqué. Lorsqu'un objet est créé, il entre dans l'état initial.

**État final**

Figure 5.4: Représentation graphique de l'état final.

L'état final est un pseudo état qui indique que le diagramme d'états-transitions, ou l'état enveloppant, est terminé.

## 5.3 Événement

### 5.3.1 Notion d'évènement

Un événement est quelque chose qui se produit pendant l'exécution d'un système et qui mérite d'être modélisé. Les diagrammes d'états-transitions permettent justement de spécifier les réactions d'une partie du système à des événements discrets. Un événement se produit à un instant précis et est dépourvu de durée. Quand un événement est reçu, une transition peut être déclenchée et faire basculer l'objet dans un nouvel état. On peut diviser les événements en plusieurs types explicites et implicites : signal, appel, changement et temporel.

### 5.3.2 Événement de type signal (*signal*)

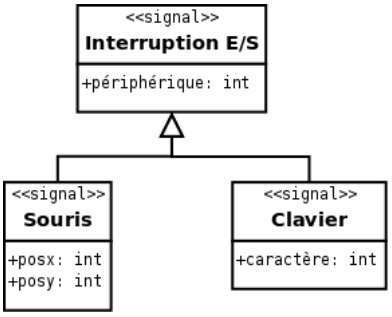



Figure 5.5: Déclaration de signaux et héritage.

Un signal est un type de classeur destiné explicitement à véhiculer une communication asynchrone à sens unique entre deux objets. L'objet expéditeur crée et initialise explicitement une instance de signal et l'envoi à un objet explicite ou à tout un groupe d'objets. Il n'attend pas que le destinataire traite le signal pour poursuivre son déroulement. La réception d'un signal est un événement pour l'objet destinataire. Un même objet peut être à la fois expéditeur et destinataire.

Les signaux sont déclarés par la définition d'un classeur portant le stéréotype *« signal »* ne fournissant pas d'opération et dont les attributs sont interprétés comme des arguments (cf. figure 5.5). La syntaxe d'un signal est la suivante :

```
<nom_événement> ( [ <paramettre> : <type> [; <paramettre> : <type> ... ] ] )
```

Les signaux supporte la relation de généralisation (cf. figure 5.5). Les signaux héritent des attributs de leurs parents (héritage) et ils déclenchent des transitions contenant le type du signal parent (polymorphisme).

### 5.3.3 Événement d'appel (*call*)

Un événement d'appel représente la réception de l'appel d'une opération par un objet. Les paramètres de l'opération sont ceux de l'événement d'appel. La syntaxe d'un événement d'appel est la même que celle d'un signal. Par contre, les événements d'appel sont des méthodes déclarées au niveau du diagramme de classes.

### 5.3.4 Événement de changement (*change*)

Un événement de changement est généré par la satisfaction (*i.e.* passage de faux à vrai) d'une expression booléenne sur des valeurs d'attributs. Il s'agit d'une manière

déclarative d'attendre qu'une condition soit satisfaite. La syntaxe d'un événement de changement est la suivante :

```
when ( <condition_booléenne> )
```

Notez la différence entre une condition de garde (cf. section 5.4.2) et un événement de changement. La première est évaluée une fois que l'événement déclencheur de la transition a lieu et que le destinataire le traite. Si elle est fausse, la transition ne se déclenche pas et la condition n'est pas réévaluée. Un événement de changement est évalué continuellement jusqu'à ce qu'il devienne vrai, et c'est à ce moment-là que la transition se déclenche.

### 5.3.5 Événement temporel (*after* ou *when*)

Les événements temporels sont générés par le passage du temps. Ils sont spécifiés soit de manière absolue (date précise), soit de manière relative (temps écoulé). Par défaut, le temps commence à s'écouler dès l'entrée dans l'état courant.

La syntaxe d'un événement temporel spécifié de manière relative est la suivante :

```
after ( <durée> )
```

Un événement temporel spécifié de manière absolue est défini en utilisant un événement de changement :

```
when ( date = <date> )
```

## 5.4 Transition

### 5.4.1 Définition et syntaxe

Une transition définit la réponse d'un objet à l'occurrence d'un événement. Elle lie, généralement, deux états $E1$ et $E2$ et indique qu'un objet dans un état $E1$ peut entrer dans l'état $E2$ et exécuter certaines activités, si un événement déclencheur se produit et que la condition de garde est vérifiée.

La syntaxe d'une transition est la suivante :

```
[ <événement> ][ '[' <garde> ']' ] [ '/' <activité> ]
```

La syntaxe de `<événement>` a été définie dans la section 5.3

Le même événement peut être le déclencheur de plusieurs transitions quittant un même état. Chaque transition avec le même événement doit avoir une condition de garde différente. En effet, une seule transition peut se déclencher dans un même flot d'exécution. Si deux transitions sont activées en même temps par un même événement, une seule se déclenche et le choix n'est pas prévisible (*i.e.* pas déterministe).

### 5.4.2 Condition de garde

Une transition peut avoir une condition de garde (spécifiée par `'[' <garde> ']'` dans la syntaxe). Il s'agit d'une expression logique sur les attributs de l'objet, associé au diagramme d'états-transitions, ainsi que sur les paramètres de l'événement déclencheur. La condition de garde est évaluée uniquement lorsque l'événement déclencheur se produit. Si l'expression est fausse à ce moment là, la transition ne se déclenche pas, si elle est vraie, la transition se déclenche et ses effets se produisent.

### 5.4.3 Effet d'une transition

Lorsqu'une transition se déclenche (on parle également de tir d'une transition), son effet (spécifié par `'/' <activité>` dans la syntaxe) s'exécute. Il s'agit généralement d'une activité qui peut être

- une opération primitive comme une instruction d'assignation ;
- l'envoi d'un signal ;
- l'appel d'une opération ;
- une liste d'activités, etc.

La façon de spécifier l'activité à réaliser est laissée libre (langage naturel ou pseudo-code).

Lorsque l'exécution de l'effet est terminée, l'état cible de la transition devient actif.

### 5.4.4 Transition externe

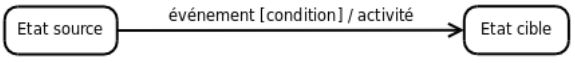



Figure 5.6: Représentation graphique d'une transition externe entre deux états.

Une transition externe est une transition qui modifie l'état actif. Il s'agit du type de transition le plus répandu. Elle est représentée par une flèche allant de l'état source vers l'état cible.

La figure 5.6 illustre la représentation graphique d'une transition externe entre deux états.

### 5.4.5 Transition d'achèvement

Une transition dépourvue d'événement déclencheur explicite se déclenche à la fin de l'activité contenue dans l'état source (y compris les état imbriqués). Elle peut contenir une condition de garde qui est évaluée au moment où l'activité contenue dans l'état s'achève, et non pas ensuite.

Les transitions de garde sont, par exemple, utilisées pour connecter les états initiaux et les états historiques (cf. section 5.6.3) avec leur état successeurs puisque ces pseudo-états ne peuvent rester actifs.

### 5.4.6 Transition interne

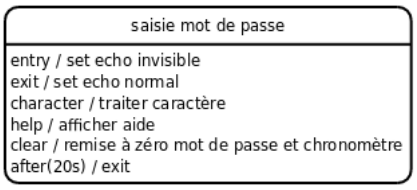



Figure 5.7: Représentation de la saisie d'un mot de passe dans un état unique en utilisant des transitions internes.

Les règles de déclenchement d'une transition interne sont les mêmes que pour une transition externe excepté qu'une transition interne ne possède pas d'état cible et que l'état actif reste le même à la suite de son déclenchement. La syntaxe d'une transition interne reste la même que celle d'une transition classique (cf. section 5.4.1). Par contre, les transitions internes ne sont pas représentées par des arcs mais sont spécifiées dans un compartiment de leur état associé (cf. figure 5.7).

Les transitions internes possèdent des noms d'événement prédéfinis correspondant à des déclencheurs particuliers : *entry*, *exit*, *do* et *include*. Ces mots clefs réservés viennent prendre la place du nom de l'événement dans la syntaxe d'une transition interne.

***entry* –**
*entry* permet de spécifier une activité qui s'accomplit quand on entre dans l'état.

***exit* –**
*exit* permet de spécifier une activité qui s'accomplit quand on sort de l'état.

***do* –**
Une activité *do* commence dès que l'activité *entry* est terminée. Lorsque cette activité est terminée, une transition d'achèvement peut être déclenchée, après l'exécution de l'activité *exit* bien entendu. Si une transition se déclenche pendant que l'activité *do* est en cours, cette dernière est interrompue et l'activité *exit* de l'état s'exécute.

***include* –**
permet d'invoquer un sous-diagramme d'états-transitions.

Les activités *entry* servent souvent à effectuer la configuration nécessaire dans un état. Comme il n'est pas possible de l'éluder, toute action interne à l'état peut supposer que la configuration est effectuée indépendamment de la manière dont on entre dans l'état. De manière analogue, une activité *exit* est une occasion de procéder à un nettoyage. Cela peut s'avérer particulièrement utile lorsqu'il existe des transitions de haut niveau qui représentent des conditions d'erreur qui abandonnent les états imbriqués.

Le déclenchement d'une transition interne ne modifie pas l'état actif et n'entraîne donc pas l'activation des activités *entry* et *exit*.

## 5.5 Point de choix

Il est possible de représenter des alternatives pour le franchissement d'une transition. On utilise pour cela des pseudo-états particuliers : les points de jonction (représentés par un petit cercle plein) et les points de décision (représenté par un losange).

### 5.5.1 Point de jonction

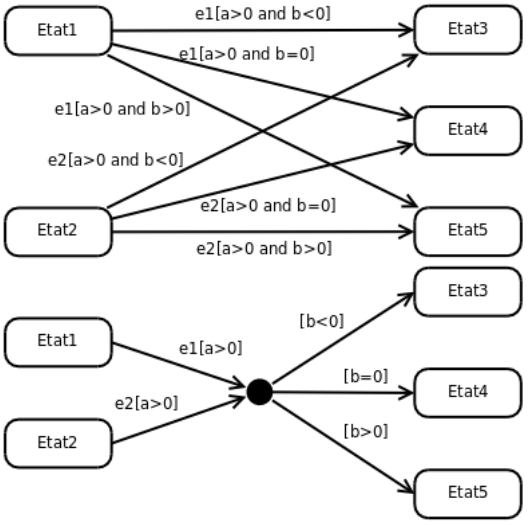



Figure 5.8: En haut, un diagramme sans point de jonction. En bas, son équivalent utilisant un point de jonction.

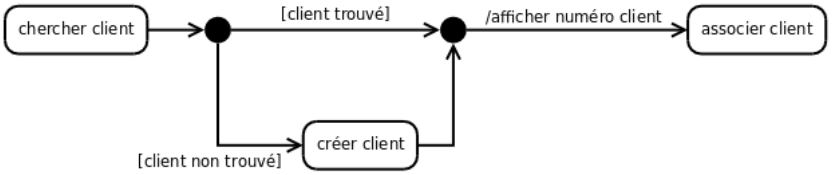



Figure 5.9: Exemple d'utilisation de deux points de jonction pour représenter une alternative.

Les points de jonction sont un artefact graphique (un pseudo-état en l'occurrence) qui permet de partager des segments de transition, l'objectif étant d'aboutir à une notation plus compacte ou plus lisible des chemins alternatifs.

Un point de jonction peut avoir plusieurs segments de transition entrante et plusieurs segments de transition sortante. Par contre, il ne peut avoir d'activité interne ni des

transitions sortantes dotées de déclencheurs d'événements.

Il ne s'agit pas d'un état qui peut être actif au cours d'un laps de temps fini. Lorsqu'un chemin passant par un point de jonction est emprunté (donc lorsque la transition associée est déclenchée) toutes les gardes le long de ce chemin doivent s'évaluer à vrai dès le franchissement du premier segment.

La figure 5.8 illustre bien l'utilité des points de jonction.

La figure 5.9 illustre l'utilisation de points de jonction pour représenter le branchement d'une clause conditionnelle.

### 5.5.2 Point de décision

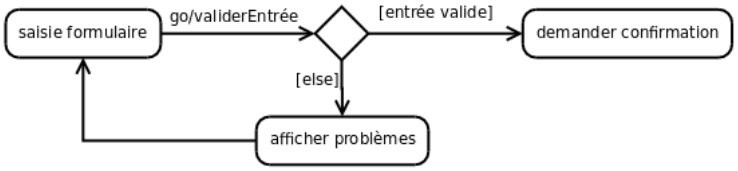



Figure 5.10: Exemple d'utilisation d'un point de décision.

Un point de décision possède une entrée et au moins deux sorties. Contrairement à un point de jonction, les gardes situées après le point de décision sont évaluées au moment où il est atteint. Cela permet de baser le choix sur des résultats obtenus en franchissant le segment avant le point de choix (cf. figure 5.10). Si, quand le point de décision est atteint, aucun segment en aval n'est franchissable, c'est que le modèle est mal formé.

Il est possible d'utiliser une garde particulière, notée `[else]`, sur un des segments en aval d'un point de choix. Ce segment n'est franchissable que si les gardes des autres segments sont toutes fausses. L'utilisation d'une clause `[else]` est recommandée après un point de décision car elle garantit un modèle bien formé.

## 5.6 États composites

### 5.6.1 Présentation

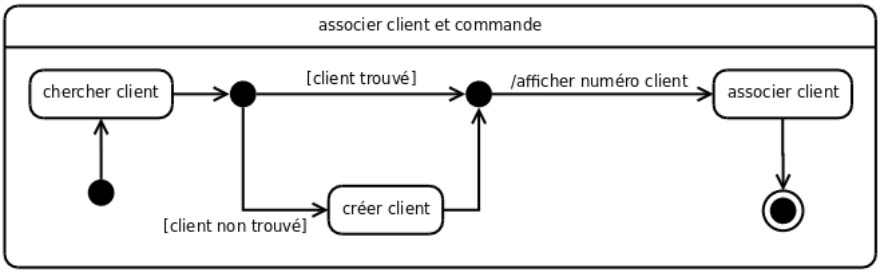



Figure 5.11: Exemple d'état composite modélisant l'association d'une commande à un client.

Un état simple ne possède pas de sous-structure mais uniquement, le cas échéant, un jeu de transitions internes. Un état composite est un état décomposé en régions contenant chacune un ou plusieurs sous-états.

Quand un état composite comporte plus d'une région, il est qualifié d'*état orthogonal*. Lorsqu'un état orthogonal est actif, un sous-état direct de chaque région est simultanément actif, il y a donc concurrence (cf. section 5.6.5). Un état composite ne comportant qu'une région est qualifié d'*état non orthogonal*.

Implicitement, tout diagramme d'états-transitions est contenu dans un état externe qui n'est usuellement pas représenté. Cela apporte une plus grande homogénéité dans la description : tout diagramme d'états-transitions est implicitement un état composite.

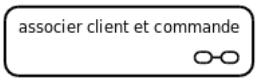



Figure 5.12: Notation abrégée d'un état composite.

L'utilisation d'états composites permet de développer une spécification par raffinements. Il n'est pas nécessaire de représenter les sous-états à chaque utilisation de l'état englobant. Une notation abrégée (figure 5.12) permet d'indiquer qu'un état est composite et que sa définition est donnée sur un autre diagramme.

La figure figure 5.11 montre un exemple d'état composite et la figure 5.12 montre sa notation abrégée.

### 5.6.2 Transition

Les transitions peuvent avoir pour cible la frontière d'un état composite et sont équivalentes à une transition ayant pour cible l'état initial de l'état composite.

Une transition ayant pour source la frontière d'un état composite est équivalente à une transition qui s'applique à tout sous-état de l'état composite source. Cette relation est transitive : la transition est franchissable depuis tout état imbriqué, quelle que soit sa profondeur.

Par contre, si la transition ayant pour source la frontière d'un état composite ne porte pas de déclencheur explicite (*i.e.* s'il s'agit d'une transition d'achèvement), elle est franchissable quand l'état final de l'état composite est atteint.

Les transitions peuvent également toucher des états de différents niveaux d'imbrication en traversant les frontières des états composites.

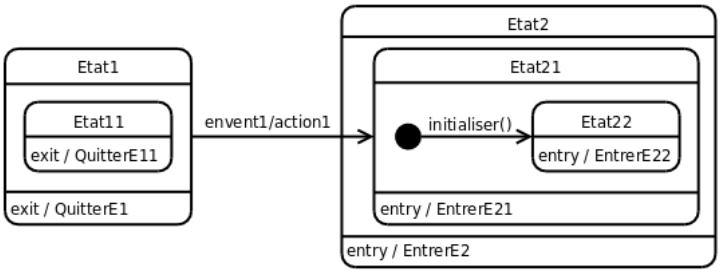



Figure 5.13: Exemple de configuration complexe de transition. Depuis l'état *État 1*, la réception de l'événement *event1* produit la séquence d'activités *QuitterE11*, *QuitterE1*, *action1*, *EntrerE2*, *EntrerE21*, *initialiser()*, *EntrerE22*, et place le système dans l'état *État22*.

La figure 5.13 illustre une configuration complexe de transition produisant une cascade d'activités.

### 5.6.3 État historique

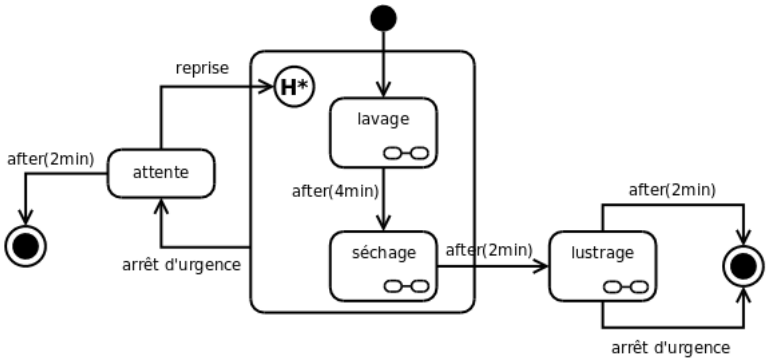



Figure 5.14: Exemple de diagramme possédant un état historique profond permettant de reprendre le programme de lavage ou de séchage d'une voiture à l'endroit où il était arrivé avant d'être interrompu.

Un état historique, également qualifié d'*état historique plat*, est un pseudo-état qui mémorise le dernier sous-état actif d'un état composite. Graphiquement, il est représenté par un cercle contenant un *H*.

Une transition ayant pour cible l'état historique est équivalente à une transition qui a pour cible le dernier état visité de l'état englobant. Un état historique peut avoir une transition sortante non étiquetée indiquant l'état à exécuter si la région n'a pas encore été visitée.

Il est également possible de définir un *état historique profond* représenté graphiquement par un cercle contenant un *H**. Cet état historique profond permet d'atteindre le dernier état visité dans la région, quel que soit sont niveau d'imbrication, alors que le l'état historique plat limite l'accès aux états de son niveau d'imbrication.

La figure 5.14 montre un diagramme d'états-transitions modélisant le lavage automatique d'une voiture. Les états de *lavage*, *séchage* et *lustrage* sont des états composites définis sur trois autres diagrammes d'états-transitions non représentés ici. En phase de lavage ou de séchage, le client peut appuyer sur le bouton d'arrêt d'urgence. S'il appuie sur ce bouton, la machine se met en attente. Il a alors deux minutes pour reprendre le lavage ou le lustrage, exactement où le programme à été interrompue, c'est à dire au niveau du dernier sous-état actif des états de lavage ou de lustrage (état historique profond). Si l'état avait été un état historique plat, c'est toute la séquence de lavage ou de lustrage qui aurait recommencée. En phase de lustrage, le client peut aussi interrompre la machine. Mais dans ce cas, la machine s'arrêtera définitivement.

### 5.6.4 Interface : les points de connexion

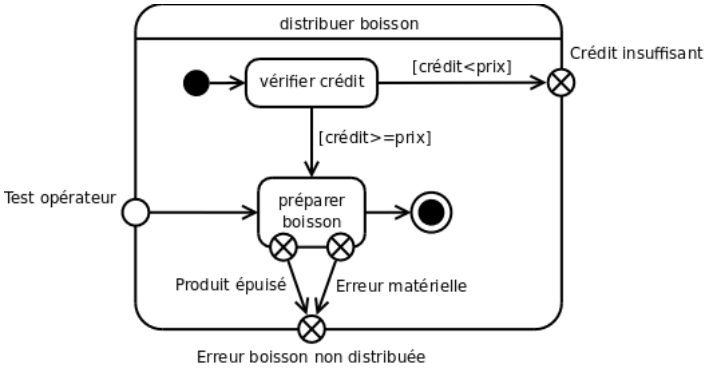



Figure 5.15: Exemple d'utilisation de points de connexion.

Comme nous l'avons déjà dit, il est possible de masquer les sous-états d'un état composite et de les définir dans un autre diagramme. Cette pratique nécessite parfois l'utilisation de pseudo-états appelés *points de connexion*.

Lorsque l'on utilise le comportement par défaut de l'état composite, c'est-à-dire entrer par l'état initial par défaut et considérer les traitements finis quand l'état final est atteint, aucun problème ne se pose car on utilise des transitions ayant pour cible, ou pour source, la frontière de l'état composite. Dans ce cas, les points de connexion sont inutiles.

Le problème se pose lorsqu'il est possible d'entrer ou de sortir d'un état composite de plusieurs façons. C'est, par exemple, le cas lorsqu'il existe des transitions traversant la frontière de l'état composite et visant directement, ou ayant pour source, un sous-état de l'état composite. Dans ce cas, la solution est d'utiliser des points de connexion sur la frontière de l'état composite.

Les points de connexion sont des points d'entrée et de sortie portant un nom, et situés sur la frontière d'un état composite. Ils sont respectivement représentés par un cercle vide et un cercle barré d'une croix (cf. figure 5.15). Il ne s'agit que de références à un état défini dans l'état composite . Une unique transition d'achèvement, dépourvue de garde, relie le pseudo-état source (*i.e.* le point de connexion) à l'état référencé. Cette transition d'achèvement n'est que le prolongement de la transition qui vise le point de connexion (il peut d'ailleurs y en avoir plusieurs). Les points de connexions offrent ainsi une façon de représenter l'interface (au sens objet) d'un état composite en masquant l'implémentation de son comportement.

On peut considérer que les pseudo-états initiaux et finaux sont des points de connexion sans nom.

### 5.6.5 Concurrence

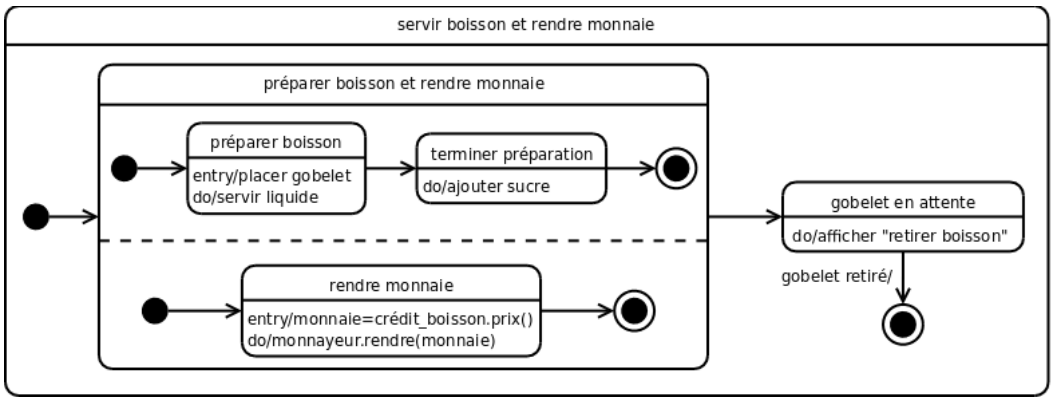



Figure 5.16: Exemple d'utilisation d'un état composite orthogonal.

Les diagrammes d'états-transitions permettent de décrire efficacement les mécanismes concurrents grâce à l'utilisation d'*états orthogonaux*. Un état orthogonal est un état composite comportant plus d'une région, chaque région représentant un flot d'exécution. Graphiquement, dans un état orthogonal, les différentes régions sont séparées par un trait horizontal en pointillé allant du bord gauche au bord droit de l'état composite.

Chaque région peut posséder un état initial et final. Une transition qui atteint la bordure d'un état composite orthogonal est équivalente à une transition qui atteint les états initiaux de toutes ses régions concurrentes.

Toutes les régions concurrentes d'un état composite orthogonal doivent atteindre leur état final pour que l'état composite soit considéré comme terminé.

La figure 5.16 illustre l'utilisation d'un état composite orthogonal pour modéliser le fait que la préparation de la boisson d'un distributeur de boisson se fait en parallèle au rendu de la monaie.

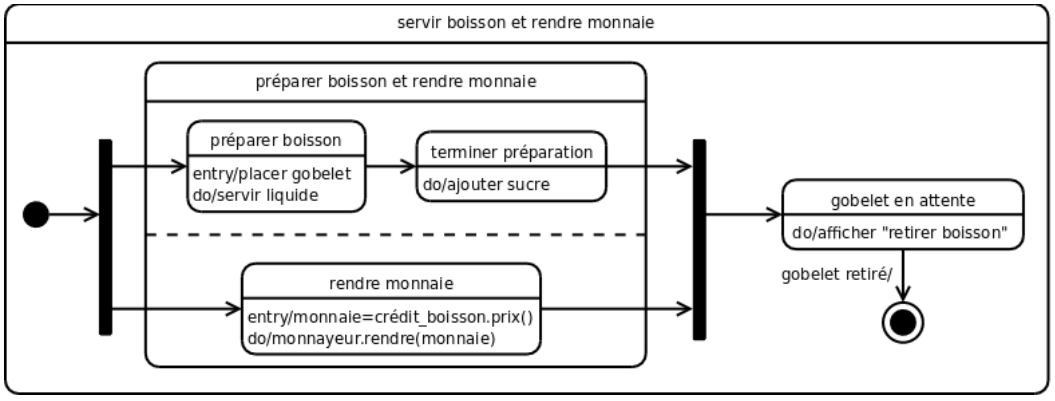



Figure 5.17: Exemple d'utilisation de transitions complexes.

Il est également possible de représenter ce type de comportement au moyen de transitions concurrentes. De telles transitions sont qualifiées de *complexes*. Les transitions complexes sont représentées par une barre épaisse et peuvent, éventuellement, être nommées. La figure 5.17 montre la mise en œuvre de ce type de transition. Sur ce diagramme, l'état orthogonal *préparer boisson et rendre monnaie* peut éventuellement ne pas apparaître (tout en gardant la représentation de ses sous-états) pour alléger la représentation, car la notion de concurrence est clairement apparente de par l'utilisation des transitions complexes.

# Chapitre 6 Diagramme d'activités *(Activity diagram)*

## 6.1 Introduction au formalisme

### 6.1.1 Présentation

Les diagrammes d'activités permettent de mettre l'accent sur les traitements. Ils sont donc particulièrement adaptés à la modélisation du cheminement de flots de contrôle et de flots de données. Ils permettent ainsi de représenter graphiquement le comportement d'une méthode ou le déroulement d'un cas d'utilisation.

Les diagrammes d'activités sont relativement proches des diagrammes d'états-transitions dans leur présentation, mais leur interprétation est sensiblement différente. Les diagrammes d'états-transitions sont orientés vers des systèmes réactifs, mais ils ne donnent pas une vision satisfaisante d'un traitement faisant intervenir plusieurs classeurs et doivent être complétés, par exemple, par des diagrammes de séquence. Au contraire, les diagrammes d'activités ne sont pas spécifiquement rattachés à un classeur particulier. On peut attacher un diagramme d'activités à n'importe quel élément de modélisation afin de visualiser, spécifier, construire ou documenter le comportement de cet élément.

La différence principale entre les diagrammes d'interaction et les diagrammes d'activités est que les premiers mettent l'accent sur le flot de contrôle d'un objet à l'autre, tandis que les seconds insistent sur le flot de contrôle d'une activité à l'autre.

### 6.1.2 Utilisation courante

Dans la phase de conception, les diagrammes d'activités sont particulièrement adaptés à la description des cas d'utilisation. Plus précisément, ils viennent illustrer et consolider la description textuelle des cas d'utilisation (cf. section 2.5.3). De plus, leur représentation sous forme d'organigrammes les rend facilement intelligibles et beaucoup plus accessibles que les diagrammes d'états-transitions. On parle généralement dans ce cas de modélisation de *workflow*. On se concentre ici sur les activités telles que les voient les acteurs qui collaborent avec le système dans le cadre d'un processus métier. La modélisation du flot d'objets est souvent importante dans ce type d'utilisation des diagrammes d'activités.

Les diagrammes d'activités permettent de spécifier des traitements *a priori* séquentiels et offrent une vision très proche de celle des langages de programmation impératifs comme C++ ou Java. Ainsi, ils peuvent être utiles dans la phase de réalisation car ils permettent une description si précise des opérations qu'elle autorise la génération automatique du code. La modélisation d'une opération peut toutefois être assimilé à une utilisation d'UML comme langage de programmation visuelle, ce qui n'est pas sa finalité. Il ne faut donc pas y avoir recours de manière systématique mais la réserver à des opérations dont le comportement est complexe ou sensible.

## 6.2 Activité et Transition

### 6.2.1 Action (*action*)

Une action est le plus petit traitement qui puisse être exprimé en UML. Une action a une incidence sur l'état du système ou en extrait une information. Les actions sont des étapes discrètes à partir desquelles se construisent les comportements. La notion d'action est à rapprocher de la notion d'instruction élémentaire d'un langage de programmation (comme C++ ou Java). Une action peut être, par exemple :

- une affectation de valeur à des attributs ;
- un accès à la valeur d'une propriété structurelle (attribut ou terminaison d'association) ;
- la création d'un nouvel objet ou lien ;
- un calcul arithmétique simple ;
- l'émission d'un signal ;
- la réception d'un signal ;
- …

Nous décrivons ci-dessous les types d'actions les plus courants prédéfinis dans la notation UML.

**Action appeler (*call operation*) –**
L'action *call operation* correspond à l'invocation d'une opération sur un objet de manière synchrone ou asynchrone. Lorsque l'action est exécutée, les paramètres sont transmis à l'objet cible. Si l'appel est asynchrone, l'action est terminée et les éventuelles valeurs de retour seront ignorées. Si l'appel est synchrone, l'appelant est bloqué pendant l'exécution de l'opération et, le cas échéant, les valeurs de retour pourront être réceptionnées.

**Action comportement (*call behavior*) –**
L'action *call behavior* est une variante de l'action *call operation* car elle invoque directement une activité plutôt qu'une opération.

**Action envoyer (*send*) –**
Cette action crée un message et le transmet à un objet cible, où elle peut déclencher un comportement. Il s'agit d'un appel asynchrone (*i.e.* qui ne bloque pas l'objet appelant) bien adapté à l'envoi de signaux (*send signal*).

**Action accepter événement (*accept event*) –**
L'exécution de cette action bloque l'exécution en cours jusqu'à la réception du type d'événement spécifié, qui généralement est un signal. Cette action est utilisée pour la réception de signaux asynchrones.

**Action accepter appel (*accept call*) –**
Il s'agit d'une variante de l'action *accept event* pour les appels synchrones.

**Action répondre (*reply*) –**
Cette action permet de transmettre un message en réponse à la réception d'une action de type *accept call*.

**Action créer (*create*) –**
Cette action permet d'instancier un objet.

**Action détruire (*destroy*) –**
Cette action permet de détruire un objet.

**Action lever exception (*raise exception*) –**
Cette action permet de lever explicitement une exception.

Graphiquement, les actions apparaissent dans des nœuds d'action, décrits section 6.3.1.

### 6.2.2 Activité (*activity*)

Une activité définit un comportement décrit par un séquencement organisé d'unités dont les éléments simples sont les actions. Le flot d'exécution est modélisé par des nœuds reliés par des arcs (transitions). Le flot de contrôle reste dans l'activité jusqu'à ce que les traitements soient terminés.

Une activité est un comportement (*behavior* en anglais) et à ce titre peut être associée à des paramètres.

### 6.2.3 Groupe d'activités (*activity group*)

Un groupe d'activités est une activité regroupant des nœuds et des arcs. Les nœuds et les arcs peuvent appartenir à plus d'un groupe. Un diagramme d'activités est lui-même un groupe d'activités (cf. figure 6.2).

### 6.2.4 Nœud d'activité (*activity node*)

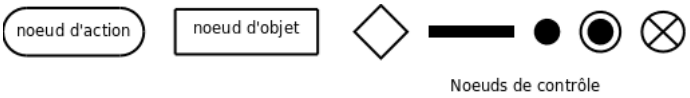



Figure 6.1: Représentation graphique des nœuds d'activité. De la gauche vers la droite, on trouve : le nœud représentant une action, qui est une variété de nœud exécutable, un nœud objet, un nœud de décision ou de fusion, un nœud de bifurcation ou d'union, un nœud initial, un nœud final et un nœud final de flot.

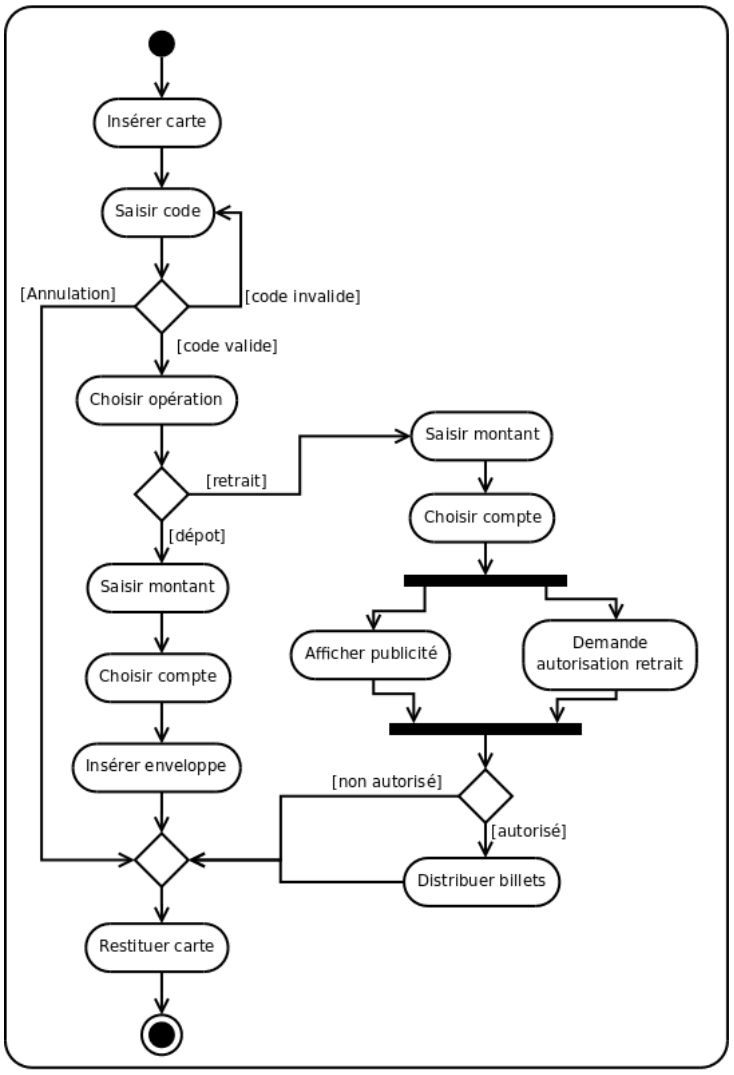



Figure 6.2: Exemple de diagramme d'activités modélisant le fonctionnement d'une borne bancaire.

Un nœud d'activité est un type d'élément abstrait permettant de représenter les étapes le long du flot d'une activité. Il existe trois familles de nœuds d'activités :

- les nœuds d'exécutions (*executable node* en anglais) ;
- les nœuds objets (*object node* en anglais) ;
- et les nœuds de contrôle (*control nodes* en anglais).

La figure 6.1 représente les différents types de nœuds d'activité. La figure 6.2 montre comment certains de ces nœuds sont utilisés pour former un diagramme d'activités.

### 6.2.5 Transition


Figure 6.3: Représentation graphique d'une transition.

Le passage d'une activité vers une autre est matérialisé par une transition. Graphiquement les transitions sont représentées par des flèches en traits pleins qui connectent les activités entre elles (figure 6.3). Elles sont déclenchées dès que l'activité source est terminée et provoquent automatiquement et immédiatement le début de la prochaine activité à déclencher (l'activité cible). Contrairement aux activités, les transitions sont franchies de manière atomique, en principe sans durée perceptible.

Les transitions spécifient l'enchaînement des traitements et définissent le flot de contrôle.

## 6.3 Nœud exécutable (*executable node*)

Un nœud exécutable est un nœud d'activité qu'on peut exécuter (*i.e.* une activité). Il possède un gestionnaire d'exception qui peut capturer les exceptions levées par le nœud, ou

un de ses nœuds imbriqués.

### 6.3.1 Nœud d'action

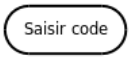



Figure 6.4: Représentation graphique d'un nœud d'action.

Un nœud d'action est un nœud d'activité exécutable qui constitue l'unité fondamentale de fonctionnalité exécutable dans une activité. L'exécution d'une action représente une transformation ou un calcul quelconque dans le système modélisé. Les actions sont généralement liées à des opérations qui sont directement invoquées. Un nœud d'action doit avoir au moins un arc entrant.

Graphiquement, un nœud d'action est représenté par un rectangle aux coins arrondis (figure 6.4) qui contient sa description textuelle. Cette description textuelle peut aller d'un simple nom à une suite d'actions réalisées par l'activité. UML n'impose aucune syntaxe pour cette description textuelle, on peut donc utiliser une syntaxe proche de celle d'un langage de programmation particulier ou du pseudo-code.

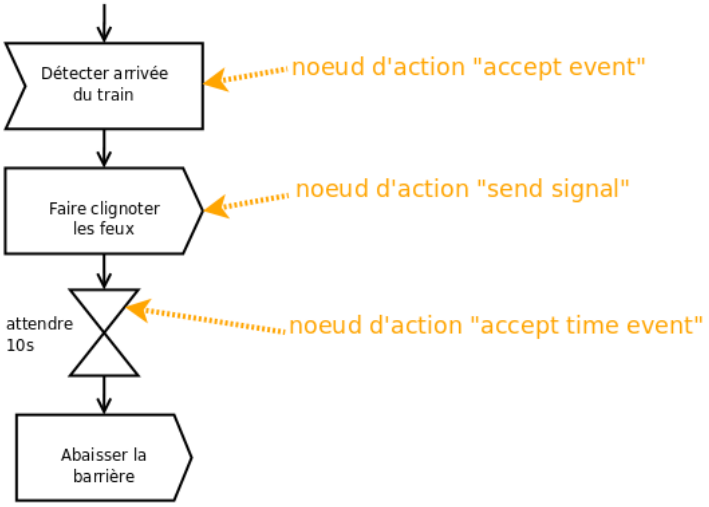



Figure 6.5: Représentation particulière des nœuds d'action de communication.

Certaines actions de communication ont une notation spéciale (cf. figure 6.5).

### 6.3.2 Nœud d'activité structurée (*structured activity node*)

Un nœud d'activité structurée est un nœud d'activité exécutable qui représente une portion structurée d'une activité donnée qui n'est partagée avec aucun autre nœud structuré, à l'exception d'une imbrication éventuelle.

Les transitions d'une activité structurée doivent avoir leurs nœuds source et cible dans le même nœud d'activité structurée. Les nœuds et les arcs contenus par nœud d'activité structuré ne peuvent pas être contenus dans un autre nœud d'activité structuré.

Un nœud structuré est dénoté par le stéréotype *« structured »* et identifié par un nom unique décrivant le comportement modélisé dans l'activité structurée.

Graphiquement, le contour d'un nœud d'activité structurée est en pointillé. Une ligne horizontale en trait continu sépare le compartiment contenant le stéréotype *« structured »* et le nom de l'activité structurée du corps de l'activité structurée.

## 6.4 Nœud de contrôle (*control node*)

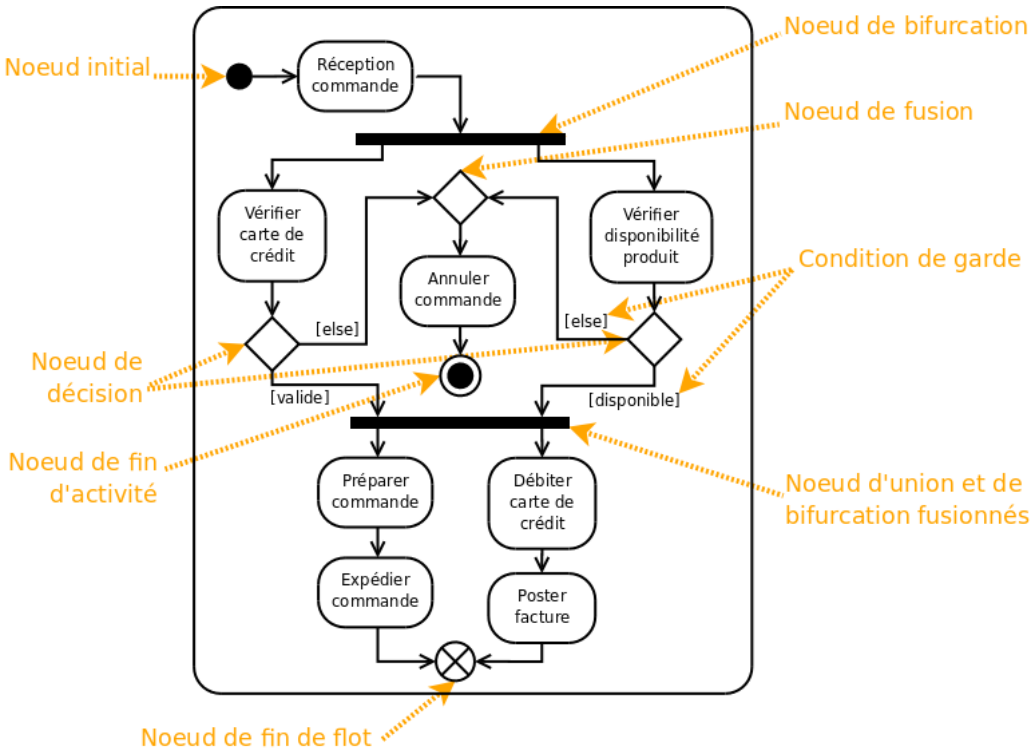



Figure 6.6: Exemple de diagramme d'activité illustrant l'utilisation de nœuds de contrôle. Ce diagramme décrit la prise en compte d'une commande.

---

Un nœud de contrôle est un nœud d'activité abstrait utilisé pour coordonner les flots entre les nœuds d'une activité.

Il existe plusieurs types de nœuds de contrôle :

- nœud initial (*initial node* en anglais) ;
- nœud de fin d'activité (*final node* en anglais)
- nœud de fin de flot (*flow final* en anglais) ;
- nœud de décision (*decision node* en anglais) ;
- nœud de fusion (*merge node* en anglais) ;
- nœud de bifurcation (*fork node* en anglais) ;
- nœud d'union (*join node* en anglais).

La figure 6.6 illustre l'utilisation de ces nœuds de contrôle.

### 6.4.1 Nœud initial

Un nœud initial est un nœud de contrôle à partir duquel le flot débute lorsque l'activité enveloppante est invoquée. Une activité peut avoir plusieurs nœuds initiaux. Un nœud initial possède un arc sortant et pas d'arc entrant.

Graphiquement, un nœud initial est représenté par un petit cercle plein (cf. figure 6.6).

### 6.4.2 Nœud final

Un nœud final est un nœud de contrôle possédant un ou plusieurs arcs entrants et aucun arc sortant.

#### Nœud de fin d'activité

Lorsque l'un des arcs d'un nœud de fin d'activité est activé (*i.e.* lorsqu'un flot d'exécution atteint un nœud de fin d'activité), l'exécution de l'activité enveloppante s'achève et tout nœud ou flot actif au sein de l'activité enveloppante est abandonné. Si l'activité a été invoquée par un appel synchrone, un message (*reply*) contenant les valeurs sortantes est transmis en retour à l'appelant.

Graphiquement, un nœud de fin d'activité est représenté par un cercle vide contenant un petit cercle plein (cf. figure 6.6).

#### Nœud de fin de flot

Lorsqu'un flot d'exécution atteint un nœud de fin de flot, le flot en question est terminé, mais cette fin de flot n'a aucune incidence sur les autres flots actifs de l'activité enveloppante.

Graphiquement, un nœud de fin de flot est représenté par un cercle vide barré d'un *X*.

Les nœuds de fin de flot sont particuliers et à utiliser avec parcimonie. Dans l'exemple de la figure 6.6, le nœud de fin de flot n'est pas indispensable : on peut le remplacer par un nœud d'union possédant une transition vers un nœud de fin d'activité.

### 6.4.3 Nœud de décision et de fusion

#### Nœud de décision (*decision node*)

Un nœud de décision est un nœud de contrôle qui permet de faire un choix entre plusieurs flots sortants. Il possède un arc entrant et plusieurs arcs sortants. Ces derniers sont généralement accompagnés de conditions de garde pour conditionner le choix. Si, quand le nœud de décision est atteint, aucun arc en aval n'est franchissable (*i.e.* aucune condition de garde n'est vraie), c'est que le modèle est mal formé. L'utilisation d'une garde `[else]` est recommandée après un nœud de décision car elle garantit un modèle bien formé. En effet, la condition de garde `[else]` est validée si et seulement si toutes les autres gardes des transitions ayant la même source sont fausses. Dans le cas où plusieurs arcs sont franchissables (*i.e.* plusieurs conditions de garde sont vraies), seul l'un d'entre eux est retenu et ce choix est non déterministe.

Graphiquement, on représente un nœud de décision par un losange (cf. figure 6.6).

**Nœud de fusion (*merge node*)**

Un nœud de fusion est un nœud de contrôle qui rassemble plusieurs flots alternatifs entrants en un seul flot sortant. Il n'est pas utilisé pour synchroniser des flots concurrents (c'est le rôle du nœud d'union) mais pour accepter un flot parmi plusieurs.

Graphiquement, on représente un nœud de fusion, comme un nœud de décision, par un losange (cf. figure 6.6).

**Remarque**

Graphiquement, il est possible de fusionner un nœud de fusion et un nœud de décision, et donc d'avoir un losange possédant plusieurs arcs entrants et sortants. Il est également possible de fusionner un nœud de décision ou de fusion avec un autre nœud, comme un nœud de fin de flot sur la figure 6.6, ou avec une activité. Cependant, pour mieux mettre en évidence un branchement conditionnel, il est préférable d'utiliser un nœud de décision (losange).

### 6.4.4 Nœud de bifurcation et d'union

**Nœud de bifurcation ou de débranchement (*fork node*)**

Un nœud de bifurcation, également appelé nœud de débranchement est un nœud de contrôle qui sépare un flot en plusieurs flots concurrents. Un tel nœud possède donc un arc entrant et plusieurs arcs sortants. On apparie généralement un nœud de bifurcation avec un nœud d'union pour équilibrer la concurrence (cf. figure 6.2).

Graphiquement, on représente un nœud de bifurcation par un trait plein (cf. figure 6.6).

**Nœud d'union ou de jointure (*join node*)**

Un nœud d'union, également appelé nœud de jointure est un nœud de contrôle qui synchronise des flots multiples. Un tel nœud possède donc plusieurs arcs entrants et un seul arc sortant. Lorsque tous les arcs entrants sont activés, l'arc sortant l'est également.

Graphiquement, on représente un nœud de union, comme un nœud de bifurcation, par un trait plein (cf. figure 6.2).

**Remarque**

Graphiquement, il est possible de fusionner un nœud de bifurcation et un nœud d'union, et donc d'avoir un trait plein possédant plusieurs arcs entrants et sortants (cf. figure 6.6).

## 6.5 Nœud d'objet (*object node*)

### 6.5.1 Introduction

Jusqu'ici, nous avons montré comment modéliser le comportement du flot de contrôle dans un diagramme d'activités. Or, les flots de données n'apparaissent pas et sont pourtant un élément essentiel des traitements (arguments des opérations, valeurs de retour, …).

Justement, un nœud d'objet permet de définir un flot d'objet (*i.e.* un flot de données) dans un diagramme d'activités. Ce nœud représente l'existence d'un objet généré par une action dans une activité et utilisé par d'autres actions.

### 6.5.2 Pin d'entrée ou de sortie

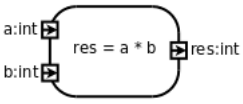



Figure 6.7: Représentation des pins d'entrée et de sortie sur une activité.

Pour spécifier les valeurs passées en argument à une activité et les valeurs de retour, on utilise des nœuds d'objets appelés pins (*pin* en anglais) d'entrée ou de sortie. L'activité ne peut débuter que si l'on affecte une valeur à chacun de ses pins d'entrée. Quand l'activité se termine, une valeur doit être affectée à chacun de ses pins de sortie.

Les valeurs sont passées par copie : une modification des valeurs d'entrée au cours du traitement de l'action n'est visible qu'à l'intérieur de l'activité.

Graphiquement, un pin est représenté par un petit carré attaché à la bordure d'une activité (cf. figure 6.7). Il est typé et éventuellement nommé. Il peut contenir des flèches indiquant sa direction (entrée ou sortie) si l'activité ne permet pas de le déterminer de manière univoque.

### 6.5.3 Pin de valeur (*value pin*)

Un pin valeur est un pin d'entrée qui fournit une valeur à une action sans que cette valeur ne provienne d'un arc de flot d'objets. Un pin valeur est toujours associé à une valeur spécifique.

Graphiquement, un pin de valeur se représente comme un pin d'entrée avec la valeur associée écrite à proximité.

### 6.5.4 Flot d'objet

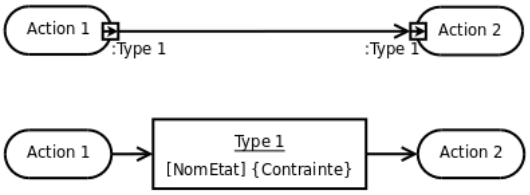



Figure 6.8: Deux notations possibles pour modéliser un flot de données.

Un flot d'objets permet de passer des données d'une activité à une autre. Un arc reliant un pin de sortie à un pin d'entrée est, par définition même des pins, un flot d'objets (en haut de la figure 6.8). Dans cette configuration, le type du pin récepteur doit être identique ou parent (au sens de la relation de généralisation) du type du pin émetteur.

Il existe une autre représentation possible d'un flot d'objets, plus axée sur les données proprement dites car elle fait intervenir un nœud d'objet détaché d'une activité particulière (en bas de la figure 6.8). Graphiquement, un tel nœud d'objet est représenté par un rectangle dans lequel est mentionné le type de l'objet (souligné). Des arcs viennent ensuite relier ce nœud d'objet à des activités sources et cibles. Le nom d'un état, ou d'une liste d'états, de l'objet peut être précisé entre crochets après ou sous le type de l'objet. On peut également préciser des contraintes entre accolades, soit à l'intérieur, soit en dessous du rectangle du nœud d'objet.

La figure 6.11 montre l'utilisation de nœuds d'objets dans un diagramme d'activités.

Un flot d'objets peut porter une étiquette stéréotypée mentionnant deux comportements particuliers :

- *«transformation»* indique une interprétation particulière de la donnée véhiculée par le flot ;
- *«selection»* indique l'ordre dans lequel les objets sont choisis dans le nœud pour le quitter (cf. figure 6.10).

### 6.5.5 Nœud tampon central (*central buffer node*)

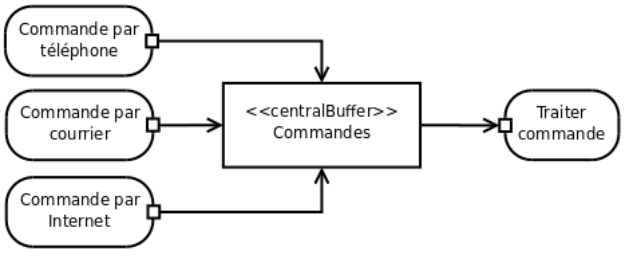



Figure 6.9: Exemple d'utilisation d'un nœud tampon central pour centraliser toutes les commandes prises par différents procédés, avant qu'elles soient traitées.

Un nœud tampon central est un nœud d'objet qui accepte les entrées de plusieurs nœuds d'objets ou produit des sorties vers plusieurs nœuds d'objets. Les flots en provenance d'un nœud tampon central ne sont donc pas directement connectés à des actions. Ce nœud modélise donc un tampon traditionnel qui peut contenir des valeurs en provenance de diverses sources et livrer des valeurs vers différentes destinations.

Graphiquement, un nœud tampon central est représenté comme un nœud d'objet détaché (en bas de la figure 6.8) stéréotypé *«centralBuffer»* (cf. figure 6.9).

### 6.5.6 Nœud de stockage des données (*data store node*)

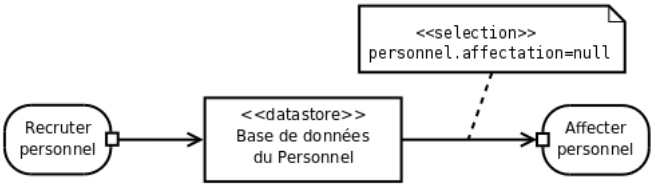



Figure 6.10: Dans cette modélisation, le personnel, après avoir été recruté par l'activité *Recruter personnel*, est stocké de manière persistante dans le nœud de stockage *Base de donnée du Personnel*. Bien qu'ils restent dans ce nœud, chaque employé qui n'a pas encore reçu d'affectation (étiquette stéréotypée *«selection»* : `personnel.affectation=null`) est disponible pour être utilisé par l'activité *Affecter personnel*.

Un nœud de stockage des données est un nœud tampon central particulier qui assure la persistance des données. Lorsqu'une information est sélectionnée par un flux sortant, l'information est dupliquée et ne disparaît pas du nœud de stockage des données comme ce serait le cas dans un nœud tampon central. Lorsqu'un flux entrant véhicule une donnée déjà stockée par le nœud de stockage des données, cette dernière est écrasée par la nouvelle.

Graphiquement, un nœud tampon central est représenté comme un nœud d'objet détaché (en bas de la figure 6.8) stéréotypé *«datastore»* (cf. figure 6.10).

## 6.6 Partitions

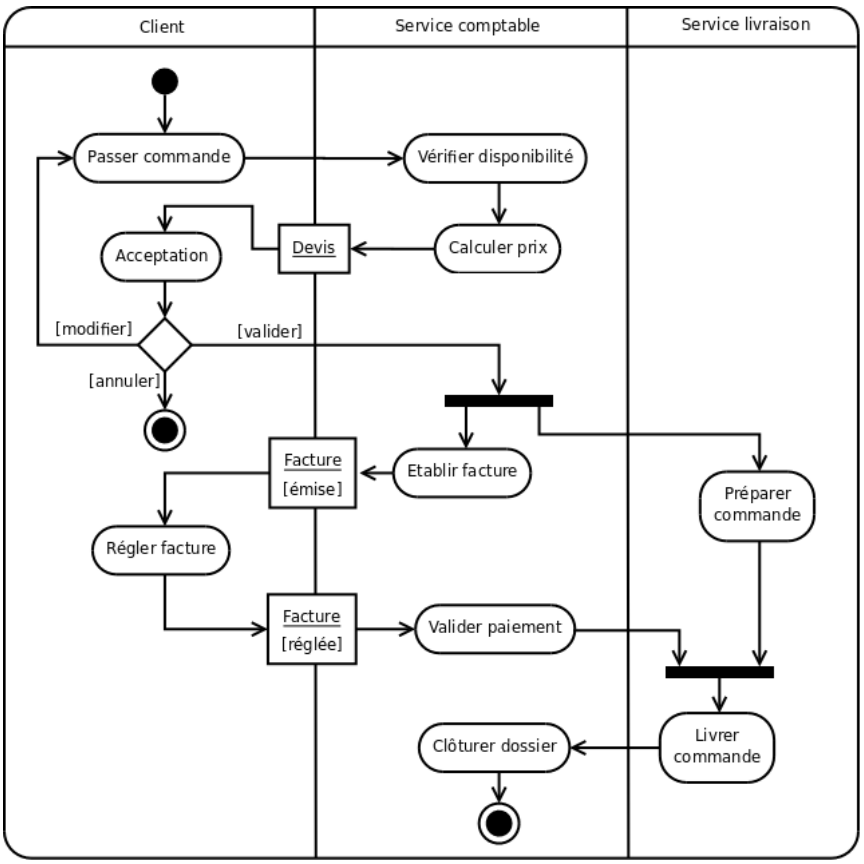



Figure 6.11: Illustration de l'utilisation de nœuds d'objets et de partitions dans un diagramme d'activités.

Les partitions, souvent appelées couloirs ou lignes d'eau (*swimlane*) du fait de leur notation, permettent d'organiser les nœuds d'activités dans un diagramme d'activités en opérant des regroupements (cf. figure 6.11).

Les partitions n'ont pas de signification bien arrêtée, mais correspondent souvent à des unités d'organisation du modèle. On peut, par exemple, les utiliser pour spécifier la classe responsable de la mise en œuvre d'un ensemble tâche. Dans ce cas, la classe en question est responsable de l'implémentation du comportement des nœuds inclus dans ladite partition.

Graphiquement, les partitions sont délimitées par des lignes continues. Il s'agit généralement de lignes verticales, comme sur la figure 6.11, mais elle peuvent être horizontales ou même courbes. Dans la version 2.0 d'UML, les partitions peuvent être bidimensionnelles, elles prennent alors la forme d'un tableau. Dans le cas d'un diagramme d'activités partitionné, les nœuds d'activités appartiennent forcément à une et une seule partition. Les transitions peuvent, bien entendu, traverser les frontières des partitions.

Les partitions d'activités étant des catégories arbitraires, on peut les représenter par d'autre moyens quand une répartition géométrique s'avère difficile à réaliser. On peut ainsi utiliser des couleurs ou tout simplement étiqueter les nœuds d'activité par le nom de leur partition d'appartenance.

## 6.7 Exceptions

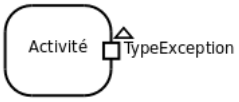



Figure 6.12: Notation graphique du fait qu'une activité peut soulever une exception.

Une exception est générée quand une situation anormale entrave le déroulement nominal d'une tâche. Elle peut être générée automatiquement pour signaler une erreur d'exécution (débordement d'indice de tableau, division par zéro, …), ou être soulevée explicitement par une action (*RaiseException*) pour signaler une situation problématique qui n'est pas prise en charge par la séquence de traitement normale. Graphiquement, on peut représenter le fait qu'une activité peut soulever une exception comme un pin de sortie orné d'un petit triangle et en précisant le type de l'exception à proximité du pin de sortie (cf. figure 6.12).

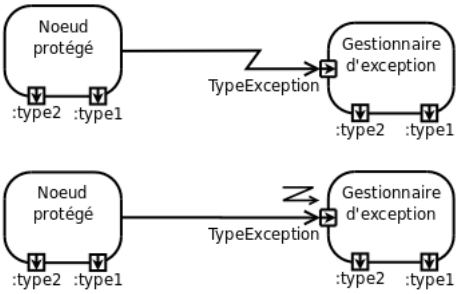



Figure 6.13: Les deux notations graphiques de la connexion entre une activité protégée et son gestionnaire d'exception associé.

Un gestionnaire d'exception est une activité possédant un pin d'entrée du type de l'exception qu'il gère et lié à l'activité qu'il protège par un arc en zigzag ou un arc classique

orné d'une petite flèche en zigzag. Le gestionnaire d'exception doit avoir les mêmes pins de sortie que le bloc qu'il protège (cf. figure 6.13).

Les exceptions sont des classeurs et, à ce titre, peuvent posséder des caractéristiques comme des attributs ou des opérations. Il est également possible d'utiliser la relation d'héritage sur les exceptions. Un gestionnaire d'exception spécifie toujours le type des exceptions qu'il peut traiter, toute exception dérivant de ce type est donc également prise en charge.

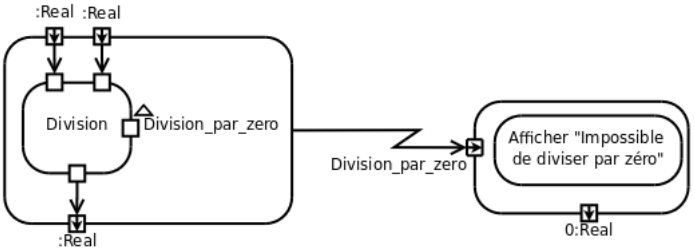



Figure 6.14: Exemple d'utilisation d'un gestionnaire d'exception pour protéger une activité de l'exception *Division_par_zero* déclenchée en cas de division par zéro.

Lorsqu'une exception survient, l'exécution de l'activité en cours est abandonnée sans générer de valeur de sortie. Le mécanisme d'exécution recherche alors un gestionnaire d'exception susceptible de traiter l'exception levée ou une de ses classes parentes. Si l'activité qui a levé l'exception n'est pas protégée de cette exception, l'exception est propagée à l'activité englobante. L'exécution de cette dernière est abandonnée, ses valeurs de sortie ne sont pas générées et un gestionnaire d'exception est recherché à son niveau. Ce mécanisme de propagation se poursuit jusqu'à ce qu'un gestionnaire adapté soit trouvé. Si l'exception se propage jusqu'au sommet d'une activité (*i.e.* il n'y a plus d'activité englobante), trois cas de figure se présentent. Si l'activité a été invoquée de manière asynchrone, aucun effet ne se produit et la gestion de l'exception est terminée. Si l'activité a été invoquée de manière synchrone, l'exception est propagée au mécanisme d'exécution de l'appelant. Si l'exception s'est propagée à la racine du système, le modèle est considéré comme incomplet ou mal formé. Dans la plupart des langages orientés objet, une exception qui se propage jusqu'à la racine du programme implique son arrêt. Quand un gestionnaire d'exception adapté a été trouvé et que son exécution se termine, l'exécution se poursuit comme si l'activité protégée s'était terminée normalement, les valeurs de sortie fournies par le gestionnaire remplaçant celle que l'activité protégée aurait dû produire.

# Chapitre 7 Diagrammes d'interaction *(Interaction diagram)*

## 7.1 Présentation du formalisme

Porte de garage motorisée à enroulement

### 7.1.1 Introduction

Un objet interagit pour implémenter un comportement. On peut décrire cette interaction de deux manières complémentaires : l'une est centrée sur des objets individuels (diagramme d'états-transitions) et l'autre sur une collection d'objets qui coopèrent (diagrammes d'interaction).

La spécification d'un diagramme d'états-transitions est précise et conduit immédiatement au code. Elle ne permet pas pour autant d'expliquer le fonctionnement global d'un système, car elle se concentre sur un seul objet à la fois. Un diagramme d'interaction permet d'offrir une vue plus holistique[1] du comportement d'un jeu d'objets.

Le diagramme de communication (section 7.2) est un diagramme d'interaction mettant l'accent sur l'organisation structurelle des objets qui envoient et reçoivent des messages. Le diagramme de séquence (section 7.3) est un diagramme d'interaction mettant l'accent sur la chronologie de l'envoi des messages. Les diagrammes d'interaction permettent d'établir un lien entre les diagrammes de cas d'utilisation et les diagrammes de classes : ils montrent comment des objets (*i.e.* des instances de classes) communiquent pour réaliser une certaine fonctionnalité. Ils apportent ainsi un aspect dynamique à la modélisation du système.

Pour produire un diagramme d'interaction, il faut focaliser son attention sur un sous-ensemble d'éléments du système et étudier leur façon d'interagir pour décrire un comportement particulier. UML permet de décrire un comportement limité à un contexte précis de deux façons : dans le cadre d'un classeur structuré (cf. section 7.1.2) ou dans celui d'une collaboration (cf. section 7.1.3).

### 7.1.2 Classeur structuré

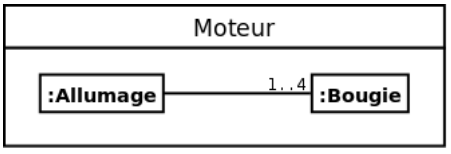



Figure 7.1: Exemple de classeur structuré montrant qu'un classeur *Moteur* est en fait constitué d'un *Allumage* et de quatre *Bougie*.

Les classes découvertes au moment de l'analyse (celles qui figurent dans le diagramme de classes) ne sont parfois pas assez détaillées pour pouvoir être implémentées par des développeurs. UML propose de partir des classeurs découverts au moment de l'analyse (tels que les classes, mais aussi les sous-systèmes, les cas d'utilisation, …) et de les décomposer en éléments suffisamment fins pour permettre leur implémentation. Les classeurs ainsi décomposés s'appellent des classeurs structurés.

Un classeur structuré est donc la description de la structure d'implémentation interne d'une classe. Graphiquement, un classeur structuré se représente par un rectangle en trait plein comprenant deux compartiments. Le compartiment supérieur contient le nom du classeur et le compartiment inférieur montre les parties internes reliées par des connecteurs (cf. figure 7.1).

Un classeur structuré possède des ports (cf. section 8.2.3), des parties et des connecteurs. Lorsque l'on crée l'instance d'un classeur structuré, on crée également une instance de ses ports, de ses parties et de ses connecteurs.

### 7.1.3 Collaboration

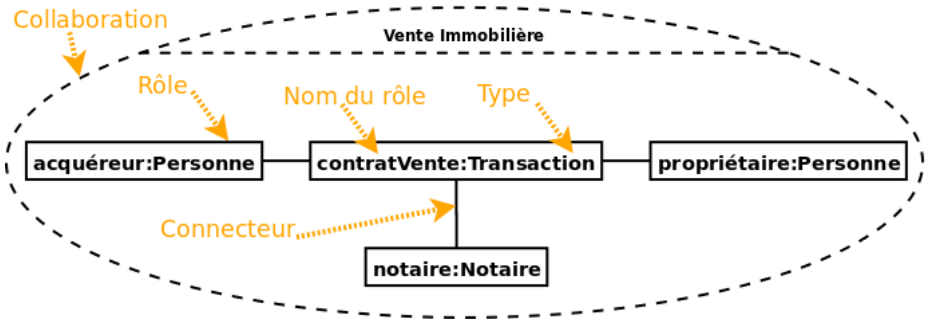



Figure 7.2: Diagramme de collaboration d'une transaction immobilière.

Une collaboration permet de décrire la mise en œuvre d'une fonctionnalité par un jeu de participants. Un rôle est la description d'un participant. Contrairement aux paquetages et aux classeurs structurés, une collaboration ne détient pas les instances liées à ses rôles. Les instances existent avant l'établissement d'une instance de la collaboration, mais la collaboration les rassemble et précise des connecteurs entre elles. Une collaboration peut donc traverser plusieurs niveaux d'un système et un même élément peut apparaître dans plusieurs collaborations.

Par exemple, pour implémenter un cas d'utilisation, il faut utiliser un ensemble de classes, et d'autres éléments, fonctionnant ensemble pour réaliser le comportement de ce cas d'utilisation. Cette ensemble d'éléments, comportant à la fois une structure statique et dynamique, est modélisé en UML par une collaboration.

Graphiquement, une collaboration se représente par une ellipse en trait pointillé comprenant deux compartiments. Le compartiment supérieur contient le nom de la collaboration et le compartiment inférieur montre les participants à la collaboration (cf. figure 7.2).

### 7.1.4 Interactions et lignes de vie

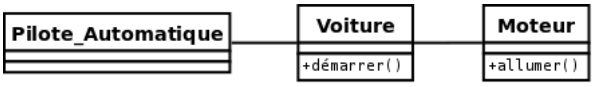



Figure 7.3: Diagramme de classe d'un système de pilotage.

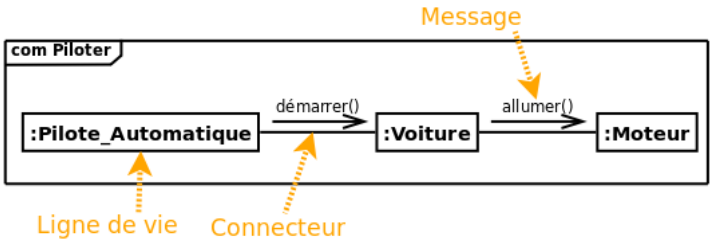



Figure 7.4: Diagramme de communication d'un système de pilotage.

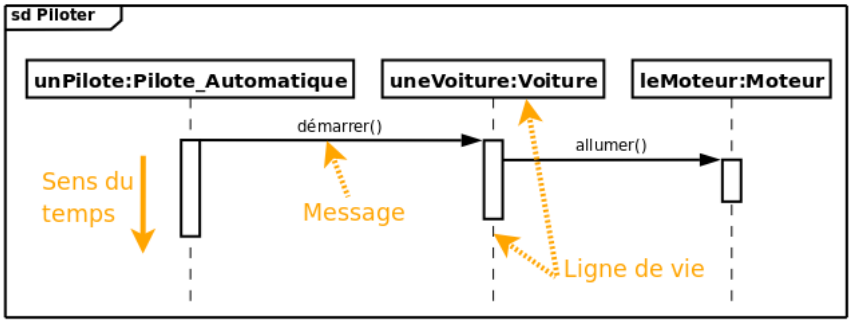



Figure 7.5: Diagramme de séquence d'un système de pilotage.

Une interaction montre le comportement d'un classeur structuré ou d'une collaboration en se focalisant sur l'échange d'informations entre les éléments du classeur ou de la collaboration. Une interaction contient un jeu de ligne de vie. Chaque ligne de vie correspond à une partie interne d'un classeur ou d'une collaboration (*i.e.* un rôle dans le cas d'une collaboration). L'interaction décrit donc l'activité interne des éléments du classeur ou de la collaboration, appelés lignes de vie, par les messages qu'ils échangent.

UML propose principalement deux diagrammes pour illustrer une interaction : le diagramme de communication et celui de séquence. Une même interaction peut être présentée aussi bien par l'un que par l'autre (cf. figure 7.4 et 7.5).

**Remarque**

A ces deux diagrammes, UML 2.0 en ajoute un troisième : le diagramme de timing. Son usage est limité à la modélisation des systèmes qui s'exécutent sous de fortes contraintes de temps, comme les systèmes temps réel.

### 7.1.5 Représentation générale

Un diagramme d'interaction se représente par un rectangle contenant, dans le coin supérieur gauche, un pentagone accompagné du mot-clef *sd* lorsqu'il s'agit d'un diagramme de séquence (cf. figure 7.5) et *com* lorsqu'il s'agit d'un diagramme de communication (cf. figure 7.4). Le mot clé est suivi du nom de l'interaction. Dans le pentagone, on peut

aussi faire suivre le nom par la liste des lignes de vie impliquées, précédée par le mot clé *lifelines :*. Enfin, des attributs peuvent être indiqués dans la partie supérieure du rectangle contenant le diagramme (cf. figure 7.11). La syntaxe de ces attributs est la même que celle des attributs d'un classe.

---

1
Doctrine ou point de vue qui consiste à considérer les phénomènes comme des totalités.

## 7.2 Diagramme de communication *(Communication diagram)*

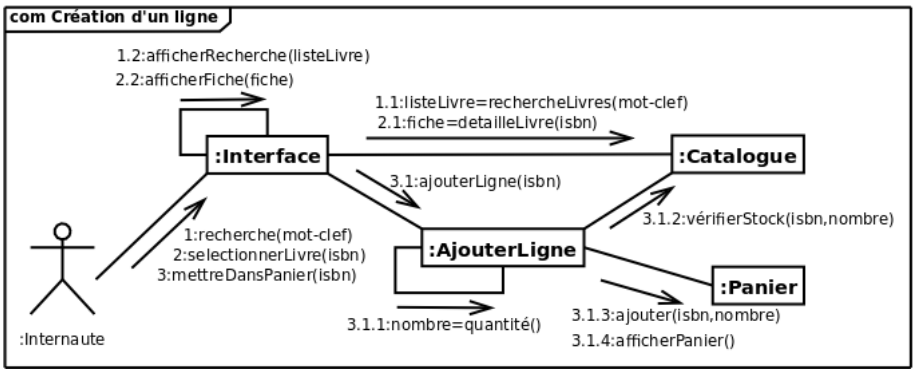



Figure 7.6: Diagramme de communication illustrant la recherche puis l'ajout, dans son panier virtuel, d'un livre lors d'une commande sur Internet.

---

Contrairement à un diagramme de séquence, un diagramme de communication[2] rend compte de l'organisation spatiale des participants à l'interaction, il est souvent utilisé pour illustrer un cas d'utilisation ou pour décrire une opération. Le diagramme de communication aide à valider les associations du diagramme de classe en les utilisant comme support de transmission des messages.

### 7.2.1 Représentation des lignes de vie

Les lignes de vie sont représentées par des rectangles contenant une étiquette dont la syntaxe est :

```
[<nom_du_rôle>] : [<Nom_du_type>]
```

Au moins un des deux noms doit être spécifié dans l'étiquette, les deux points (:) sont, quand à eux, obligatoire.

### 7.2.2 Représentation des connecteurs

Les relations entre les lignes de vie sont appelées connecteurs et se représentent par un trait plein reliant deux lignes de vies et dont les extrémités peuvent être ornées de multiplicités.

### 7.2.3 Représentation des messages

Dans un diagramme de communication, les messages sont généralement ordonnés selon un numéro de séquence croissant.

Un message est, habituellement, spécifié sous la forme suivante:

```
[ '['<cond>']' [<séq>] [ *[||] ['['<iter>']'] ] :] [<var> :=] <msg>([<par>])
```

**<cond>**
est une condition sous forme d'expression booléenne entre crochets.

**<séq>**
est le numéro de séquence du message. On numérote les messages par envoi et sous-envoi désignés par des chiffres séparés par des points : ainsi l'envoi du message 1.4.4 est postérieur à celui du message 1.4.3, tous deux étant des conséquences (*i.e.* des sous-envois) de la réception d'un message 1.4. La simultanéité d'un envoi est désignée par une lettre : les messages 1.6*a* et 1.6*b* sont envoyés en même temps.

**<iter>**
spécifie (en langage naturel, entre crochets) l'envoi séquentiel (ou en parallèle, avec `||`) de plusieurs message. On peut omettre cette spécification et ne garder que le caractère `*` (ou `*||`) pour désigner un message récurrent envoyé un certain nombre de fois.

**<var>**
est la valeur de retour du message, qui sera par exemple transmise en paramètre à un autre message.

**<msg>**
est le nom du message.

**<par>**
désigne les paramètres (optionnels) du message.

Cette syntaxe un peu complexe permet de préciser parfaitement l'ordonnancement et la synchronisation des messages entre les objets du diagramme de communication (cf. figure 7.6). La direction d'un message est spécifiée par une flèche pointant vers l'un ou l'autre des objets de l'interaction, reliés par ailleurs avec un trait continu (connecteur).

---

2
Dans la norme UML 1, le diagramme de communication s'appelait diagramme de collaboration.

## 7.3 Diagramme de séquence *(Sequence diagram)*

Les principales informations contenues dans un diagramme de séquence sont les messages échangés entre les lignes de vie, présentés dans un ordre chronologique. Ainsi, contrairement au diagramme de communication, le temps y est représenté explicitement par une dimension (la dimension verticale) et s'écoule de haut en bas (cf. figure 7.5).

### 7.3.1 Représentation des lignes de vie

Une ligne de vie se représente par un rectangle, auquel est accroché une ligne verticale pointillée, contenant une étiquette dont la syntaxe est :

```
[<nom_du_rôle>] : [<Nom_du_type>]
```

Au moins un des deux noms doit être spécifié dans l'étiquette, les deux points (:) sont, quand à eux, obligatoire.

### 7.3.2 Représentation des messages

Un message définit une communication particulière entre des lignes de vie. Plusieurs types de messages existent, les plus commun sont :

- l'envoi d'un signal ;
- l'invocation d'une opération ;
- la création ou la destruction d'une instance.

**Messages asynchrones**


Figure 7.7: Représentation d'un message asynchrone.

Une interruption ou un évènement sont de bons exemples de signaux. Ils n'attendent pas de réponse et ne bloquent pas l'émetteur qui ne sait pas si le message arrivera à destination, le cas échéant quand il arrivera et s'il serra traité par le destinataire. Un signal est, par définition, un message asynchrone.

Graphiquement, un message asynchrone se représente par une flèche en traits pleins et à l'extrémité ouverte partant de la ligne de vie d'un objet expéditeur et allant vers celle de l'objet cible (figure 7.7).

**Messages synchrones**

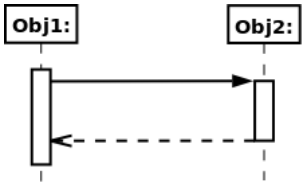



Figure 7.8: Représentation d'un message synchrone.

L'invocation d'une opération est le type de message le plus utilisé en programmation objet. L'invocation peut être asynchrone ou synchrone. Dans la pratique, la pluspart des invocations sont synchrones, l'émetteur reste alors bloqué le temps que dure l'invocation de l'opération.

Graphiquement, un message synchrone se représente par une flèche en traits pleins et à l'extrémité pleine partant de la ligne de vie d'un objet expéditeur et allant vers celle de l'objet cible (figure 7.8). Ce message peut être suivi d'une réponse qui se représente par une flèche en pointillé (figure 7.8).

**Messages de création et destruction d'instance**

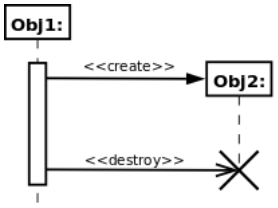



Figure 7.9: Représentation d'un message de création et destruction d'instance.

La création d'un objet est matérialisée par une flèche qui pointe sur le sommet d'une ligne de vie (figure 7.9).

La destruction d'un objet est matérialisée par une croix qui marque la fin de la ligne de vie de l'objet (figure 7.9). La destruction d'un objet n'est pas nécessairement consécutive à la réception d'un message.

**Événements et messages**

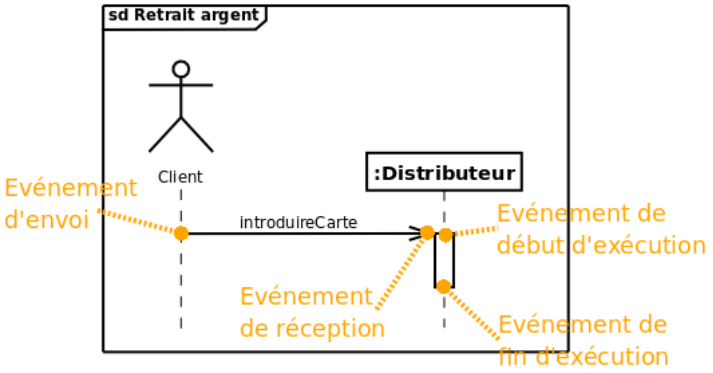



Figure 7.10: Les différents évènements correspondant à un message asynchrone.

UML permet de séparer clairement l'envoi du message, sa réception, ainsi que le début de l'exécution de la réaction et sa fin (figure 7.10).

**Syntaxe des messages et des réponses**

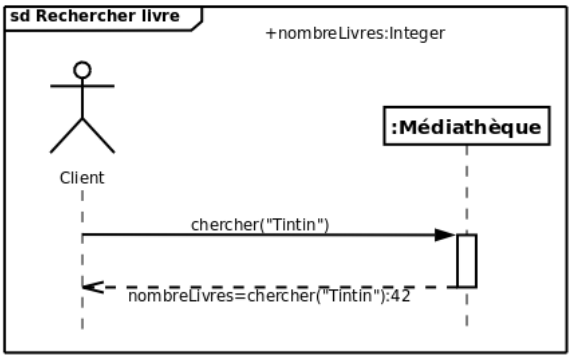



Figure 7.11: Syntaxe des messages et des réponses.

Dans la plupart des cas, la réception d'un message est suivie de l'exécution d'une méthode d'une classe. Cette méthode peut recevoir des arguments et la syntaxe des messages permet de transmettre ces arguments. La syntaxe de ces messages est la même que pour un diagramme de communication (cf. section 7.2.3) excepté deux points :

- la direction du message est directement spécifiée par la direction de la flèche qui matérialise le message, et non par une flèche supplémentaire au dessus du connecteur reliant les objets comme c'est le cas dans un diagramme de communication ;
- les numéros de séquence sont généralement omis puisque l'ordre relatif des messages est déjà matérialisé par l'axe vertical qui représente l'écoulement du temps.

La syntaxe de réponse à un message est la suivante :

```
[<attribut> = ] message [ : <valeur_de_retour>]
```

où `message` représente le message d'envoi.

La figure 7.11 montre un exemple d'exécution d'une méthode avec une réponse.

**Message perdu et trouvé**

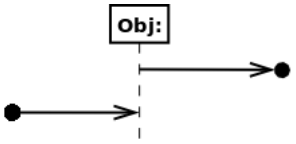



Figure 7.12: Représentation d'un message perdu et d'un message trouvé.

Un message complet est tel que les événements d'envoi et de réception sont connus. Comme nous l'avons déjà vu, un message complet se représente par une simple flèche dirigée de l'émetteur vers le récepteur.

Un message perdu est tel que l'événement d'envoi est connu, mais pas l'événement de réception. Il se représente par une flèche qui pointe sur une petite boule noire (figure 7.12).

Un message trouvé est tel que l'événement de réception est connu, mais pas l'événement d'émission. Une flèche partant d'une petite boule noire représente un message trouvé (figure 7.12).

**Porte**

Une porte est un point de connexion qui permet de représenter un même message dans plusieurs fragments d'interaction. Ces messages entrants et sortants vont d'un bord d'une diagramme à une ligne de vie (ou l'inverse).

**Exécution de méthode et objet actif**

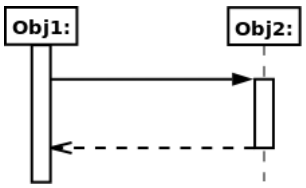



Figure 7.13: Représentation d'un objet actif (à gauche) et d'une exécution sur un objet passif (à droite).

Un objet actif initie et contrôle le flux d'activités. Graphiquement, la ligne pointillée verticale d'un objet actif est remplacée par un double trait vertical (cf. figures 7.13).

Un objet passif, au contraire, a besoin qu'on lui donne le flux d'activité pour pouvoir exécuter une méthode. La spécification de l'exécution d'une réaction sur un objet passif se représente par un rectangle blanc ou gris placé sur la ligne de vie en pointillée (cf. figures 7.13). Le rectangle peut éventuellement porter un label.

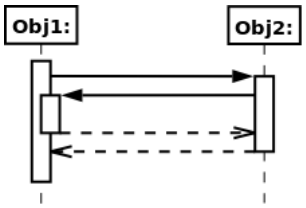



Figure 7.14: Représentation d'une exécution simultanée (à gauche).

Les exécutions simultanées sur une même ligne de vie sont représentées par un rectangle chevauchant comme le montre la figure 7.14.

### 7.3.3 Fragments d'interaction combinés

**Introduction**

Un fragment combiné représente des articulations d'interactions. Il est défini par un opérateur et des opérandes. L'opérateur conditionne la signification du fragment combiné. Il existe 12 d'opérateurs définis dans la notation UML 2.0. Les fragments combinés permettent de décrire des diagrammes de séquence de manière compacte. Les fragments combinés peuvent faire intervenir l'ensemble des entités participant au scénario ou juste un sous-ensemble.

Un fragment combiné se représente de la même façon qu'une interaction. Il est représenté dans un rectangle dont le coin supérieur gauche contient un pentagone. Dans le pentagone figure le type de la combinaison, appelé *opérateur d'interaction*. Les opérandes d'un opérateur d'interaction sont séparés par une ligne pointillée. Les conditions de choix des opérandes sont données par des expressions booléennes entre crochets ([ ]).

La liste suivante regroupe les opérateurs d'interaction par fonctions :

- les opérateurs de choix et de boucle : **alternative**, **option**, **break** et **loop** ;
- les opérateurs contrôlant l'envoi en parallèle de messages : **parallel** et **critical region** ;
- les opérateurs contrôlant l'envoi de messages : **ignore**, **consider**, **assertion** et **negative** ;
- les opérateurs fixant l'ordre d'envoi des messages : **weak sequencing** , **strict sequencing**.

Nous n'aborderons que quelques-unes de ces interactions dans la suite de cette section.

**Opérateur *alt***

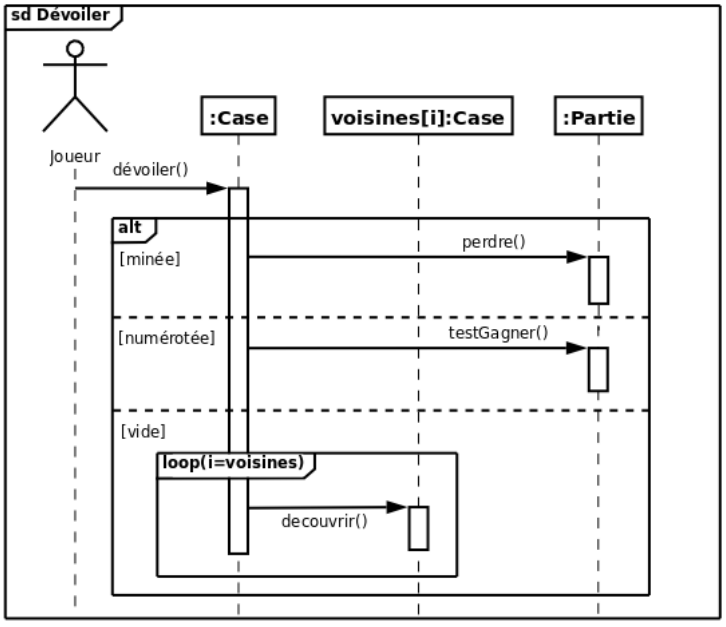



Figure 7.15: Représentation d'un choix dans un diagramme de séquence illustrant le découvrement d'une case au jeu du démineur.

L'opérateur *alternative*, ou *alt*, est un opérateur conditionnel possédant plusieurs opérandes (cf. figure 7.15). C'est un peu l'équivalent d'une exécution à choix multiple

(condition *switch* en C++). Chaque opérande détient une condition de garde. L'absence de condition de garde implique une condition vraie (*true*). La condition *else* est vraie si aucune autre condition n'est vraie. Exactement un opérande dont la condition est vraie est exécuté. Si plusieur opérandes prennent la valeur vraie, le choix est non déterministe.

**Opérateurs *opt***

L'opérateur *option*, ou *opt*, comporte une opérande et une condition de garde associée. Le sous-fragment s'exécute si la condition de garde est vraie et ne s'exécute pas dans le cas contraire.

**Opérateur *loop***

Un fragment combiné de type *loop* possède un sous-fragment et spécifie un compte minimum et maximum (boucle) ainsi qu'une condition de garde.

La syntaxe de la boucle est la suivante :

```
loop[ '('<minInt> [ ',' <maxInt> ] ')' ]
```

La condition de garde est placée entre crochets sur la ligne de vie. La boucle est répétée au moins `minInt` fois avant qu'une éventuelle condition de garde booléenne ne soit testée. Tant que la condition est vraie, la boucle continue, au plus `maxInt` fois. Cette syntaxe peut être remplacée par une indication intelligible comme sur la figure 7.15.

**Opérateur *par***

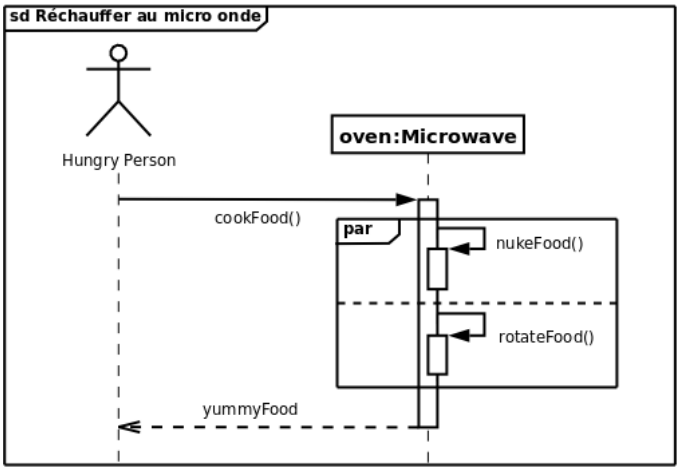



Figure 7.16: *Microwave* est un exemple d'objet effectuant deux tâches en parallèle.

Un fragments combiné de type *parallel*, ou *par*, possède au moins deux sous-fragments exécutés simultanément (cf. figure 7.16). La concurrence est logique et n'est pas nécessairement physique : les exécutions concurrentes peuvent s'entrelacer sur un même chemin d'exécution dans la pratique.

**Opérateur *strict***

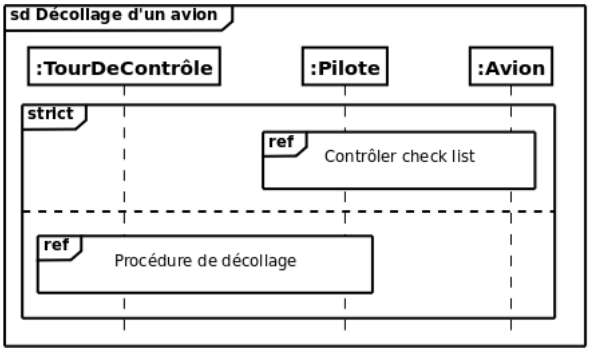



Figure 7.17: Procédures de décollage d'un avion dans l'ordre.

Un fragments combiné de type *strict sequencing*, ou *strict*, possède au moins deux sous-fragments. Ceux-ci s'exécutent selon leur ordre d'apparition au sein du fragment combiné. Ce fragment combiné est utile surtout lorsque deux parties d'un diagramme n'ont pas de ligne de vie en commun (cf. figure 7.17).

### 7.3.4 Utilisation d'interaction (*interaction use*)

Il est possible de faire référence à une interaction (on appelle cela une *utilisation d'interaction*) dans la définition d'une autre interaction. Comme pour toute référence modulaire, cela permet la réutilisation d'une définition dans de nombreux contextes différents.

Lorsqu'une utilisation d'interaction s'exécute, elle produit le même effet que l'exécution d'une interaction référencée avec la substitution des arguments fournie dans le cadre de l'utilisation de l'interaction. L'utilisation de l'interaction doit couvrir toutes les lignes de vie qui apparaissent dans l'interaction référencée. L'interaction référencée ne peut ajouter des lignes de vie que si elles ont lieu en son sein.

Graphiquement, une utilisation apparaît dans un diagramme de séquence sous forme de rectangle avec le tag *ref* (pour référence). On place dans le rectangle le nom de l'interaction référencée (cf. figure 7.17). La syntaxe complète pour spécifier l'interaction à réutiliser est la suivante :

```
[ <nomAttributValeurRetour> '=' ] <nomInteraction>
```

```
[ '(' [<arguments>] ')' ][ ':' <valeurRetour> ]
```

# Chapitre 8 Diagrammes de composants (*Component diagram*) & Diagrammes de déploiement (*Deployment diagram*)

## 8.1 Introduction

Les diagrammes de composants et les diagrammes de déploiement sont les deux derniers types de vues statiques en UML. Les premiers décrivent le système modélisé sous forme de composants réutilisables et mettent en évidence leurs relations de dépendance. Les seconds se rapprochent encore plus de la réalité physique, puisqu'ils identifient les éléments matériels (PC, Modem, Station de travail, Serveur, etc.), leur disposition physique (connexions) et la disposition des exécutables (représentés par des composants) sur ces éléments matériels.

## 8.2 Diagrammes de composants

### 8.2.1 Pourquoi des composants ?

Dans la section 1.1.4, parmi tous les facteurs qui concourent à la qualité d'un logiciel, nous avons introduit la notion de *réutilisabilité* comme étant l'aptitude d'un logiciel à être réutilisé, en tout ou en partie, dans de nouvelles applications. Or, la notion de classe, de part sa faible granularité et ses connexions figées (les associations avec les autres classes matérialisent des liens structurels), ne constitue pas une réponse adaptée à la problématique de la réutilisation.

Pour faire face à ce problème, les notions de patrons et de canevas d'applications ont percé dans les années 1990 pour ensuite laisser la place à un concept plus générique et fédérateur : celui de *composant*. La programmation par composants constitue une évolution technologique soutenue par de nombreuses plateformes (composants EJB, CORBA, .Net, WSDL, …). Ce type de programmaiton met l'accent sur la réutilisation du composant et l'indépendance de son évolution vis-à-vis des applications qui l'utilisent.

La programmation orientée composant s'intègre très bien dans le contexte de la programmation orientée objet puisqu'il ne s'agit, finalement, que d'un facteur d'échelle. En effet, l'utilisation de composants est assimilable à une approche objet, non pas au niveau du code, mais au niveau de l'architecture générale du logiciel.

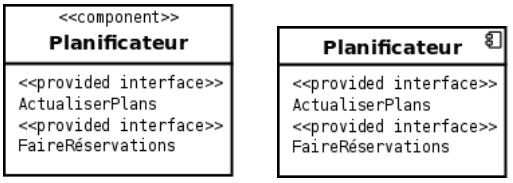



Figure 8.1: Représentation d'un composant et de ses interfaces requises ou offertes sous la forme d'un classeur structuré stéréotypé *«component»*. Au lieu ou en plus du mot clé, on peut faire figurer une icône de composant (petit rectangle équipé de deux rectangles plus petits dépassant sur son côté gauche) dans l'angle supérieur droit (comme sur la figure de droite).


Figure 8.2: Représentation d'un composant accompagnée de la représentation explicite de ses interfaces requise et offerte.

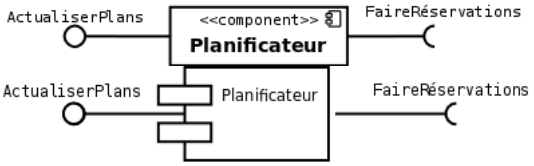



Figure 8.3: Représentation classique d'un composant et de ses interfaces requise (représenté par un demi-cercle) et offerte (représentée par un cercle). Cette représentation est souvent utilisée dans les diagrammes de composants (cf. figure 8.5). Sur la figure du bas, le stéréotype « *component* » est rendu inutile par la représentation même du composant.


Figure 8.4: Représentation d'un composant et de ses interfaces requise et offerte avec la représentation explicite de leur port correspondant.

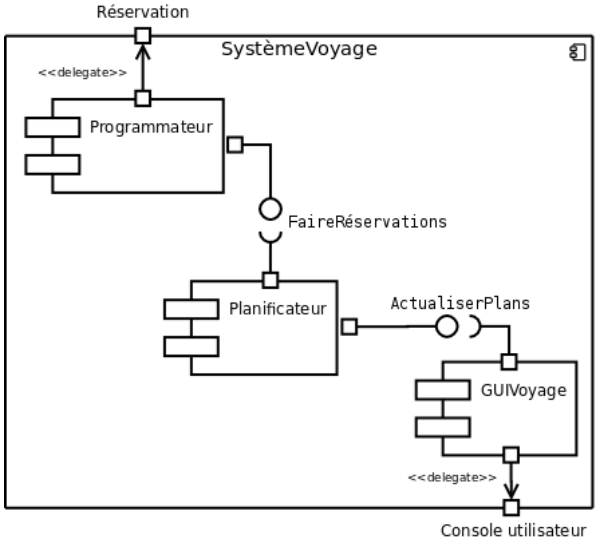



Figure 8.5: Représentation de l'implémentation d'un composant complexe contenant des sous-composants.

### 8.2.2 Notion de composant

Un composant doit fournir un service bien précis. Les fonctionnalités qu'il encapsule doivent être cohérentes entre elles et génériques (par opposition à spécialisées) puisque sa vocation est d'être réutilisable.

Un composant est une unité autonome représentée par un classeur structuré, stéréotypé *«component»*, comportant une ou plusieurs interfaces requises ou offertes. Son comportement interne, généralement réalisé par un ensemble de classes, est totalement masqué : seules ses interfaces sont visibles. La seule contrainte pour pouvoir substituer un composant par un autre est de respecter les interfaces requises et offertes.

Les figures 8.1, 8.2, 8.3 et 8.4 illustrent différentes façons de représenter un composant.

Un composant étant un classeur structuré, on peut en décrire la structure interne. L'implémentation d'un composant peut être réalisée par d'autres composants, des classes ou des artefacts (cf. section 8.3.3). Les éléments d'un composant peuvent être représentés dans le symbole du composant (cf. figure 8.5), ou à côté en les reliant au composant par une relation de dépendance.

Pour montrer les instances des composants, un diagramme de déploiement doit être utilisé (cf. section 8.3).

### 8.2.3 Notion de port

Un port est un point de connexion entre un classeur et son environnement.

Graphiquement, un port est représenté par un petit carré à cheval sur la bordure du contour du classeur. On peut faire figurer le nom du port à proximité de sa représentation.

Généralement, un port est associé à une interface requise ou offerte (cf. figure 8.4). Parfois, il est relié directement à un autre port situé sur la limite du composant englobant (cf. figure 8.5) par une flèche en trait plein, pouvant être stéréotypée *«delegate»*, et appelée *connecteur de délégation*.

L'utilisation des ports permet de modifier la structure interne d'un classeur sans affecter les clients externes.

### 8.2.4 Diagramme de composants

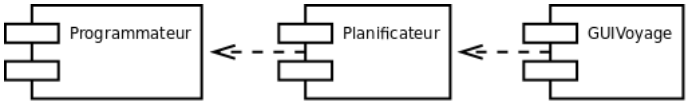



Figure 8.6: Exemple de diagramme montrant les dépendances entre composants.

La relation de dépendance est utilisée dans les diagrammes de composants pour indiquer qu'un élément de l'implémentation d'un composant fait appel aux services offerts par les éléments d'implémentation d'un autre composant (cf. figure 8.6).

Lorsqu'un composant utilise l'interface d'un autre composant, on peut utiliser la représentation de la figure 8.3 en imbriquant le demi-cercle d'une interface requise dans le cercle de l'interface offerte correspondante (cf. figure 8.5).

## 8.3 Diagramme de déploiement

### 8.3.1 Objectif du diagramme de déploiement

Un diagramme de déploiement décrit la disposition physique des ressources matérielles qui composent le système et montre la répartition des composants sur ces matériels. Chaque ressource étant matérialisée par un nœud, le diagramme de déploiement précise comment les composants sont répartis sur les nœuds et quelles sont les connexions entre les composants ou les nœuds. Les diagrammes de déploiement existent sous deux formes : spécification et instance.

### 8.3.2 Représentation des nœuds

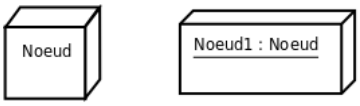



Figure 8.7: Représentation d'un nœud (à gauche) et d'une instance de nœud (à droite).

Chaque ressource est matérialisée par un nœud représenté par un cube comportant un nom (cf. figure 8.7). Un nœud est un classeur et peut posséder des attributs (quantité de mémoire, vitesse du processeur, …).

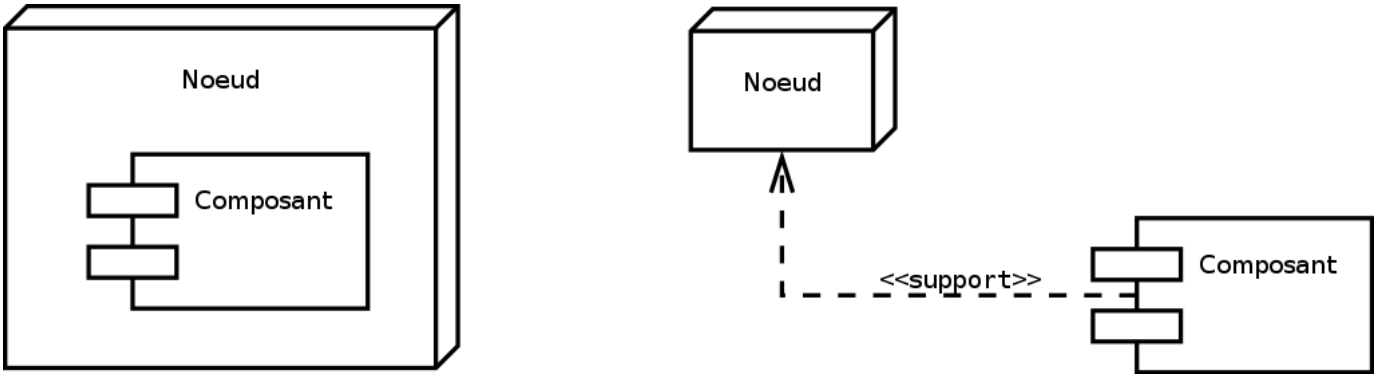



Figure 8.8: Deux possibilités pour représenter l'affectation d'un composant à un nœud.

Pour montrer qu'un composant est affecté à un nœud, il faut soit placer le composant dans le nœud, soit les relier par une relation de dépendance stéréotypée *«support»* orientée du composant vers le nœud (cf. figure 8.8).

### 8.3.3 Notion d'artefact (*artifact*)

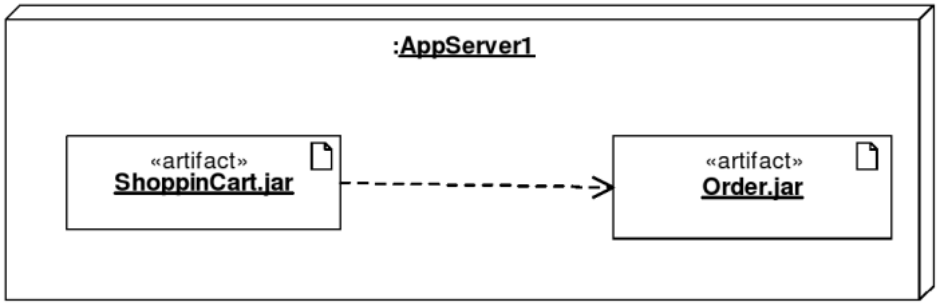



Figure 8.9: Représentation du déploiement de deux artefacts dans un nœud. La dépendance entre les deux artefacts est également représentée.

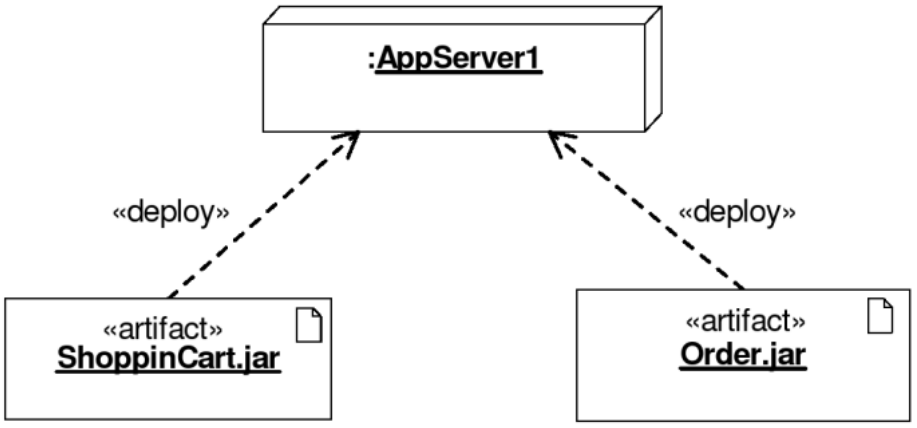



Figure 8.10: Représentation du déploiement de deux artefacts dans un nœud utilisant la relation de dépendance stéréotypée *«deploy»*.

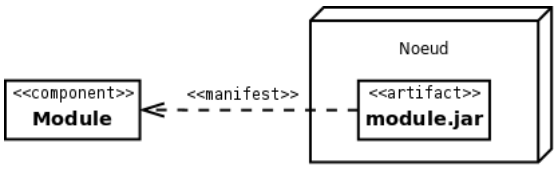



Figure 8.11: Représentation du déploiement dans un nœud d'un artefact manifestant un composant.

Un artefact correspond à un élément concret existant dans le monde réel (document, exécutable, fichier, tables de bases de données, script, …). Il se représente comme un classeur par un rectangle contenant le mot-clé *«artifact»* suivi du nom de l'artefact (cf. figures 8.9 et 8.9).

L'implémentation des modèles (classes, …) se fait sous la forme de jeu d'artefacts. On dit qu'un artefact peut manifester, c'est-à-dire résulter et implémenter, un ensemble d'éléments de modèle. On appelle *manifestation* la relation entre un élément de modèle et l'artefact qui l'implémente. Graphiquement, une manifestation se représente par une relation de dépendance stéréotypée *«manifest»* (cf. figure 8.11).

Une instance d'un artefact se déploie sur une instance de nœud. Graphiquement, on utilise une relation de dépendance (flèche en trait pointillé) stéréotypée *«deploy»* pointant vers le nœud en question (cf. figure 8.10). L'artefact peut aussi être inclus directement dans le cube représentant le noeud (cf. figure 8.9). En toute rigueur, seul des artefacts doivent être déployés sur des nœuds. Un composant doit donc être manifesté par un artefact qui, lui-même, peut être déployé sur un nœud.

### 8.3.4 Diagramme de déploiement

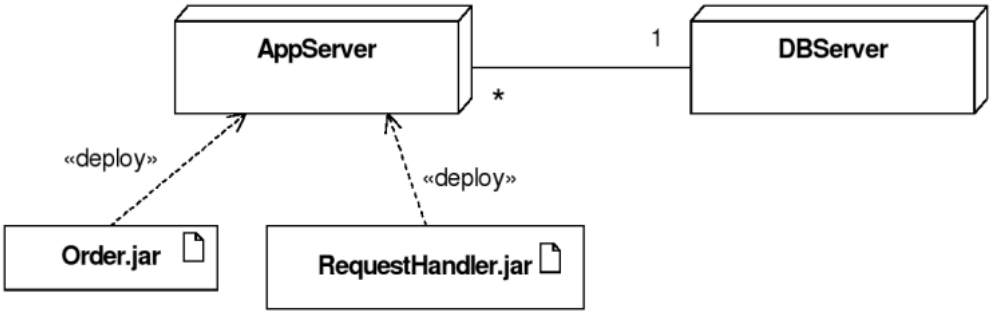



Figure 8.12: Exemple de diagramme de déploiement illustrant la communication entre plusieurs nœuds.

Dans un diagramme de déploiement, les associations entre nœuds sont des chemins de communication qui permettent l'échange d'informations (cf. figure 8.12).

# Chapitre 9 Mise en œuvre d'UML

## 9.1 Introduction

### 9.1.1 UML n'est pas une méthode

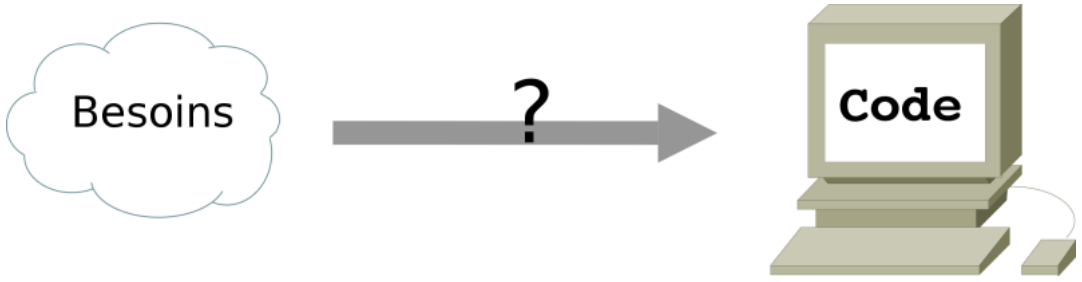



Figure 9.1: Quelle méthode pour passer de l'expression des besoins au code de l'application ?

La problématique que pose la mise en œuvre d'UML est simple : comment passer de l'expression des besoins au code de l'application ? Cette problématique est parfaitement illustrée par la figure 9.1.

Comme nous l'avons déjà dit, à maintes reprises, UML n'est qu'un langage de modélisation, ce n'est pas une méthode. En effet, UML ne propose pas une démarche de modélisation explicitant et encadrant toutes les étapes d'un projet, de la compréhension des besoins à la production du code de l'application. Une méthode se doit de définir une séquence d'étapes, partiellement ordonnées, dont l'objectif est de produire un logiciel de qualité qui répond aux besoins des utilisateurs dans des temps et des coûts prévisibles.

Bien qu'UML ne soit pas une méthode, ses auteurs précisent néanmoins qu'une méthode basée sur l'utilisation UML doit être :

**Pilotée par les cas d'utilisation :**
La principale qualité d'un logiciel étant son utilité, c'est-à-dire son adéquation avec les besoins des utilisateurs, toutes les étapes, de la spécification des besoins à la maintenance, doivent être guidées par les cas d'utilisation qui modélisent justement les besoins des utilisateurs.

**Centrée sur l'architecture :**

L'architecture est conçue pour satisfaire les besoins exprimés dans les cas d'utilisation, mais aussi pour prendre en compte les évolutions futures et les contraintes de réalisation. La mise en place d'une architecture adaptée conditionne le succès d'un développement. Il est important de la stabiliser le plus tôt possible.

**Itérative et incrémentale :**

L'ensemble du problème est décomposé en petites itérations, définies à partir des cas d'utilisation et de l'étude des risques. Les risques majeurs et les cas d'utilisation les plus importants sont traités en priorité. Le développement procède par des itérations qui conduisent à des livraisons incrémentales du système. Nous avons déjà présenté le modèle de cycle de vie par incrément dans la section 1.2.3.

### 9.1.2 Une méthode simple et générique

Dans les sections qui suivent (sections 9.2, 9.3 et 9.4) nous allons présenter une méthode simple et générique qui se situe à mi-chemin entre UP (*Unified Process*), qui constitue un cadre général très complet de processus de développement, et XP (*eXtreme Programming*) qui est une approche minimaliste à la mode centrée sur le code. Cette méthode est issue de celle présentée par [23] dans son livre « *UML - Modéliser un site e-commerce* » qui résulte de plusieurs années d'expérience sur de nombreux projets dans des domaines variés. Elle a donc montré son efficacité dans la pratique et est :

- conduite par les cas d'utilisation, comme UP, mais bien plus simple ;
- relativement légère et restreinte, comme XP, mais sans négliger les activités de modélisation en analyse et conception ;
- fondée sur l'utilisation d'un sous-ensemble nécessaire et suffisant du langage UML (modéliser 80% des problèmes en utilisant 20% d'UML).

Dans tous les cas, il faut garder à l'esprit qu'une méthode n'est pas une formule magique. Le fait de produire des diagrammes UML selon un ordre établi n'est en aucun cas une garantie de réussite. Une méthode ne sert qu'à canaliser et ordonner les étapes de la modélisation. La valeur n'est pas dans la méthode mais dans les personnes qui la mettent en œuvre.

## 9.2 Identification des besoins et spécification des fonctionnalités

### 9.2.1 Identification et représentation des besoins : diagramme de cas d'utilisation

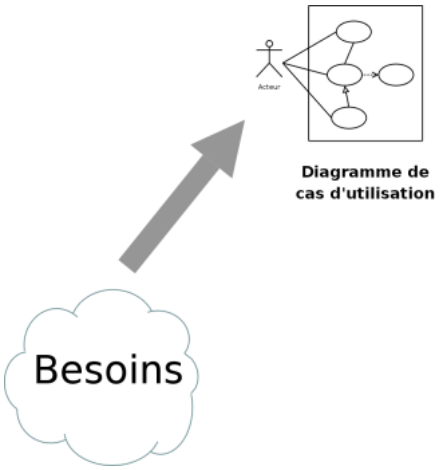



Figure 9.2: Les besoins sont modélisés par un diagramme de cas d'utilisation.

Les cas d'utilisation sont utilisés tout au long du projet. Dans un premier temps, on les crée pour identifier et modéliser les besoins des utilisateurs (figure 9.2). Ces besoins sont déterminés à partir des informations recueillies lors des rencontres entre informaticiens et utilisateurs. Il faut impérativement proscrire toute considération de réalisation lors de cette étape.

Durant cette étape, vous devrez déterminer les limites du système, identifier les acteurs et recenser les cas d'utilisation (cf. section 2.5). Si l'application est complexe, vous pourrez organiser les cas d'utilisation en paquetages.

Dans le cadre d'une approche itérative et incrémentale, il faut affecter un degré d'importance et un coefficient de risque à chacun des cas d'utilisation pour définir l'ordre des incréments à réaliser.

Les interactions entre les acteurs et le système (au sein des cas d'utilisation) seront explicitées sous forme textuelle et sous forme graphique au moyen de diagrammes de séquence (cf. section 9.2.2). Les utilisateurs ont souvent beaucoup de difficultés à exprimer clairement et précisément ce qu'ils attendent du système. L'objectif de cette étape et des deux suivantes (section 9.2.2 et 9.2.3) est justement de les aider à formuler et formaliser ces besoins.

### 9.2.2 Spécification détaillée des besoins : diagrammes de séquence système

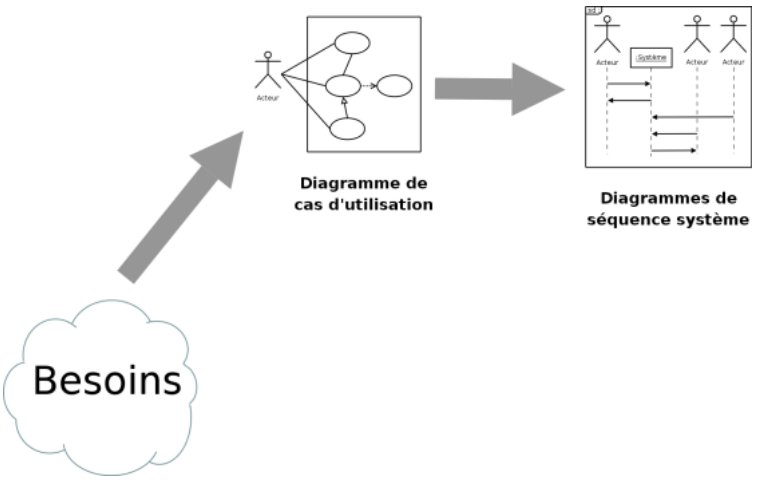



Figure 9.3: Les diagrammes de séquence système illustrent la description textuelle des cas d'utilisation.

Dans cette étape, on cherche à détailler la description des besoins par la description textuelle des cas d'utilisation (cf. section 2.5.3) et la production de diagrammes de séquence système illustrant cette description textuelle (figure 9.3). Cette étape amène souvent à mettre à jour le diagramme de cas d'utilisation puisque nous somme toujours dans la spécification des besoins.

Les scénarii de la description textuelle des cas d'utilisation peuvent être vus comme des instances de cas d'utilisation et sont illustrés par des diagrammes de séquence système. Il faut, au minimum, représenter le scénario nominal de chacun des cas d'utilisation par un diagramme de séquence qui rend compte de l'interaction entre l'acteur, ou les acteurs, et le système. Le système est ici considéré comme un tout et est représenté par une ligne de vie. Chaque acteur est également associé à une ligne de vie.

Lorsque les scénarii alternatifs d'un cas d'utilisation sont nombreux et importants, l'utilisation d'un diagramme d'états-transitions ou d'activités peut s'avérer préférable à une multitude de diagrammes de séquence.

### 9.2.3 Maquette de l'IHM de l'application (non couvert par UML)

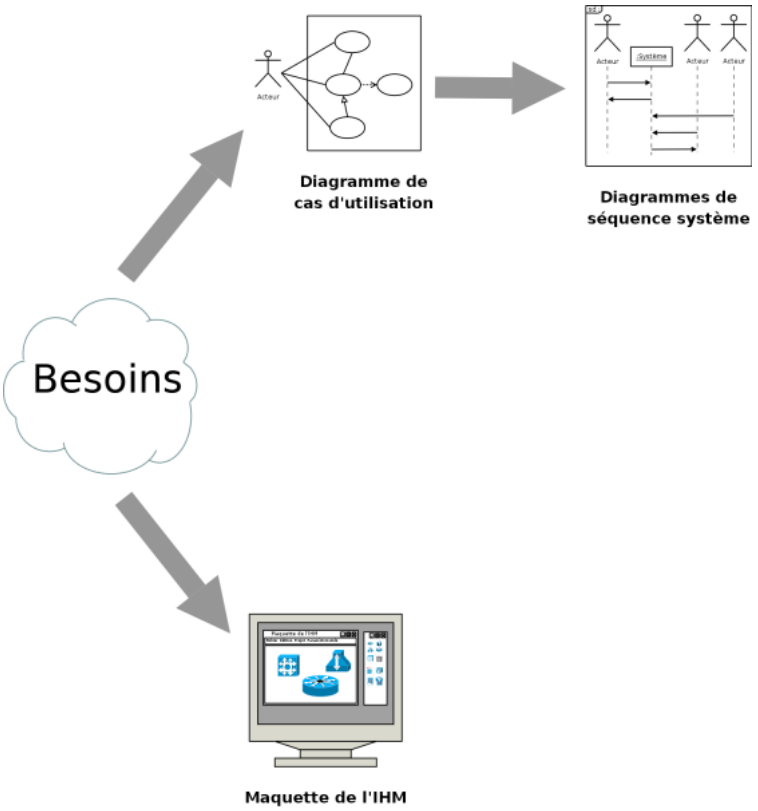



Figure 9.4: Une maquette d'IHM facilite les discussions avec les futurs utilisateurs.

Une maquette d'IHM (Interface Homme-Machine) est un produit jetable permettant aux utilisateurs d'avoir une vue concrète mais non définitive de la future interface de l'application (figure 9.4). La maquette peut très bien consister en un ensemble de dessins produits par un logiciel de présentation ou de dessin. Par la suite, la maquette pourra intégrer des fonctionnalités de navigation permettant à l'utilisateur de tester l'enchaînement des écrans ou des menus, même si les fonctionnalités restent fictives. La maquette doit être développée rapidement afin de provoquer des retours de la part des utilisateurs.

## 9.3 Phases d'analyse

### 9.3.1 Analyse du domaine : modèle du domaine

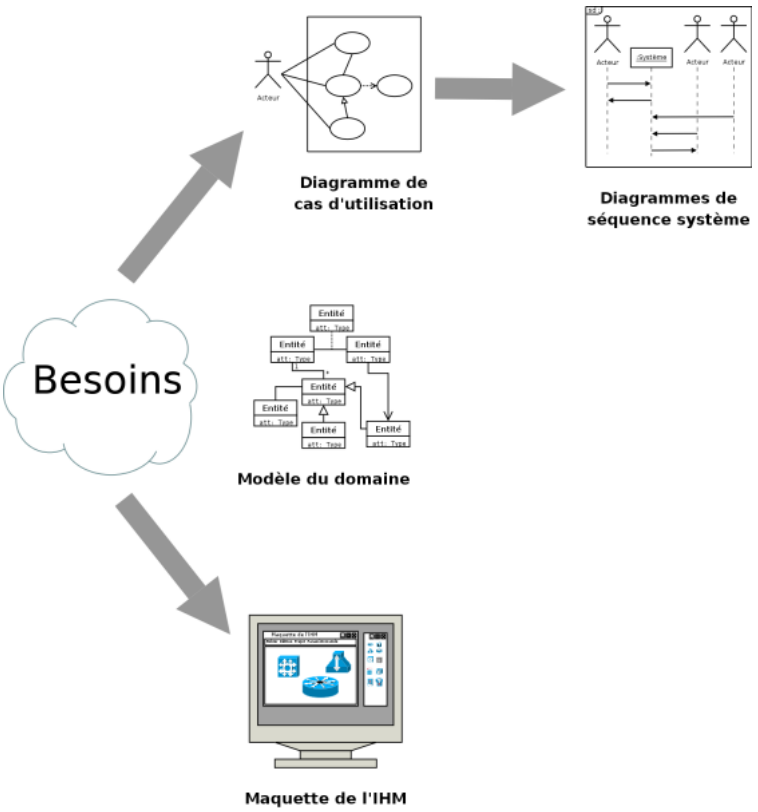



Figure 9.5: La phase d'analyse du domaine permet d'élaborer la première version du diagramme de classes.

---

La modélisation des besoins par des cas d'utilisation s'apparente à une analyse fonctionnelle classique. L'élaboration du modèle des classes du domaine permet d'opérer une transition vers une véritable modélisation objet. L'analyse du domaine est une étape totalement dissociée de l'analyse des besoins (sections 9.2.1, 9.2.2 et 9.2.3). Elle peut être menée avant, en parallèle ou après cette dernière.

La phase d'analyse du domaine permet d'élaborer la première version du diagramme de classes (figure 9.5) appelée modèle du domaine. Ce modèle doit définir les classes qui modélisent les entités ou concepts présents dans le domaine (on utilise aussi le terme de *métier*) de l'application. Il s'agit donc de produire un modèle des objets du monde réel dans un domaine donné. Ces entités ou concepts peuvent être identifiés directement à partir de la connaissance du domaine ou par des entretiens avec des experts du domaine. Il faut absolument utiliser le vocabulaire du métier pour nommer les classes et leurs attributs. Les classes du modèle du domaine ne doivent pas contenir d'opération, mais seulement des attributs. Les étapes à suivre pour établir ce diagramme sont (cf. section 3.6.1) :

- identifier les entités ou concepts du domaine ;
- identifier et ajouter les associations et les attributs ;
- organiser et simplifier le modèle en éliminant les classes redondantes et en utilisant l'héritage ;
- le cas échéant, structurer les classes en paquetage selon les principes de cohérence et d'indépendance.

L'erreur la plus courante lors de la création d'un modèle du domaine consiste à modéliser un concept par un attribut alors que ce dernier devait être modélisé par une classe. Si la seule chose que recouvre un concept est sa valeur, il s'agit simplement d'un attribut. Par contre, si un concept recouvre un ensemble d'informations, alors il s'agit plutôt d'une classe qui possède elle-même plusieurs attributs.

### 9.3.2 Diagramme de classes participantes

---

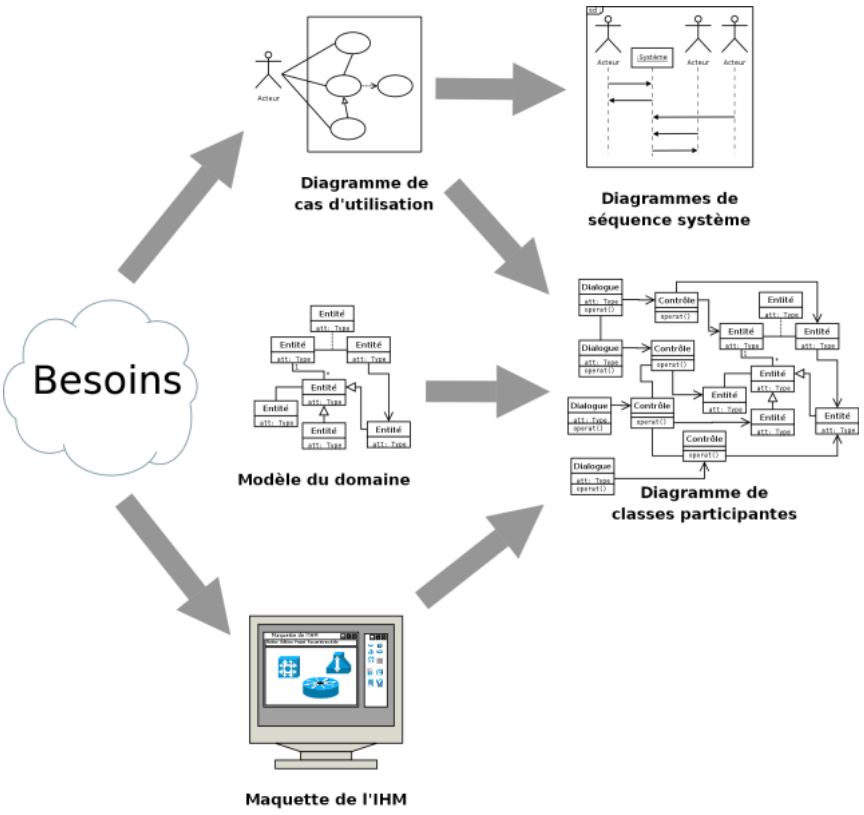



Figure 9.6: Le diagramme de classes participantes effectue la jonction entre les cas d'utilisation, le modèle du domaine et les diagrammes de conception logicielle.

---

Le diagramme de classes participantes est particulièrement important puisqu'il effectue la jonction entre, d'une part, les cas d'utilisation (section 9.2.1), le modèle du domaine (section 9.3.1) et la maquette (section 9.2.3), et d'autre part, les diagrammes de conception logicielle que sont les diagrammes d'interaction (section 9.4.1) et le diagramme de classes de conception (section 9.4.2). Les diagrammes de conception logicielle n'apparaissent pas encore sur la figure 9.6.

Il n'est pas souhaitable que les utilisateurs interagissent directement avec les instances des classes du domaine par le biais de l'interface graphique. En effet, le modèle du domaine doit être indépendant des utilisateurs et de l'interface graphique. De même, l'interface graphique du logiciel doit pouvoir évoluer sans répercussion sur le cœur de l'application. C'est le principe fondamental du découpage en couches d'une application. Ainsi, le diagramme de classes participantes modélise trois types de classes d'analyse, les *dialogues*, les *contrôles* et les *entités* ainsi que leurs relations.

**Les classes de dialogues –**
Les classes qui permettent les interactions entre l'IHM et les utilisateurs sont qualifiées de *dialogues*. Ces classes sont directement issues de l'analyse de la maquette présentée section 9.2.3. Il y a au moins un dialogue pour chaque association entre un acteur et un cas d'utilisation du diagramme de cas d'utilisation de la section 9.2.1. En général, les dialogues vivent seulement le temps du déroulemet du cas d'utilisation concerné.

**Les classes de contrôles –**
Les classes qui modélisent la cinématique de l'application sont appelées *contrôles*. Elles font la jonction entre les dialogues et les classes métier en permettant au différentes vues de l'application de manipuler des informations détenues par un ou plusieurs objets métier. Elles contiennent les règles applicatives et les isolent à la fois des dialogues et des entités.

**Les classes entités –**
Les classes métier, qui proviennent directement du modèle du domaine (cf. section 9.3.1), sont qualifiées d'*entités*. Ces classes sont généralement persistantes, c'est-à-dire qu'elles survivent à l'exécution d'un cas d'utilisation particulier et qu'elles permettent à des données et des relations d'être stockées dans des fichiers ou des bases de données. Lors de l'implémentation, ces classes peuvent ne pas se concrétiser par des classes mais par des relations, au sens des bases de données relationnelles (cf. section 3.6.3).

Lors de l'élaboration du diagramme de classes participantes, il faut veiller au respect des règles suivantes :

- Les entités, qui sont issues du modèle du domaine, ne comportent que des attributs (cf. section 9.3.1).
- Les entités ne peuvent être en association qu'avec d'autres entités ou avec des contrôles, mais, dans ce dernier cas, avec une contrainte de navigabilité interdisant de traverser une association d'une entité vers un contrôle.
- Les contrôles ne comportent que des opérations. Ils implémentent la logique applicative (*i.e.* les fonctionnalités de l'application), et peuvent correspondre à des règles transverses à plusieurs entités. Chaque contrôle est généralement associé à un cas d'utilisation, et *vice versa*. Mais rien n'empêche de décomposer un cas d'utilisation complexe en plusieurs contrôles.
- Les contrôles peuvent être associés à tous les types de classes, y compris d'autres contrôles. Dans le cas d'une association entre un dialogue et un contrôle, une contrainte de navigabilité doit interdire de traverser l'association du contrôle vers le dialogue.
- Les dialogues comportent des attributs et des opérations. Les attributs représentent des informations ou des paramètres saisis par l'utilisateur ou des résultats d'actions. Les opérations réalisent (généralement par délégation aux contrôles) les actions que l'utilisateur demande par le biais de l'IHM.
- Les dialogues peuvent être en association avec des contrôles ou d'autres dialoques, mais pas directement avec des entités.
- Il est également possible d'ajouter les acteurs sur le diagramme de classes participantes en respectant la règle suivante : un acteur ne peut être lié qu'à un dialogue.

Certaines classes possèdent un comportement dynamique complexe. Ces classes auront intérêt à être détaillées par des diagrammes d'états-transitions.

L'attribution des bonnes responsabilités, dégagée dans la section 9.2.2, aux bonnes classes est l'un des problèmes les plus délicats de la conception orientée objet. Ce problème sera affronté en phase de conception lors de l'élaboration des diagrammes d'interaction (section 9.4.1) et du diagramme de classes de conception (section 9.4.2).

Lors de la phase d'élaboration du diagramme de classes participantes, le chef de projet a la possibilité de découper le travail de son équipe d'analystes par cas d'utilisation. L'analyse et l'implémentation des fonctionnalités dégagées par les cas d'utilisation définissent alors les itérations à réaliser. L'ordonnancement des itérations étant défini par le degré d'importance et le coefficient de risque affecté à chacun des cas d'utilisation dans la section 9.2.1.

### 9.3.3 Diagrammes d'activités de navigation

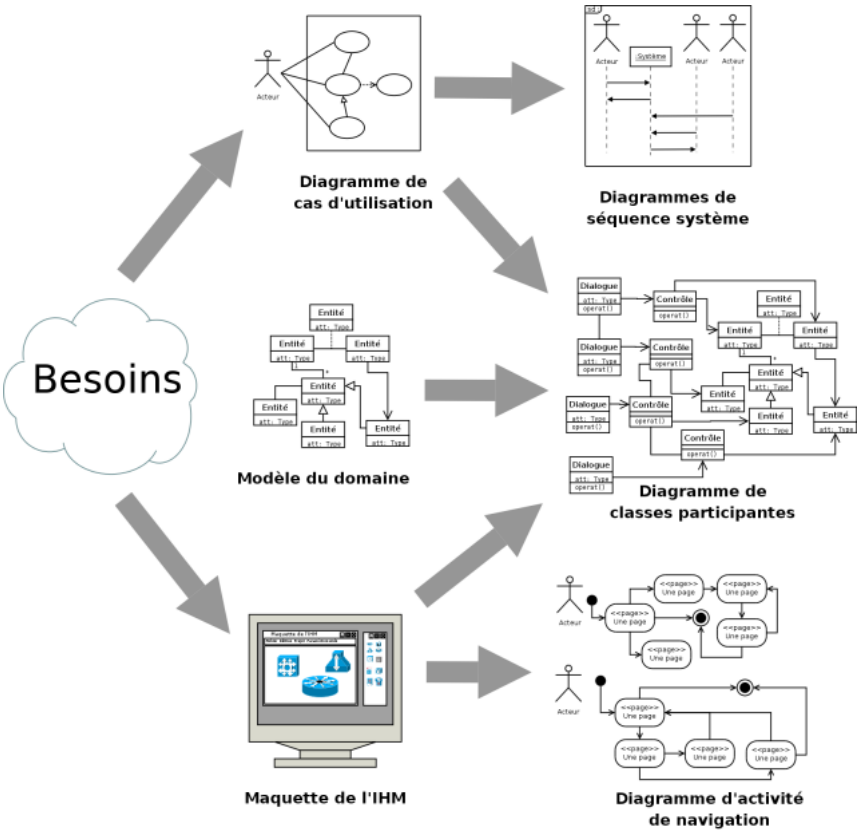



Figure 9.7: Les diagrammes d'activités de navigation représentent graphiquement l'activité de navigation dans l'IHM.

Les IHM modernes facilitent la communication entre l'application et l'utilisateur en offrant toute une gamme de moyens d'action et de visualisation comme des menus déroulants ou contextuels, des palettes d'outils, des boîtes de dialogues, des fenêtres de visualisation, etc. Cette combinaison possible d'options d'affichage, d'interaction et de navigation aboutis aujourd'hui à des interfaces de plus en plus riches et puissantes.

UML offre la possibilité de représenter graphiquement cette activité de navigation dans l'interface en produisant des diagrammes dynamiques. On appelle ces diagrammes des diagrammes de navigation. Le concepteur à le choix d'opter pour cette modélisation entre des diagrammes d'états-transitions et des diagrammes d'activités. Les diagrammes d'activités constituent peut-être un choix plus souple et plus judicieux.

Les diagrammes d'activités de navigation sont à relier aux classes de dialogue du diagramme de classes participantes. Les différentes activités du diagramme de navigation peuvent être stéréotypées en fonction de leur nature : « *fenêtre* », « *menu* », « *menu contextuel* », « *dialogue* », etc.

La modélisation de la navigation à intérêt à être structurée par acteur.

## 9.4 Phase de conception

### 9.4.1 Diagrammes d'interaction

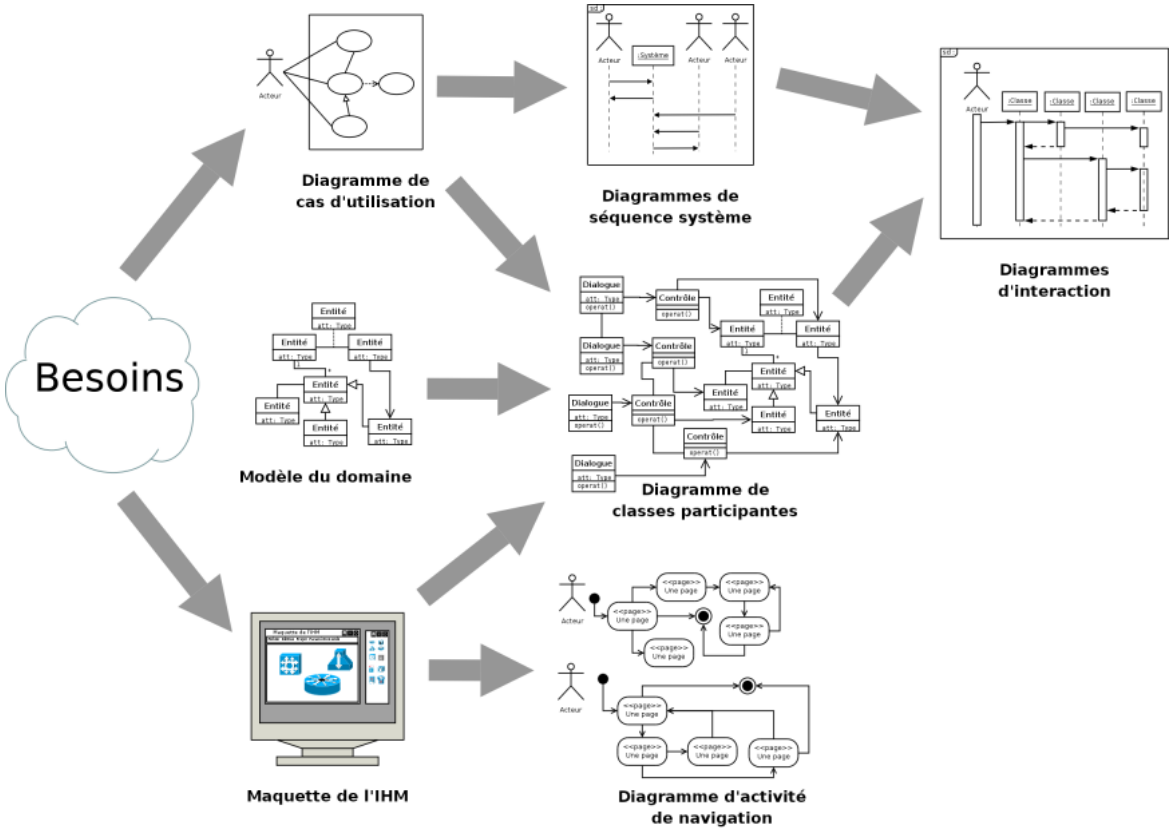



Figure 9.8: Les diagrammes d'interaction permettent d'attribuer précisément les responsabilités de comportement aux classes d'analyse.

Maintenant, il faut attribuer précisément les responsabilités de comportement, dégagée par le diagrammes de séquence système dans la section 9.2.2, aux classes d'analyse du diagramme de classes participantes élaboré dans la section 9.3.2. Les résultats de cette réflexion sont présentés sous la forme de diagrammes d'interaction UML (figure 9.8). Inversement, l'élaboration de ces diagrammes facilite grandement la réflexion.

Parallèlement, une première ébauche de la vue statique de conception, c'est-à-dire du diagramme de classes de conception, est construite et complétée. Durant cette phase, l'ébauche du diagramme de classes de conception reste indépendante des choix technologiques qui seront faits ultérieurement (dans la section 9.4.2).

Pour chaque service ou fonction, il faut décider quelle est la classe qui va le contenir. Les diagrammes d'interactions (*i.e* de séquence ou de communication) sont particulièrement utiles au concepteur pour représenter graphiquement ces décisions d'allocations des responsabilités. Chaque diagramme va représenter un ensemble d'objets de classes différentes collaborant dans le cadre d'un scénario d'exécution du système.

Dans les diagrammes d'interaction, les objets communiquent en s'envoyant des messages qui invoquent des opérations sur les objets récepteurs. Il est ainsi possible de suivre visuellement les interactions dynamiques entre objets, et les traitements réalisés par chacun d'eux. Avec un outil de modélisation UML (comme Rational Rose ou PowerAMC), la spécification de l'envoi d'un message entre deux objets crée effectivement une opération publique sur la classe de l'objet cible. Ce type d'outil permet réellement de mettre en œuvre l'allocation des responsabilités à partir des diagrammes d'interaction.

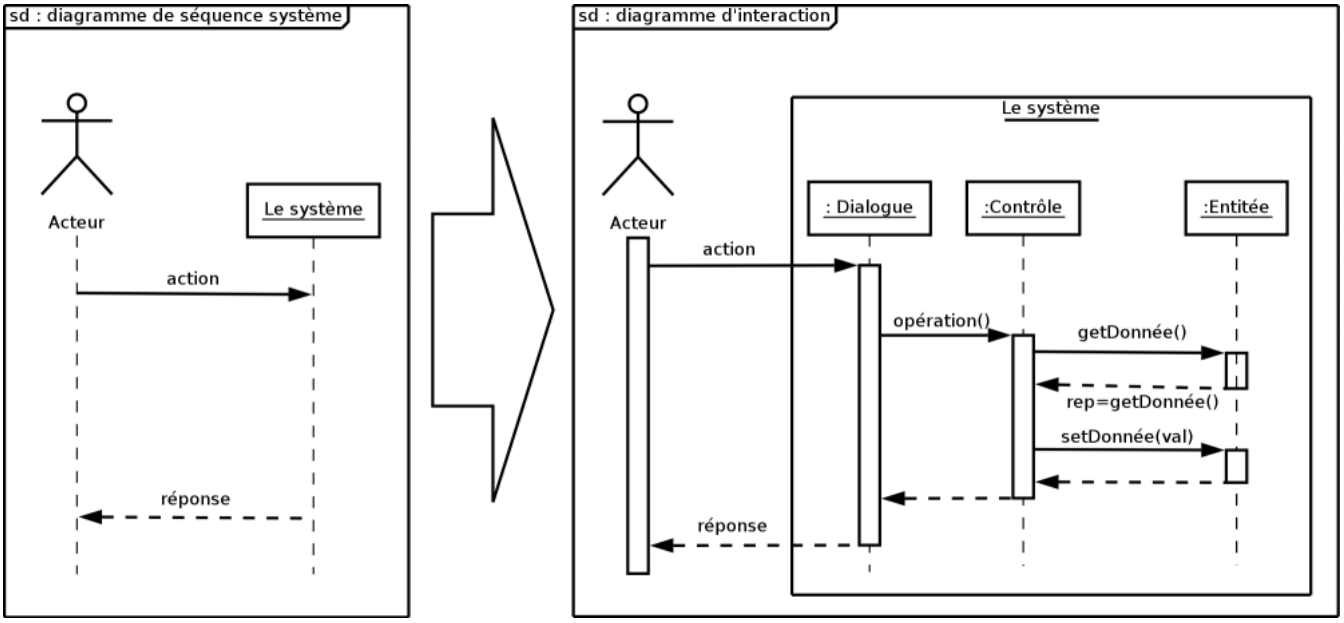



Figure 9.9: Le système des diagrammes de séquences système, vu comme une boîte noire, est remplacé par un ensemble d'objets en collaboration.

Par rapport aux diagrammes de séquences système de la section 9.2.2, nous remplaçons ici le système, vu comme une boîte noire, par un ensemble d'objets en collaboration (cf. figure 9.9). Ces objets sont des instances des trois types de classes d'analyse du diagramme de classes participantes, à savoir des dialogues, des contrôles et des entités.

Les diagrammes de séquences élaborés dans cette section doivent donc toujours respecter les règles édictées dans la section 9.3.2. Ces règles doivent cependant être transposées car, pour que deux objets puis interagir directement, il faut que :

- les classes dont ils sont issus soient en association dans le diagramme de classes participantes[1] ;
- l'interaction respecte la navigabilité de l'association en question.

### 9.4.2 Diagramme de classes de conception

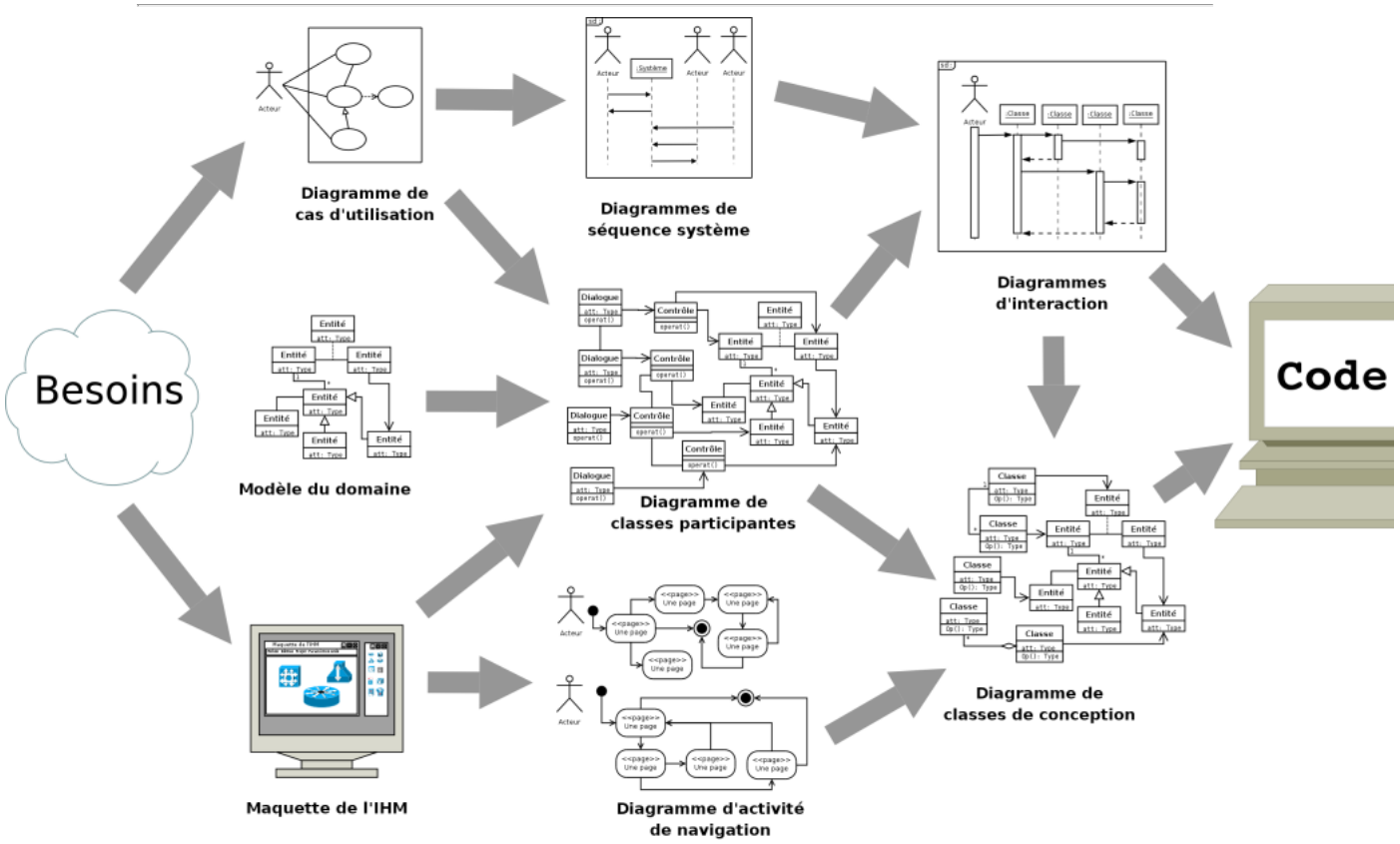



Figure 9.10: Chaîne complète de la démarche de modélisation du besoin jusqu'au code.

L'objectif de cette étape est de produire le diagramme de classes qui servira pour l'implémentation (figure 9.10). Une première ébauche du diagramme de classes de conception a déjà été élaborée en parallèle du diagrammes d'interaction (section 9.4.1). Il faut maintenant le compléter en précisant les opérations privées des différentes classes. Il faut prendre en comptes les choix techniques, comme le choix du langage de programmation, le choix des différentes librairies utilisées (notamment pour l'implémentation de l'interface graphique), etc.

Pour une classe, le *couplage* est la mesure de la quantité d'autre classes auxquelles elle est connectée par des associations, des relations de dépendances, etc. Durant toute l'élaboration du diagramme de classes de conception, il faut veiller à conserver un couplage faible pour obtenir une application plus évolutive et plus facile à maintenir. L'utilisation des *design patterns* est fortement conseillée lors de l'élaboration du diagramme de classes de conception.

Pour le passage à l'implémentation, referez vous à la section 3.6. Parfois, les classes du type entités ont intérêt à être implémentées dans une base de données relationnelle (cf. section 3.6.3).

---

1 Si elles ne sont pas en association, il doit au moins exister une relation de dépendance comme illustrée par la figure 3.21 de la section 3.3.10.

# Bibliographie

[1] Wikipédia, encyclopédie librement distribuable. Internet. http://fr.wikipedia.org/ wiki/ Accueil.

[2] Mamoun Alissali. Support de cours introduction au génie logiciel. Internet, 1998. http:// www-ic2.univ-lemans.fr/ ~alissali/ Enseignement/ Polys/ GL/ gl.html.

[3] Franck Barbier. *UML 2 et MDE*. Dunod, 2005.

[4] Donald Bell. Uml's sequence diagram. Internet, 2004. http://www-128.ibm.com/ developerworks/ rational/ library/ 3101.html#notes.

[5] Grady Booch, James Rumbaugh, and Ivar Jacobson. *Le guide de l'utilisateur UML*. Eyrolles, 2003.

[6] Eric Cariou. Support de cours "le langage de contraintes ocl". Internet, novembre 2003. http:// www.univ-pau.fr/ ~ecariou/ cours/ mde/ cours-ocl.pdf.

[7] Benoît Charroux, Aomar Osmani, and Yann Thierry-Mieg. *UML2*. Pearson Education France, 2005.

[8] Laurent Debrauwer and Fien Van der Heyd. *UML 2 Initiation, exemples et exercices corrigés*. eni, 2005.

[9]

Developpez.com. Club d'entraide des développeurs francophones. Internet. http://www.developpez.com/.

[10] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented sofware*. Addison-Wesley, 1995.

[11] Erik Gollot. Les cas d'utilisation. Internet, 2004. http:// ego.developpez.com/ uml/ tutoriel/ casUtilisation/.

[12] Pierre Gérard. Uml. Internet, 2007. http:// www-lipn.univ-paris13.fr/ ~gerard/ fr_ens_uml.html.

[13] Craig Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process*. Printice Hall, 1997.

[14] Hugues Malgouyres, Jean-Pierre Seuma-Vidal, and Gilles Motet. Règles de cohérence uml 2.0 (version 1.1). Internet, 2005. http:// www.lesia.insa-toulouse.fr/ UML/ CoherenceUML_v1_1_100605.pdf.

[15] Bernard Morand. Analyse et conception des systèmes d'information : Les diagrammes de unified modeling language (uml). Internet. http:// www.iutc3.unicaen.fr/ ~moranb/ cours/ acsi/ menucoo.htm.

[16] Pierre-Alain Muller and Nathalie Gaertner. *Modélisation objet avec UML*. Eyrolles, 2000.

[17] Object Management Group (OMG). Uml resource page. Internet. http://www.uml.org/.

[18] Object Management Group (OMG). Unified modeling language: Superstructure. Internet, août 2004. http://www.uml.org/.

[19] Object Management Group (OMG). Object constraint language. Internet, Mai 2006. http://www.uml.org/.

[20] Tom Penders. *Introduction à UML*. OEM, 2002.

[21] Laurent Piechocki. Uml en français. Internet. http://uml.free.fr/.

[22] Dan Pilone and Neil Pitman. *UML 2 en concentré*. O'Reilly, 2006.

[23] Pascal Roques. *UML - Modéliser un site e-commerce*. Les cahiers du programmeur. Eyrolles, 2002.

[24] Pascal Roques. *UML2 - Modéliser une application web*. Les cahiers du programmeur. Eyrolles, 2006.

[25] Pascal Roques. *UML2 par la pratique (étude de cas et exercices corrigés)*. Eyrolles, 5 edition, 2006.

[26] Pascal Roques and Franck Vallée. *UML en action*. Eyrolles, 2 edition, 2003.

[27] James Rumbaugh, Ivar Jacobson, and Grady Booch. *UML 2.0 Guide de référence*. CampusPress, 2004.

[28] Michel Volle. *De l'Informatique (Savoir vivre avec l'automate)*. Economica, 2006.

**Index**

---

*UML 2 – LAURENT AUDIBERT*
*La version finalisée, largement enrichie et corrigée de cette première ébauche de cours est parue, dans la collection Info+ chez les éditions Ellipses, sous le titre* ***UML 2 - de l'apprentissage à la pratique (cours et exercices)*** *(FNAC, amazon.fr)*
*Ce cours a déjà été consulté 3 861 812 fois. Ce document a été traduit de LaTeX par HeVeA*