LE LANGAGE UML

TODO :

- 3.5 : Cadres d'interaction et fragments
- 3.6 : Diagrammes complémentaires (collaboration, état, activité)
- 4.3 : Diagrammes complémentaires (objet, composants, déploiement)

v0.8.1.1 – 21/08/2009
peignotc(at)arqendra(dot)net / peignotc(at)gmail(dot)com

TABLE DES MATIÈRES

TABLE DES ANNEXES

TABLE DES ILLUSTRATIONS

Un grand nombre de diagrammes UML et de représentations d'éléments utilisés pour les figures du présent document sont extraites de captures d'écran de modélisations réalisées à l'aide du logiciel StarUML v5.0.2.1570 (http://staruml.sourceforge.net), plate-forme UML/MDA OpenSource.

Le logiciel StarUML – comme tout logiciel de modélisation – ayant ses limites et son interprétation de la norme, certaines de ces captures d'écran ont dû être retravaillées à l'aide d'un logiciel d'édition d'images afin de rester au plus proche des standards de représentation UML.

1 INTRODUCTION AU LANGAGE UML

1.1 INTRODUCTION AU GÉNIE LOGICIEL

Le développement de logiciels est une activité récente ayant débuté dans les années 70 pour devenir à l'heure actuelle l'un des piliers de l'économie mondiale. Durant cette période de passage de l'artisanat à l'industrie, les nécessités d'industrialisation (évolution perpétuelle du matériel, accroissement exponentiel des besoins en quantité et qualité, etc.) ont conduit à la mise en place du génie logiciel.

Le **génie logiciel** est un terme désignant l'ensemble des activités de conception et de mise en œuvre des produits et procédures ayant pour objectif de rationaliser la production de logiciels ainsi que leur suivi.
L'idée est de mettre en place des procédés, des méthodes, des normes, des techniques et des outils pour le développement de logiciels. Les objectifs évidents sont de produire des logiciels de qualité, répondant aux besoins exprimés, de prévoir et réduire les coûts et les délais, ainsi que de mettre en place des structures facilitant la maintenance.

1.2 CYCLE DE VIE D'UN PROJET

1.2.1 Généralités

La notion de projet apparaît lorsqu'un besoin en système informatisé est exprimé par une entreprise, afin de mettre en place le suivi de sa réalisation. Un projet possède un début (un besoin), et une fin (une réalisation).

Le **cycle de vie** d'un projet décrit les étapes successives de développement de ce projet, partant du besoin pour arriver à la réalisation : analyse des besoins, spécification, conception, codage, intégration, maintenance, retrait.
Chaque étape du cycle de vie est suivie d'une opération de vérification qui conditionne le passage à l'étape suivante.

Le cycle de vie sert de base de travail pour le planning de développement du système informatisé. Il permet ainsi d'avoir une vision sur l'état d'avancement du projet, ainsi que d'effectuer un contrôle continu des résultats produits. Il joue par ailleurs le rôle de contrat entre le client et les fournisseurs.

Le cycle de vie favorise la détection des erreurs le plus tôt possible lors du développement, car de la bonne mise en place des étapes amont dépend la bonne réalisation des étapes finales ; en effet, une modification en fin de cycle entraîne une remise à jour de tous les produits des phases précédentes. De surcroît, il est plus facile de modifier les spécifications d'un système que d'en modifier la réalisation (le code).

1.2.2 Différents cycles de vie

1.2.2.1 Prototypage rapide

Le cycle de vie type **prototypage rapide** est particulier :
- sélectionner les fonctions à valider ;
- construire le prototype ;
- évaluer et modifier ce prototype.

Son intérêt est de s'assurer très rapidement des besoins du client en offrant une vision concrète bien que partielle du produit ; il permet ainsi de prouver la faisabilité du produit, d'évaluer grossièrement les performances et de valider des choix de conception.

Son inconvénient est d'être une solution coûteuse, principalement en phase de maintenance. De plus, le temps de développement du prototype n'est pas du tout représentatif du temps de développement du système réel.

1.2.2.2 « Code and fix »

Le cycle de vie **code and fix** est défini en 2 étapes : coder et corriger (d'où le nom).

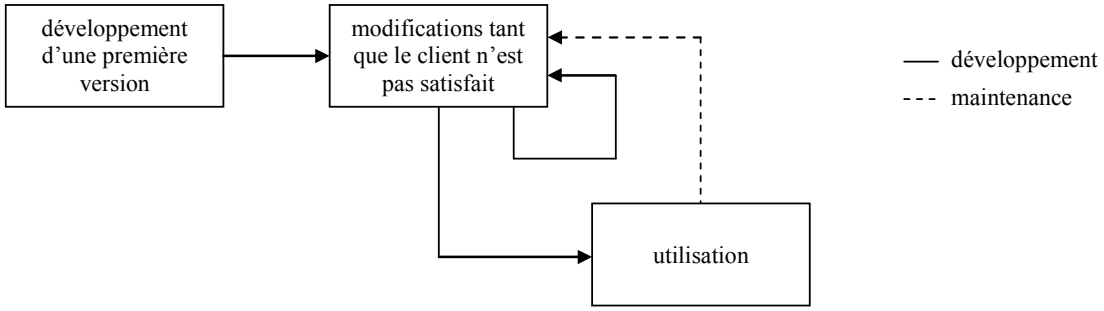

Figure 1.1 : cycle de vie « code and fix »

Son intérêt est de réaliser une mise en route très rapide en fonctionnement chez le client.
Son inconvénient est que le résultat obtenu est très éloigné du besoin, que le code perd sa structuration initiale, et que celui-ci est coûteux à modifier.

1.2.2.3 Cycle « en V »

Le **cycle en V** possède la particularité de positionner en parallèle les activités de production et les activités de contrôle.

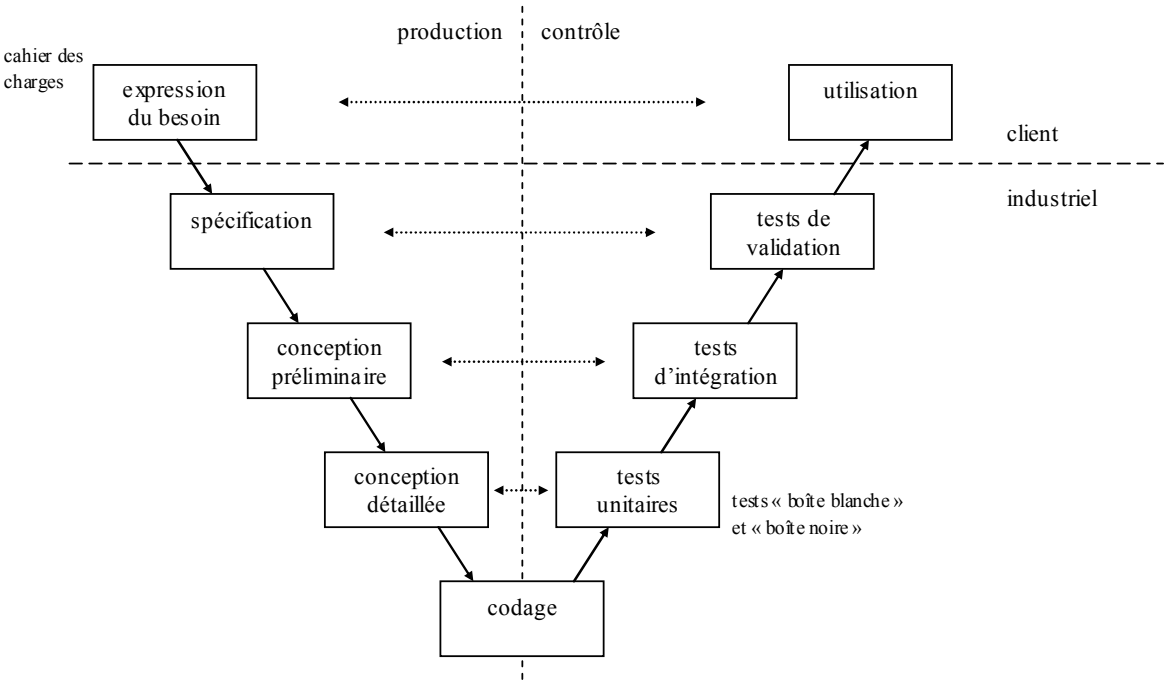

Figure 1.2 : cycle de vie « en V »

Les différentes étapes de ce cycle de vie sont les suivantes :

- cahier des charges ;
  Lorsque des besoins en système informatisé se font ressentir dans une entreprise, le client synthétise ceux-ci et rédige un cahier des charges, qui décrit ainsi le *pourquoi* du système.
  Ce document est ensuite soumis à divers prestataires concepteurs de systèmes informatisés, qui font alors une proposition de développement, incluant l'évaluation des coûts et des délais de réalisation. Une fois l'entreprise décidée sur le prestataire, le cahier des charges est ensuite remanié lors d'un travail de collaboration entre l'entreprise et le prestataire, notamment pour clarifier les éléments propres au domaine d'application du système à concevoir, lequel peut parfois être très spécifique (domaine médical par exemple).

- spécification des besoins ;
  La spécification des besoins permet de délimiter le système en décrivant les fonctions majeures de celui-ci ; elle décrit ainsi le *quoi* du système (i.e. *que faire ?*). Les besoins exprimés dans le cahier des charges sont re-formulés, hors de tout vocabulaire informatique, en utilisant un formalisme clair et vulgarisé, et sont proposés pour vérification au client afin de s'assurer que les besoins ont bien été compris.
- conception préliminaire ;
  La conception préliminaire permet de déterminer l'architecture informatique globale du système (configuration matérielle, choix des langages et outils de développement, découpage en modules et définition des modules, etc.) et décrit ainsi le *comment* du système.
- conception détaillée ;
  Le contenu (algorithme) de chaque module défini lors de la conception préliminaire est décrit précisément.
- codage ;
  Le codage correspond à l'implémentation des différents modules décrits lors de la conception détaillée, à l'implémentation des différentes IHMs [1], et à la programmation du reste de l'application logicielle du système informatisé, le tout en utilisant les langages informatiques choisis.
- tests unitaires ;
  Les tests unitaires permettent de contrôler le fonctionnement de chaque module (séquences, boucles, alternatives) en réalisant des tests *boîte blanche* (le code interne du module est visible) et *boîte noire* (le code interne est invisible, seule l'interface (prototype) du module est utilisée).
- tests d'intégration ;
  Les modules précédemment testés sont assemblés progressivement les uns avec les autres, afin de tester leur interopérabilité.
- tests de validation ;
  Les tests de validation s'assurent que le produit développé est conforme aux spécifications du cahier des charges.
- utilisation.
  L'installation, la configuration et la mise en situation au sein de l'entreprise du système développé permettent de vérifier que celui-ci répond ainsi aux besoins du client.

Ces différentes étapes se répartissent temporellement sur des périodes qui se chevauchent, afin de pouvoir minimiser les délais en travaillant sur différents travaux en parallèle et pour permettre une réévaluation des étapes antérieures lors de la mise en évidence d'erreurs.

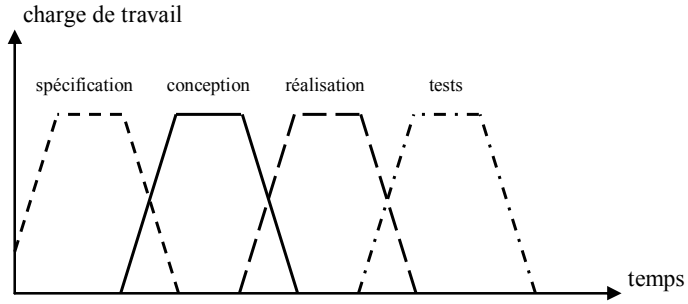

Figure 1.3 : répartition temporelle des étapes du cycle « en V »

L'intérêt de ce cycle de vie est d'imposer les activités de contrôle pour chaque étape de production.
Son inconvénient est de proposer une mise en œuvre tardive de l'implémentation, des tests et de la validation, ce qui peut rendre coûteux certaines erreurs de spécification.

1.2.2.4 Cycle incrémental

Le **cycle de vie incrémental** permet un développement sous forme de prototypes consécutifs, et est adapté au développement mettant en œuvre la programmation orientée objet [2].

[1] IHM : Interface Homme/Machine.
[2] POO : Programmation Orientée Objet.

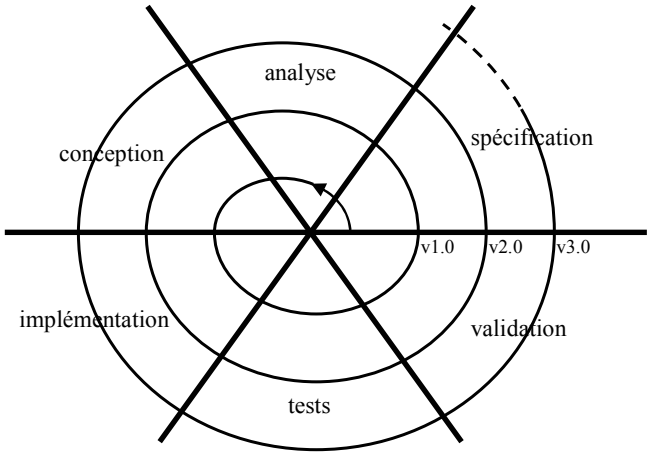

Figure 1.4 : cycle de vie incrémental

Ce cycle de vie est notamment constitué des phases suivantes :

- spécification : correspond à la définition de *ce que c'est* ;
  Elle permet de délimiter précisément le système, de décrire les différentes manières de l'utiliser du point de vue des utilisateurs.
  Cette démarche est orientée utilisateur et permet ainsi de concevoir le système en tant que *réponse à un ou des besoin(s)*.
- analyse : correspond à la définition de *ce que doit faire le système*.
  Elle permet de mettre en évidence les caractéristiques des objets [1] suivant 3 axes :
  - fonctionnel : ce que fait l'objet ;
  - statique : la structure de l'objet ;
  - dynamique : les différents états de l'objet dans le temps (comportement).

Son intérêt est de permettre un développement échelonné des différentes fonctionnalités et d'affiner les spécifications en fonction des résultats obtenus.

Son inconvénient est d'être applicable principalement avec des produits suffisamment « souples » pour prendre en compte des spécifications non prévues au départ.

1.3 UML ET GÉNIE LOGICIEL

La plupart des différents cycles de vie d'un projet font ressortir les notions de *spécification*, *conception* et *analyse* qui, depuis les années 80, ont été mises en œuvre suivant différents procédés ; parmi ceux-ci, se trouve le langage UML.

1.3.1 Définition

Le langage **UML** (Unified Modeling Language [2]) est un langage de modélisation qui a pour but de faciliter les transitions, lors du développement d'un projet, du besoin originel à la phase d'implémentation.

Ce langage, fusion de 3 méthodes orientées objet très utilisées (OMT [3], Booch et OOSE [4]) dont le développement a débuté en 1994, et qui a été standardisé en 1997 par l'OMG [5] dans sa version 1.1 [6], s'appuie essentiellement sur la technologie objet et les concepts qu'elle véhicule.

Néanmoins, l'utilisation de UML n'est pas restreinte au développement de systèmes informatisés, mais peut servir au développement de systèmes de gestion, tout comme à la résolution de problèmes d'organisation de tout style. De plus, UML, de par sa représentation essentiellement graphique, se veut extrêmement vulgarisé et intuitif, et est utilisable par les êtres humains ainsi que les machines.

Nb : On parle volontairement de langage UML et non pas de méthode, car UML propose des outils et des concepts de modélisation, et non pas une méthodologie à respecter impérativement.

[1] Au sens POO du terme.
[2] Langage de Modélisation Unifié.
[3] Object Modeling Technique.
[4] Object Oriented Software Engineering.
[5] Object Management Group.
[6] En février 2009, UML a déjà subi une révision importante et on travaille désormais avec UML 2.2.

1.3.2 Objectifs

Au final, le langage UML est une synthèse de tous les concepts et formalismes méthodologiques les plus utilisés, pouvant être utilisé, grâce à sa simplicité et à son universalité, comme langage de modélisation pour la plupart des systèmes devant être développés [1].

Le langage UML permet ainsi d'apporter des solutions lors du développement de systèmes informatisés :

- décomposer le processus de développement en distinguant la phase d'analyse (aspects fonctionnels) de la phase de réalisation (aspects technologiques et architecturaux) ;
- décomposer le système en sous-systèmes plus facilement abordables : réduction de la complexité, répartition du travail, réutilisation des sous-systèmes ;
- utiliser une technologie de haut niveau proche de la réalité pour aborder le développement.

La modélisation proposée par le langage UML se réalise principalement sous forme graphique, en usant de divers types de diagrammes spécifiques [2], répartis en deux groupes :

- *structure* du système (statique) ;
  - diagramme de cas d'utilisation : structure des fonctionnalités ;
  - diagramme de classes : structure des entités ;
  - diagrammes d'objets : cas particulier des structures de classe ;
  - diagrammes de composants : structure de l'architecture logicielle ;
  - diagrammes de déploiement : structure de l'architecture matérielle ;
  - diagramme des paquetages (UML v2) ;
  - diagrammes de structure composite (UML v2).
- *comportement* du système (dynamique).
  - diagrammes d'états-transitions : cycle de vie des objets ;
  - diagrammes d'activités : ;
  - *interactions* du système :
    - diagrammes de séquence : scénarios des fonctionnalités ;
    - diagrammes de collaboration/communication : interactions entre objets ;
    - diagrammes de vue globale des interactions (UML v2) ;
    - diagrammes de temps (UML v2).

Certains de ces diagrammes sont indépendants, alors que d'autres servent de base de travail ou bien sont la continuité d'autres diagrammes.

[1] Informatiques ou autres.
[2] 9 diagrammes différents pour UML v1.4, et 13 pour UML v2.2.

2 LE DIAGRAMME DE CAS D'UTILISATION

2.1 DÉFINITION

Le **diagramme de cas d'utilisation** décrit les fonctionnalités d'un système d'un point de vue utilisateur, sous la forme d'actions et de réactions ; l'ensemble des fonctionnalités est déterminé en examinant les besoins fonctionnels de tous les utilisateurs potentiels.

Le but est d'identifier :

- les catégories d'utilisateurs : chacune d'entre elles, appelée **acteur**, est susceptible de mettre en œuvre une ou plusieurs fonctionnalités du système ;
- les besoins du système : chaque fonctionnalité, appelée **cas d'utilisation**, doit répondre à l'un des besoins nécessités par une ou plusieurs catégories d'utilisateurs.

Le diagramme des cas d'utilisation se base sur le cahier des charges pour être construit ; il fait donc partie, en termes de gestion de projet, de la *spécification* du système.

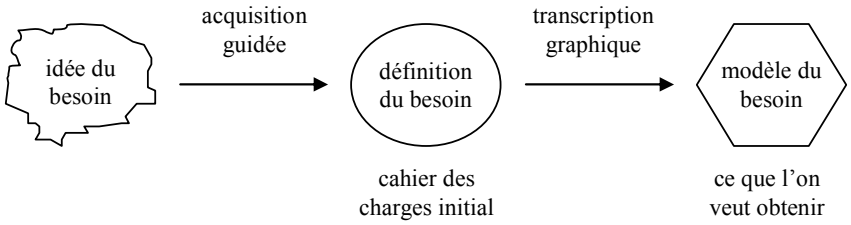

Figure 2.1 : processus de construction du diagramme de cas d'utilisation

L'objectif du diagramme de cas d'utilisation est de recentrer la problématique du cahier des charges sur les besoins des utilisateurs, l'idée sous-jacente étant qu'un système et ses fonctionnalités n'ont de raison d'être que s'ils répondent à un ou des besoins.

Le diagramme des cas d'utilisation, qui est unique, synthétise alors les résultats de cette étude en faisant apparaître tous les acteurs, tous les cas d'utilisation, et les relations entre ces divers éléments.

2.2 CONSTRUCTION DU DIAGRAMME

2.2.1 Les acteurs

2.2.1.1 Définition

Un **acteur** symbolise les actions qu'une entité autonome et extérieure peut avoir avec le système dont on désire décrire le fonctionnement. Plus globalement, un acteur correspond à une personne ou une machine extérieure, une tâche ou bien une interaction avec le système.

On peut distinguer trois types d'acteurs :

- humain : utilisateur du système, au travers des différentes interfaces (IHM [1] logicielle ou matérielle) ; exemple : client, administrateur, technicien, etc. ;

[1] IHM : Interface Homme/Machine.

- logiciel : entité logicielle existante et fonctionnelle qui communique avec le système grâce à une interface logicielle ; exemple : application de gestion, base de données, etc. ;
- matériel : entité matérielle qui exploite les données du système, ou est pilotée par le système ; exemple : imprimante, robot, serveur, etc.

Plusieurs utilisateurs potentiels peuvent être représentés par un seul acteur si leur rôle par rapport au système est identique ; exemple : deux techniciens (Pierre et Paul) seront représentés par un seul acteur « technicien ».
De la même manière, un utilisateur peut être représenté par plusieurs acteurs différents, s'il est susceptible d'utiliser le système en jouant des rôles différents ; exemple : une personne physique (Pierre) tour-à-tour simple utilisateur et technicien sera représentée par deux acteurs « utilisateur » et « technicien ».

Ainsi, un acteur peut aussi bien désigner une personne ou un système unique, qu'un groupe de personnes ou de systèmes ayant des traits communs mis en œuvre dans le système.

Chaque acteur doit interagir avec au moins un des cas d'utilisation. Les relations mises en évidence décrivent ainsi les possibilités des acteurs sur le système.

Du point de vue du système, on peut différencier deux types d'acteurs :

- primaire : utilise le système ; met en œuvre 1 ou plusieurs fonctionnalités du système ; répond aux différentes questions : *à qui va servir le système ? qui va l'utiliser ? qui le système doit-il aider ?* etc.
- secondaire : administre le système ; possède un rôle d'administration et de maintenance, paramètre le système et lui fournit toutes les informations nécessaires à son bon fonctionnement pour les acteurs primaires ; répond aux différentes questions : *qui gère le système ? qui l'administre ? qui le paramètre ?* etc.

2.2.1.2 Représentation graphique

Un acteur se représente graphiquement différemment selon qu'il s'agisse d'un acteur humain ou non-humain, respectivement par un personnage ou un rectangle. À cette représentation graphique est associé un nom correspondant au type d'acteurs représenté.

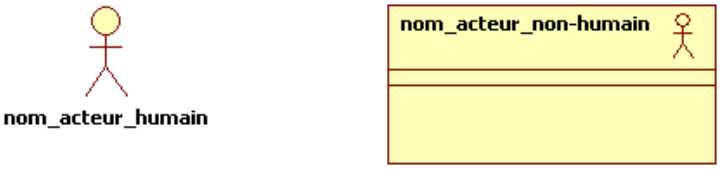

Figure 2.2 : représentation graphique d'un acteur

Ex. : Un acteur représentant n'importe quel administrateur pouvant être amené à intervenir sur le système.

Figure 2.3 : exemple d'un acteur

2.2.2 Les cas d'utilisation

2.2.2.1 Définition

Un **cas d'utilisation** désigne une fonctionnalité visible de l'extérieur du système dont on désire décrire le fonctionnement. Un cas d'utilisation doit répondre à un besoin en définissant une partie du comportement attendu du système sans révéler sa structure interne.

Un cas d'utilisation regroupe une famille de scénarios d'utilisation du système ; exemple : *configurer les paramètres* est un cas d'utilisation, alors que *configurer les paramètres d'initialisation* est un scénario du cas d'utilisation précédent.

L'exécution de chaque cas d'utilisation est indépendante des autres, ce qui implique qu'un cas d'utilisation doit être vu, la plupart du temps, comme une utilisation ayant un début et une fin propres.
Cependant, dans certains cas, un cas d'utilisation représente une partie d'une fonctionnalité dont l'exécution peut être associée à l'exécution d'autres cas d'utilisation.

2.2.2.2 Représentation graphique

Un cas d'utilisation est représenté par une ellipse. À cette représentation graphique est associé un nom correspondant à l'utilisation représentée ; de manière générale ce nom correspond à la description d'une action et comprend donc un verbe à l'infinitif.

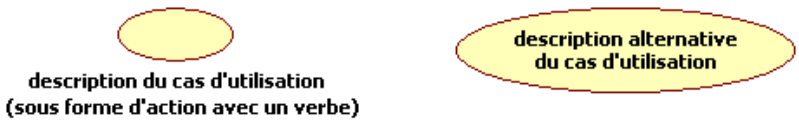

Figure 2.4 : représentation graphique d'un cas d'utilisation

Ex. : Un cas d'utilisation correspondant à l'action de configurer les paramètres.

Figure 2.5 : exemple d'un cas d'utilisation

2.2.3 Les relations

2.2.3.1 Définition

Une **relation** précise une interaction unidirectionnelle ou bidirectionnelle mettant en jeu 2 éléments de type acteur ou cas d'utilisation ; celle-ci peut être de deux types :

- entre acteur et cas d'utilisation : représente l'interaction du système avec l'extérieur, appelée *association* ;
- entre deux entités de même type (deux cas d'utilisation ou deux acteurs) : représente le comportement entre les entités (généralisation, dépendance, inclusion ou extension) [1].

2.2.3.2 Représentation graphique

Une relation est représentée graphiquement par un lien entre les deux entités qu'elle relie. Ce lien peut être fléché (flèche à tête classique [2]), le sens de la flèche symbolisant alors le sens de l'interaction (*qui est actif ? passif ?*), et pas un quelconque flux de données. Lorsque l'interaction est bidirectionnelle, le lien ne porte aucune flèche.

Figure 2.6 : représentation graphique d'une relation

Ex. : Un cas d'utilisation décrivant la possibilité pour n'importe quel administrateur de configurer les paramètres du système.

Figure 2.7 : exemple d'une relation

2.2.4 Le diagramme

2.2.4.1 Définition

Au final, le diagramme de cas d'utilisation doit faire apparaître :

- tous les acteurs susceptibles d'intervenir sur le système ;
- tous les cas d'utilisation devant représenter toutes les différentes fonctionnalités du système ;
- toutes les relations existantes décrivant les interactions entre les acteurs et les cas d'utilisation.

Ce diagramme doit être pensé comme étant définitif et exhaustif, et doit donc faire état de tout ce qui est attendu du système. Par ailleurs, il représente le système en tant que *réponse à des besoins* et pas comme une entité unique. Le

[1] cf. 2.4.
[2] Une flèche à tête triangulaire symbolise autre chose qu'une relation dans le cadre de la programmation objet (la généralisation, cf. 2.4.1).

système lui-même n'est donc pas mentionné sur le diagramme, il n'est représenté que par les besoins auxquels il répond (les fonctionnalités) ; c'est l'ensemble des fonctionnalités représentées qui décrivent alors le système.

2.2.4.2 Représentation graphique

Ce diagramme est la synthèse graphique de tous les éléments qu'il doit représenter.

Pour s'assurer de sa cohérence, plusieurs règles doivent être observées :

- chaque entité (acteur, cas d'utilisation) ne doit être représentée qu'une seule fois ;
- un acteur a obligatoirement une relation avec au moins un cas d'utilisation ;
- un cas d'utilisation a obligatoirement une relation avec au moins un acteur (sauf dans le cas des relations entre cas d'utilisation [1]).

Ex. : Cahier des charges et diagramme de cas d'utilisation d'un système informatisé implanté dans une borne automatique permettant d'acheter un billet (métro, train, etc.) qui est située à l'intérieur même d'une gare.

> Lorsque la borne est active, l'utilisateur sélectionne, grâce au clavier, sa destination, puis la classe du billet (première ou seconde). Ensuite, il paie le montant demandé en utilisant soit sa carte bleue qu'il insère dans la borne et en composant son code, soit en utilisant de la monnaie. Puis le billet est imprimé par la borne et l'utilisateur peut alors le retirer. Par défaut, on considère que la gare de départ est la gare où l'on se trouve, mais l'utilisateur peut éventuellement décider de modifier la gare de départ.

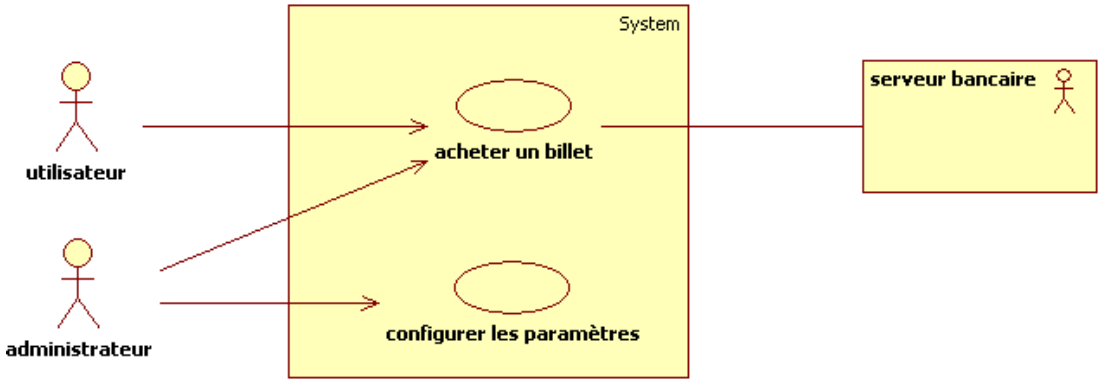

Figure 2.8 : exemple d'un diagramme de cas d'utilisation

Ce diagramme doit ainsi synthétiser l'intégralité des fonctionnalités attendues du système. Très occasionnellement, il est possible que le diagramme de cas d'utilisation ne décrive qu'une partie du système (généralement par souci de lisibilité).

2.2.4.3 Description textuelle des flots d'évènements

Pour parfaire cette description, chaque cas d'utilisation peut être décrit en langage naturel ou en pseudo-code, afin de mettre en évidence les flots d'évènements mis en jeu. On peut en distinguer trois types :

- flot d'évènements principal : description du fonctionnement du cas d'utilisation dans le cas du scénario le plus probable ;
- flots d'évènements alternatifs : description du fonctionnement du cas d'utilisation pour une partie du scénario différente du scénario le plus probable, venant ainsi se substituer à une partie de la description du flot d'évènement principal ;
- flots d'évènements exceptionnels : description du fonctionnement du cas d'utilisation pour une partie du scénario supplémentaire et optionnelle, venant ainsi se rajouter à la description du flot d'évènement principal et signalant généralement la fin du scénario.

Cette description textuelle permet ainsi de préciser l'ensemble du déroulement du cas d'utilisation :

- les préconditions (état avant) et les postconditions (état après) ;
- quand et comment le cas d'utilisation débute et se termine ;
- les relations entre le cas d'utilisation et les acteurs (ou entre plusieurs cas d'utilisation [1]) ;
- les données nécessaires pour ce cas d'utilisation ;
- les répétitions de comportement ;
- d'éventuelles situations alternatives ou exceptionnelles.

[1] cf. 2.4.

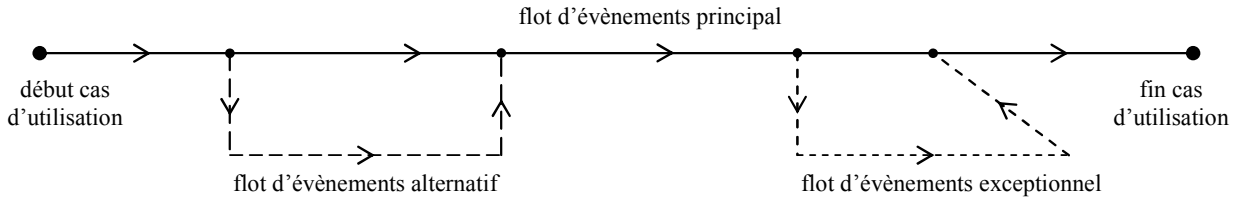

Figure 2.9 : description des flots d'évènements

Ex. : Le cas d'utilisation décrivant l'action d'*acheter un billet de train* en utilisant une borne automatique.

- flot d'évènements principal : L'utilisateur sélectionne, grâce au clavier, sa destination, la classe du billet (première ou seconde), insère sa carte bleue, paie le montant demandé, puis enfin retire le billet qui est imprimé par la borne.
- flots d'évènements alternatifs : L'utilisateur choisit de payer en utilisant de la monnaie plutôt que sa carte bleue / L'utilisateur décide d'annuler la procédure en cours / etc..
- flots d'évènements exceptionnels : L'utilisateur modifie la gare de départ, puis suit le fonctionnement principal (sélection de la destination, etc.) / etc.

2.3 DIAGRAMMES COMPLÉMENTAIRES

Certains diagrammes optionnels permettent de préciser la description générale du fonctionnement du système. Les diagrammes de contexte permettent notamment de préciser la position du système dans son environnement.

2.3.1 Le diagramme de contexte statique

Le **diagramme de contexte statique** permet de positionner le système dans son environnement selon un point de vue matériel. Le système est donc décrit physiquement, et non pas en termes de fonctionnalités.

De plus, pour chaque type d'élément matériel extérieur au système, il est précisé les nombres minimal et maximal d'éléments, appelés *cardinalités*, qui sont mis en jeu.

Ex. : Le diagramme de contexte statique d'une borne automatique permettant d'acheter un billet.

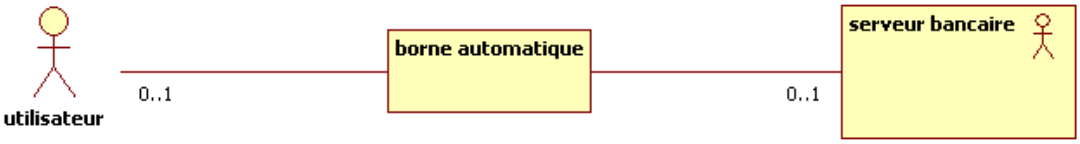

Figure 2.10 : exemple d'un diagramme de contexte statique

Aucun ou au plus 1 utilisateur utilise la borne automatique (à un instant donné).

Aucun ou au plus 1 serveur bancaire est utilisé par la borne automatique.

2.3.2 Le diagramme de contexte dynamique

Le **diagramme de contexte dynamique** permet de positionner le système dans son environnement selon le point de vue des communications. Il reprend les éléments du contexte statique et précise les échanges d'informations qui sont réalisés entre le système et les éléments matériels extérieurs au système. Le système est donc décrit physiquement et logiquement.

Ex. : Le diagramme de contexte dynamique d'une borne automatique permettant d'acheter un billet.

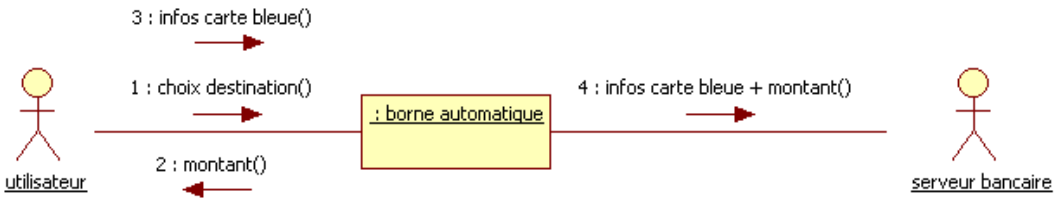

Figure 2.11 : exemple d'un diagramme de contexte dynamique

2.4 COMPLÉMENTS SUR LES RELATIONS

2.4.1 La généralisation

Une relation de **généralisation** entre deux entités est une relation unidirectionnelle symbolisant que l'une des entités, l'entité-fille, possède l'intégralité des propriétés et/ou du comportement de l'autre entité, l'entité-parente, auxquels s'ajoutent les spécificités de l'entité-fille (nda : on parle alors de *spécialisation*). La généralisation ne peut être appliquée qu'entre entités de même nature : 2 acteurs ou 2 cas d'utilisation.

La généralisation est représentée par une flèche à tête triangulaire reliant les 2 entités dans le sens *entité-fille* → *entité-parente*.

Figure 2.12 : représentation graphique d'une généralisation

On peut ainsi définir une généralisation entre deux acteurs, signifiant que l'acteur-fils possède toutes les propriétés de l'acteur-parent, ce qui inclut les relations avec les cas d'utilisation. Par ailleurs, l'acteur-fils possède en plus ses propres propriétés et relations avec d'autres cas d'utilisation.

Ex. : Tout administrateur, en plus de ses possibilités, peut aussi acheter un billet et donc se comporter tel un utilisateur.

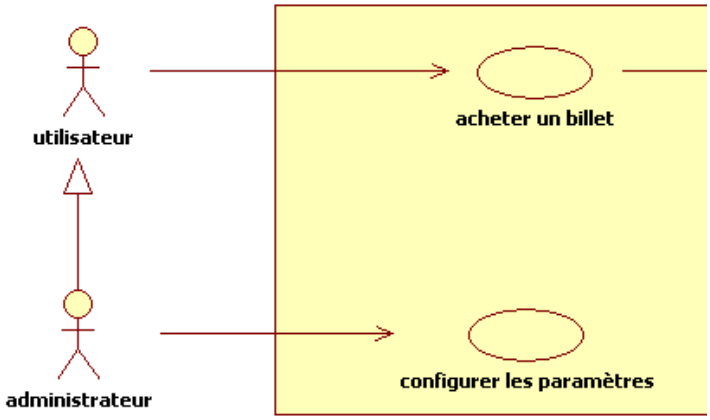

Figure 2.13 : exemple d'une généralisation entre acteurs

On peut définir une généralisation entre deux cas d'utilisation, précisant ainsi que le cas d'utilisation-fils propose le même comportement que le cas d'utilisation-parent mais dans une version spécialisée.

Ex. : Le cas d'utilisation *configurer les paramètres par défaut* est une spécialisation du cas d'utilisation *configurer les paramètres*.

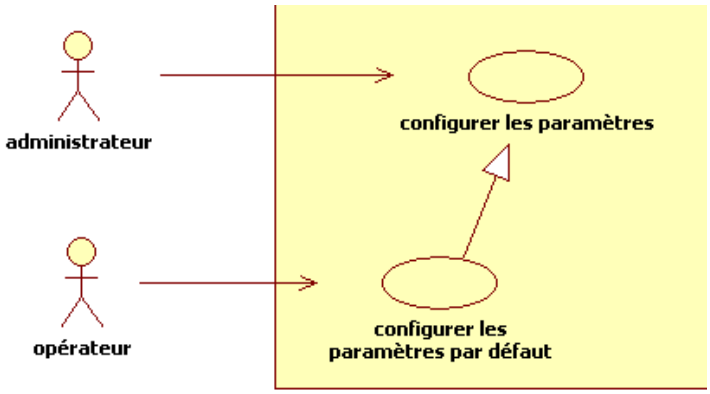

Figure 2.14 : exemple d'une généralisation entre cas d'utilisation

2.4.2 L'inclusion

Une relation d'**inclusion** entre deux cas d'utilisation est une relation unidirectionnelle symbolisant que l'un des cas d'utilisation, le cas d'utilisation-inclusif, comprend l'intégralité de l'autre cas d'utilisation, le cas d'utilisation-inclus, auquel s'ajoutent d'autres éléments de comportement. Le cas d'utilisation-inclusif est donc un sur-ensemble du cas d'utilisation-inclus.

L'inclusion est représentée par une flèche classique à trait pointillé reliant les 2 cas d'utilisation dans le sens *cas d'utilisation-inclusif → cas d'utilisation-inclus*, précisée avec le prototype `<<include>>`.

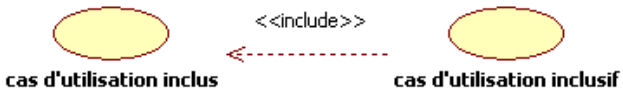

Figure 2.15 : représentation graphique d'une inclusion

L'inclusion a un caractère obligatoire, et le cas d'utilisation-inclusif peut préciser la position au sein de son propre comportement du cas d'utilisation inclus.

Cette relation est généralement utilisée pour décomposer le comportement d'un cas d'utilisation en plusieurs sous-cas d'utilisation et permettre ainsi de définir une portion de comportement, le cas d'utilisation-inclus, qui est commune à plusieurs cas d'utilisation.

Ex. : Le cas d'utilisation *configurer les paramètres* inclut la portion de comportement *authentifier l'administrateur*.

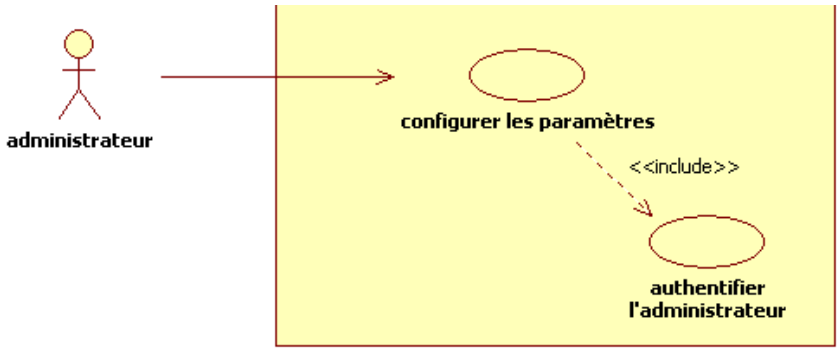

Figure 2.16 : exemple d'une relation d'inclusion

2.4.3 L'extension

Une relation d'**extension** entre deux cas d'utilisation est une relation unidirectionnelle symbolisant que l'un des cas d'utilisation, le cas d'utilisation-étendu, ajoute à son propre comportement l'intégralité du comportement de l'autre cas d'utilisation, le cas d'utilisation-de base. Le cas d'utilisation-étendu est donc un sur-ensemble du cas d'utilisation-de base.

L'extension est représentée par une flèche classique à trait pointillé reliant les 2 cas d'utilisation dans le sens *cas d'utilisation-étendu → cas d'utilisation-de base*, précisée avec le prototype `<<extend>>`.

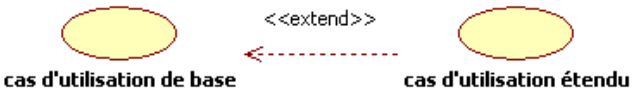

Figure 2.17 : représentation graphique d'une extension

L'extension peut être soumise à une condition, et celle-ci est alors précisée.

Cette relation est généralement utilisée pour modéliser un cas d'utilisation principal, le cas d'utilisation de base, et une extension de son comportement, le cas d'utilisation étendu, celle-ci proposant donc une variante facultative ou alternative.

Ex. : Le cas d'utilisation *acheter un billet précommandé sur internet* est une extension du cas d'utilisation *acheter un billet*.

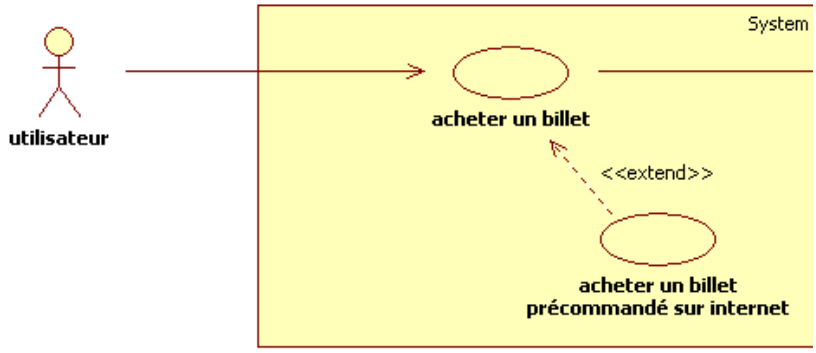

Figure 2.18 : exemple d'une relation d'extension

2.4.4 Exemple de diagramme de cas d'utilisation complet

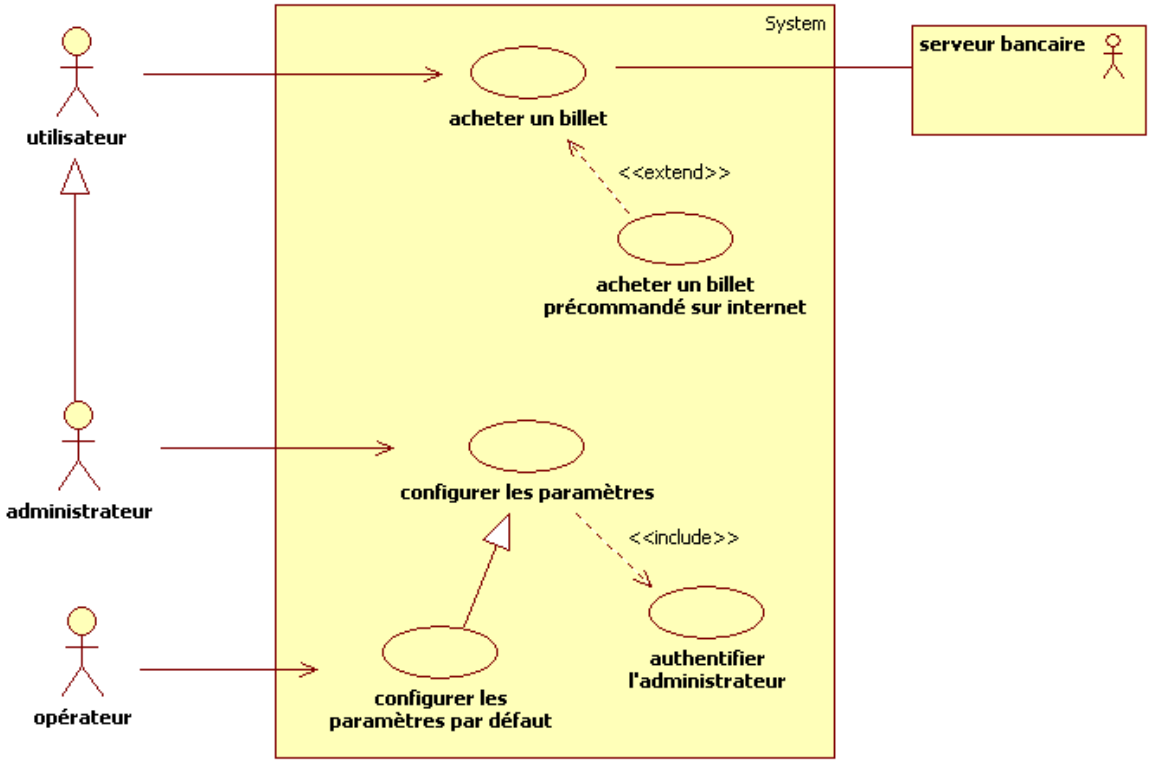

Figure 2.19 : exemple d'un diagramme de cas d'utilisation complet

---

Le diagramme de cas d'utilisation permet donc d'affiner le contenu du cahier des charges, afin de déterminer les limites du système en présentant le système dans son environnement, et en se recentrant sur le besoin *utilisateur*, donc sur le *quoi* et aucunement sur le *comment* ; il permet de s'assurer, par voie de conséquence, que le système répond bien aux exigences du client.

3 LES DIAGRAMMES DE SÉQUENCE

3.1 DÉFINITION

Les **diagrammes de séquence** décrivent le déroulement de chaque cas d'utilisation, en montrant la façon dont les diverses entités mises en œuvre dans les cas d'utilisation interagissent et collaborent dans le temps afin de réaliser les fonctionnalités attendues.

Le but est de déterminer :
- les divers entités, appelées objets, mises en jeu dans la réalisation d'une fonctionnalité ;
- les interactions entre ces divers objets ;
- le déroulement dans le temps de ces interactions.

Le diagramme de séquence est en réalité un cas particulier de *diagramme d'interaction* [1] ayant pour but de mettre en avant l'aspect chronologique des interactions décrites.

Typiquement, chaque cas d'utilisation déterminé dans le diagramme des cas d'utilisation fait l'objet d'une étude temporelle des interactions en utilisant un diagramme de séquence. Par conséquent, on doit avoir autant de diagrammes de séquence que de cas d'utilisation ; occasionnellement, un cas d'utilisation peut être décrit par plusieurs diagrammes de séquence afin de clarifier l'ensemble.

Les diagrammes de séquence sont donc la suite logique du diagramme des cas d'utilisation. En fonction du niveau de détail désiré, on peut les utiliser de deux manières différentes :
- forme simplifiée : description sommaire du cas d'utilisation en se basant sur la description textuelle – fin de la phase de *spécification* du système ;
- forme détaillée : présentation d'une ébauche de la structure du code objet – phase de *conception préliminaire*.

3.2 CONSTRUCTION DU DIAGRAMME

3.2.1 Les objets

3.2.1.1 Définition

Un **objet** mis en œuvre dans un diagramme de séquence peut symboliser :
- un acteur, humain ou non-humain ;
- le système, ou une partie ou composante de celui-ci.

3.2.1.2 Représentation graphique

Un objet est représenté par un rectangle, ainsi qu'une ligne pointillée verticale partant de ce rectangle et dirigée vers le bas qui représente ainsi la ligne de vie de l'objet. Dans le cas d'un objet symbolisant un acteur, le rectangle peut être remplacé par un personnage.

[1] Tout comme le diagramme de collaboration/communication.

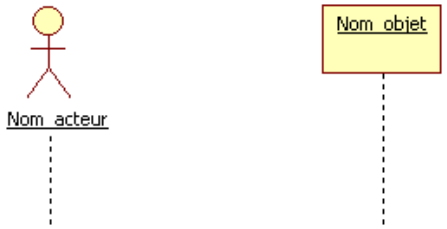

Figure 3.1 : représentation d'un objet

3.2.2 Les interactions

3.2.2.1 Définition

Une **interaction** permet d'exprimer une collaboration ou une communication nécessaire à la mise en œuvre de la fonctionnalité étudiée. De manière générale, une interaction est vue comme l'envoi d'un message entre un objet émetteur et un objet destinataire.

Il est important de bien garder à l'esprit qu'une interaction n'est pas représentative d'un flux de données, mais d'un message ; lequel peut en revanche mettre en jeu un flux de données, de sens identique ou différent à celui du message.

Une interaction peut être interne, c'est-à-dire être émise par un composant d'un objet à destination d'un autre composant du même objet ; en ce cas, le message est envoyé par l'objet à lui-même ; on parle alors de *message interne* ou *message réflexif*.

3.2.2.2 Représentation graphique

Une interaction est représentée par une flèche horizontale à tête classique [1], orientée de l'objet émetteur vers l'objet destinataire du message, précisée de l'intitulé du message.

Figure 3.2 : représentation d'une interaction

Il faut noter qu'un message est de caractère synchrone ou asynchrone, selon que l'objet objet émetteur soit ou pas bloqué durant la prise en compte du message par l'objet destinataire [2].

3.2.3 Les diagrammes : approche générale

Un diagramme de séquence doit faire apparaître :

- tous les objets mis en œuvre dans la réalisation de la fonctionnalité étudiée ;
- toutes les interactions montrant les communications et collaborations entre les objets.

Par ailleurs, le diagramme de séquence mettant l'accent sur la chronologie des interactions entre objet, l'aspect temporel est pris en compte par la dimension verticale du diagramme, avec un écoulement du temps qui va du haut vers le bas, ce qui permet de préciser l'ordre des messages échangés ; les messages peuvent cependant être éventuellement numérotés.

En revanche, la position horizontale des objets les uns par rapport aux autres est sans importance [3].

[1] En réalité, les flèches décrivant les interactions peuvent posséder diverses têtes différentes ayant une signification particulière.
[2] cf. 3.4.5.
[3] C'est donc généralement le souci de lisibilité qui guide ce choix.

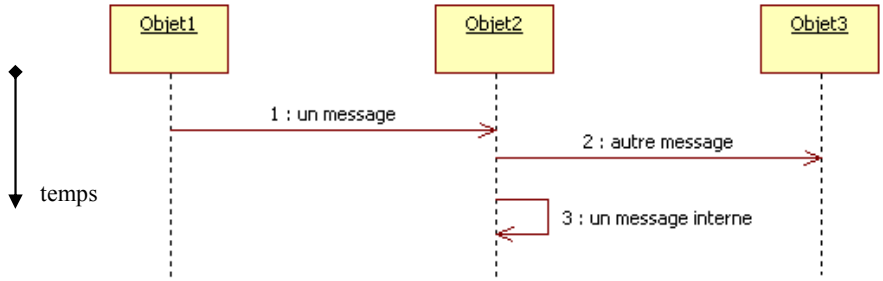

Figure 3.3 : représentation d'un diagramme de séquence

3.3 UTILISATIONS

3.3.1 Forme simplifiée : documentation d'un cas d'utilisation

La forme simplifiée des diagrammes de séquence permet de décrire graphiquement le cas d'utilisation, et ainsi de compléter la documentation de celui-ci. Généralement, le diagramme de séquence associé à un cas d'utilisation est construit sur la base de la description textuelle ou en pseudo-code de celui-ci, ou bien peut s'y substituer.

Le diagramme de séquence se concentre alors principalement sur la description temporelle des interactions du système avec les éléments extérieurs. Il s'agit de montrer la progression de la séquence dans le temps ; les messages échangés sont donc de type asynchrone, car les objets communiquent mais restent indépendants les uns des autres [1].

3.3.1.1 Description du flot d'évènements principal

Si la description textuelle ou en pseudo-code du cas d'utilisation a été faite, il suffit de retranscrire celle-ci graphiquement. On visualise ainsi la suite dans le temps des interactions entre le système et les acteurs.

Ex. : Pour le diagramme de séquence du cas d'utilisation décrivant l'action d'*acheter un billet de train* en utilisant une borne automatique, le flot d'évènements principal est le suivant : l'utilisateur sélectionne, grâce au clavier, sa destination, la classe du billet (première ou seconde), insère sa carte bleue, paie le montant demandé, puis enfin retire le billet qui est imprimé par la borne.

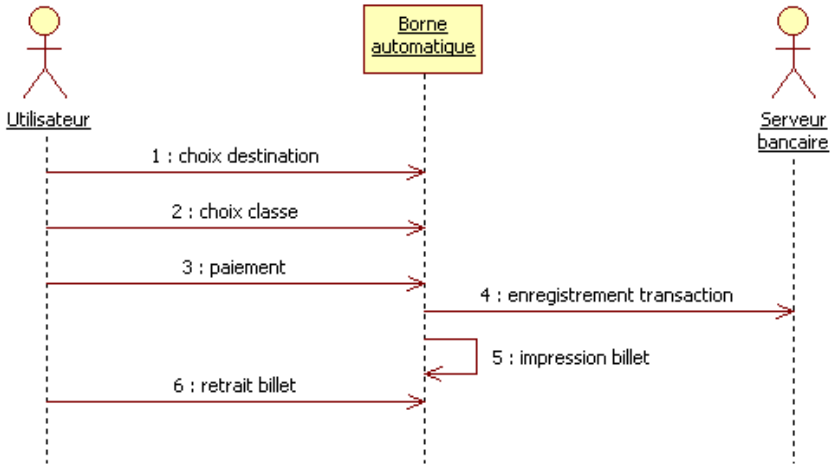

Figure 3.4 : exemple d'un diagramme de séquence simplifié

3.3.1.2 Description des flots d'évènements alternatifs et exceptionnels

Les diagrammes de séquence représentent la chronologie des interactions échangées dans le sens vertical ; il n'est donc pas toujours très simple de représenter les comportements non séquentiels (structures conditionnelles, répétitives, récursives ou concurrentes). Aussi, précise-t-on généralement ceux-ci – ainsi que tout comportement particulier en général – sous la forme de notes insérées en marge du diagramme de séquence, à la hauteur temporelle adéquate.

[1] On parle alors de *flot de contrôle à plat*.

Ex. : Pour le diagramme de séquence du cas d'utilisation décrivant l'action d'*acheter un billet de train* en utilisant une borne automatique, il y a un seul flot d'évènements exceptionnel qui est le suivant : l'utilisateur modifie la gare de départ, puis suit le fonctionnement principal (sélection de la destination, etc.).

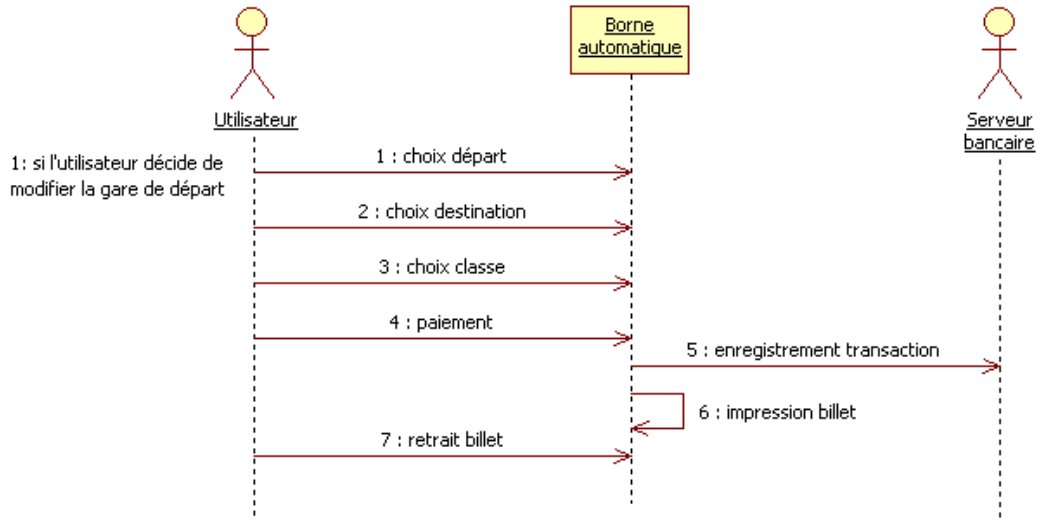

Figure 3.5 : exemple d'un diagramme de séquence simplifié annoté

3.3.2 Forme détaillée : éclatement du système en classes

Le système était vu jusqu'à présent comme une boîte noire. La forme détaillée des diagrammes de séquence permet de mettre en évidence les différentes composantes du système, tout comme elle permet de décrire plus précisément les interactions entre les objets.

Cette forme détaillée se déduit de la forme simplifiée en cherchant à se rapprocher du code objet, sachant que :

- Les différents acteurs seront représentés par des instances de classe ;
- Les différentes composantes du système seront représentées par des instances de classe ;
- Les interactions seront représentées par des messages échangés entre des instances de classes, c'est-à-dire des appels de méthode ;
- Les interactions internes sont représentées par des interactions entre diverses composantes du système, c'est-à-dire des appels de méthode à partir d'autres méthodes – la méthode appelante et la méthode appelée appartenant ou pas à la même classe.

Les acteurs et les composantes correspondent à des instances de classe, mais celles-ci n'ont pas été nommées. Si on ne décide pas d'un nom pour une instance, celle-ci est donc littéralement « anonyme » et on ne connaît que la classe à laquelle elle appartient ; la notation de l'objet sur le diagramme de séquence est donc modifiée de la sorte : `<objet anonyme> : NomClasse`, noté : `NomClasse`.

Si on décide de donner un nom à une instance, la notation de l'objet sur le diagramme de séquence est alors modifiée ainsi : `nomObjet : NomClasse`.

Les interactions correspondent généralement à des appels de méthodes (/appels de procédures) [1], soit donc des message synchrones [2]. Elles sont représentées par des flèches à têtes triangulaire [3] et leur notation sur le diagramme est modifiée de la sorte : `message()`, ce qui permet aussi de signifier les éventuels paramètres d'appel : `message(paramètres)`. L'objet émetteur du message est celui qui réalise l'appel de méthode de l'objet destinataire ; la méthode appartient donc à l'objet destinataire.

[1] Plus rarement, une interaction peut traduire un événement, un signal, une interruption matérielle, …
[2] On parle alors de *flot de contrôle emboîté*.
[3] cf. 3.4.5.

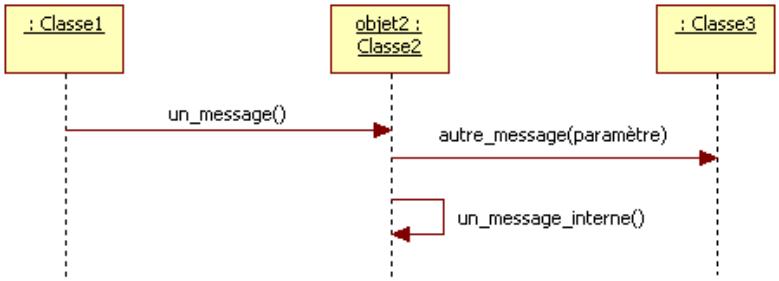

Figure 3.6 : représentation d'un diagramme de séquence détaillé

Dans la représentation ci-dessus, l'objet « objet2 » de la classe « Classe2 » effectue l'appel de la méthode autre_message( ) de l'objet anonyme de la classe « Classe3 » et lui passe un paramètre.

3.3.2.1 Recherche des classes

Décomposer le système revient donc à déterminer les différentes classes qui le composent. Pour cela, sur la base des diagrammes de séquence dans leur version simplifiée, on recherche nominativement les classes potentielles de manière intuitive, auxquelles on applique ensuite la technique CRC (Classe / Responsabilité / Collaboration). C'est-à-dire que pour chaque classe potentielle, il faut rechercher ses responsabilités et ses collaborations.

- responsabilité (parfois appelée aussi service) : description de haut niveau de la raison d'être d'une classe, qui décrit l'un des objectifs attendus de la classe au sein du système ;
- collaboration : interaction de la classe potentielle avec d'autres classes du système nécessaire à la réalisation d'une responsabilité.

Nom de la classe potentielle	
Responsabilité 1	Nom de la / des classe(s) de la collaboration / <rien>
…	…
Responsabilité N	

Au final, pour chaque classe analysée on doit avoir :

- une ou plusieurs responsabilité(s) bien définie(s), sans doublon avec les responsabilités d'une autre classe ;
- au moins une collaboration avec une autre classe pour l'une ou l'autre de ses responsabilités.

On doit donc retrouver l'ensemble des actions à mener par le système dans les responsabilités réalisées par l'une ou l'autre des classes.

Ex. : Dans le cas d'utilisation *acheter un billet de train*, on note un ensemble de classes potentielles à priori nécessaires au bon fonctionnement du système : Système, Clavier, Écran, Imprimante, Trajet.

Système	
Gérer le fonctionnement du système et le déroulement du processus	

Clavier	
Attendre les saisies de l'utilisateur	Utilisateur
Transmettre les données saisies	Système

Écran	
Afficher des messages publicitaires	
Recevoir les informations	Système
Afficher les informations	

Imprimante	
Recevoir les informations	Système
Imprimer le billet	

Trajet	
Stocker les données d'un trajet	Système

Autres classes potentielles : LecteurCarteBleue (lit les infos de la CB), etc.

3.3.2.2 Modification des diagrammes de séquence

Une fois les classes potentielles analysées et validées, il suffit de les insérer dans les diagrammes de séquence dans leur forme simplifiée, à la place de l'objet « système », et de compléter les messages nécessaires à la description du fonctionnement du cas d'utilisation. On obtient ainsi les diagrammes de séquence dans leur forme détaillée.

Ex. : Le diagramme de séquence du cas d'utilisation décrivant l'action d'*acheter un billet de train* en utilisant une borne automatique.

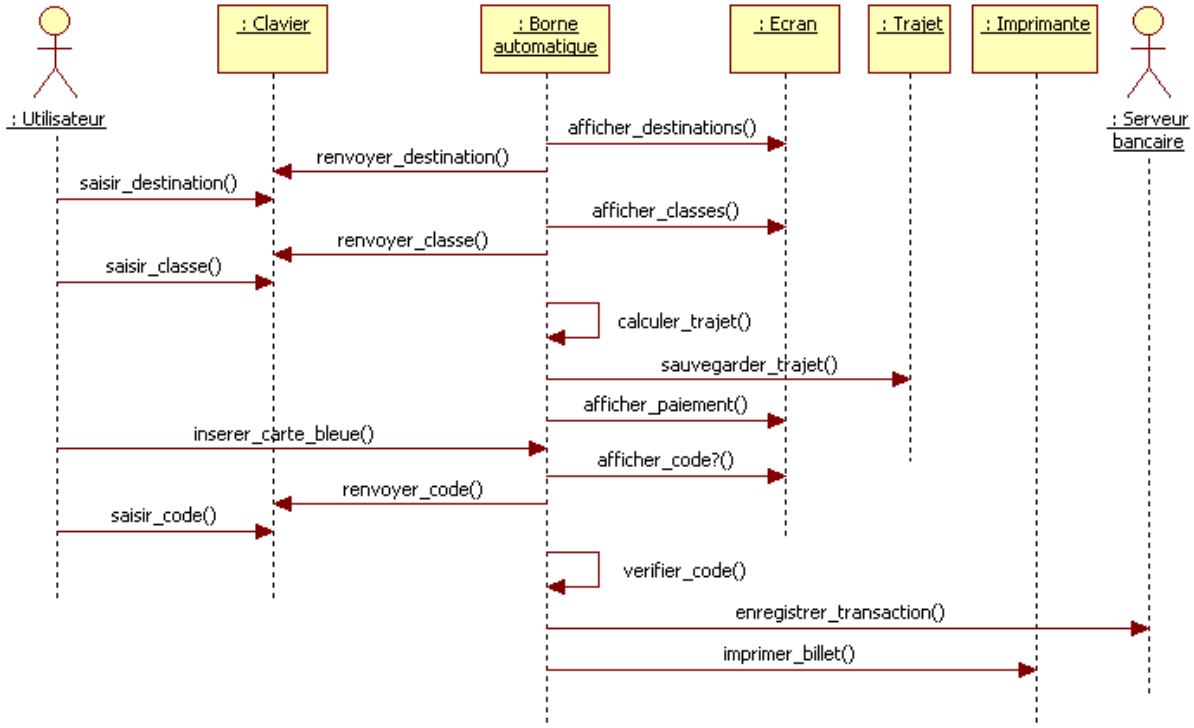

Figure 3.7 : exemple d'un diagramme de séquence détaillé

3.4 COMPLÉMENTS

3.4.1 Stéréotypes de Jacobson

Les **stéréotypes de Jacobson** [12] sont des catégories de classes qui permettent de préciser un rôle générique pour 1 ou plusieurs classes du système par rapport à son fonctionnement global.

On dispose ainsi de 3 stéréotypes différents :

- `<<boundary>>` (frontière) : classe faisant office d'IHM ;
- `<<control>>` (contrôle) : classe contrôlant le système (généralement une seule) ;
- `<<entity>>` (entité) : classe de stockage de données (généralement sans méthodes autre que méthodes accesseurs [3]).

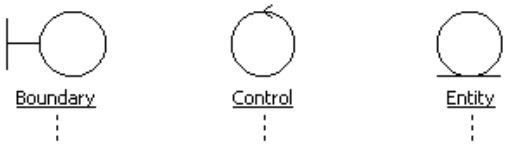

Figure 3.8 : représentation graphique des classes selon les stéréotypes de Jacobson

Ex. : Les classes du diagramme de séquence du cas d'utilisation décrivant l'action d'*acheter un billet de train* en utilisant une borne automatique.

[1] Des stéréotypes de classe autres que « de Jacobson » existent.
[2] L'un des fondateurs du langage UML.
[3] get( ) / set( ).

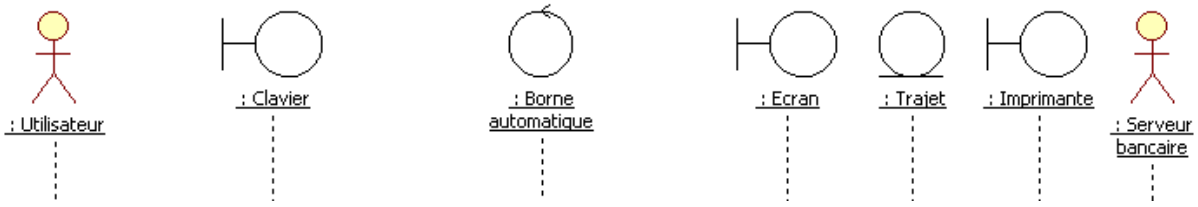

Figure 3.9 : exemple d'un diagramme de séquence en utilisant les stéréotypes de Jacobson

Les stéréotypes de Jacobson sont ainsi une bonne base de départ pour la recherche des classes potentielles ; en effet, dans tout système, il est rare de ne pas disposer d'une classe `<<control>>`, d'au moins 1 classe `<<boundary>>`, et d'une ou plusieurs classes `<<entity>>`.

3.4.2 Flot de contrôle

Le terme de *contrôle* détermine l'objet actif à l'instant t dans le déroulement du cas d'utilisation. Au départ, le contrôle est initié par un objet (généralement un acteur), lorsque celui-ci envoie un message à l'un des objets du système. Se faisant, il passe le contrôle à cet objet, qui lui-même passe le contrôle à un autre objet lorsqu'il envoie un message, etc.

Le terme de **flot de contrôle**, ou flot d'exécution, désigne le déroulement du cas d'utilisation du point de vue de la suite d'objets actifs au fur et à mesure des appels de méthode.

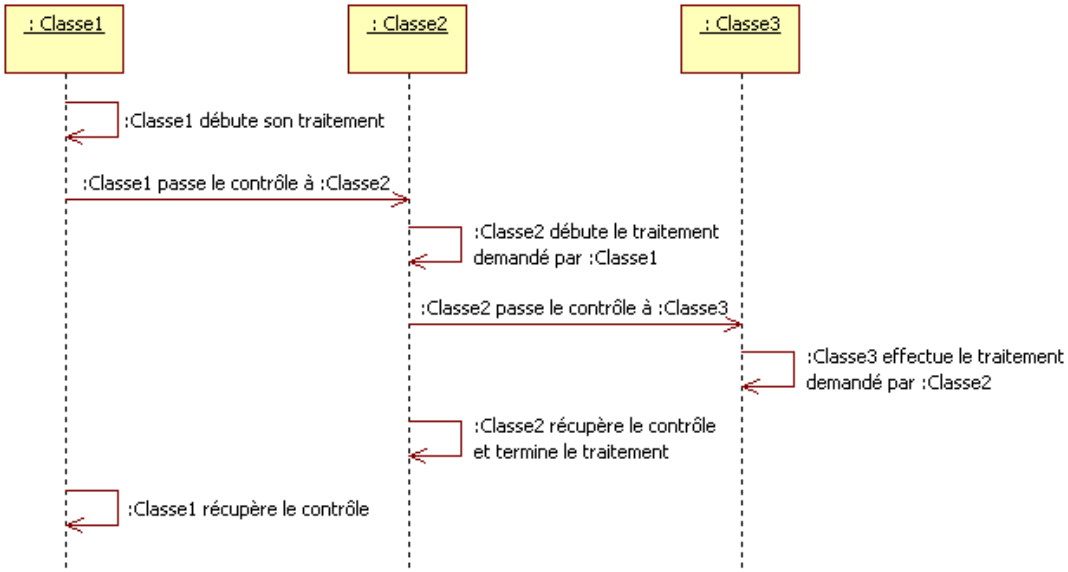

Figure 3.10 : flot de contrôle

3.4.3 Période d'activité

Une **période d'activité**, ou *durée d'activation*, d'un objet correspond à un temps durant lequel l'objet est en activité, c'est-à-dire une période pendant laquelle :

- L'objet effectue directement une action (il possède le contrôle) ;
- Une action est réalisée par un autre objet qu'il a lui-même mandaté (il a passé le contrôle), objet qui est donc également en activité.

L'activité d'un objet ne correspond pas à sa durée de vie [1], et est généralement déclenchée par la réception d'un message, lequel peut donc être qualifié de *stimulus d'activation*. Un même objet peut ainsi être en activité plusieurs fois pendant le déroulement du cas d'utilisation s'il reçoit plusieurs stimuli d'activation.

Une période d'activité est représentée par une bande rectangulaire sur la ligne de vie de l'objet en activité, positionnée précisément sur l'axe vertical et déterminant ainsi le début et la fin de la période d'activité.

[1] Même si un objet doit bien entendu exister pour pouvoir être en activité.

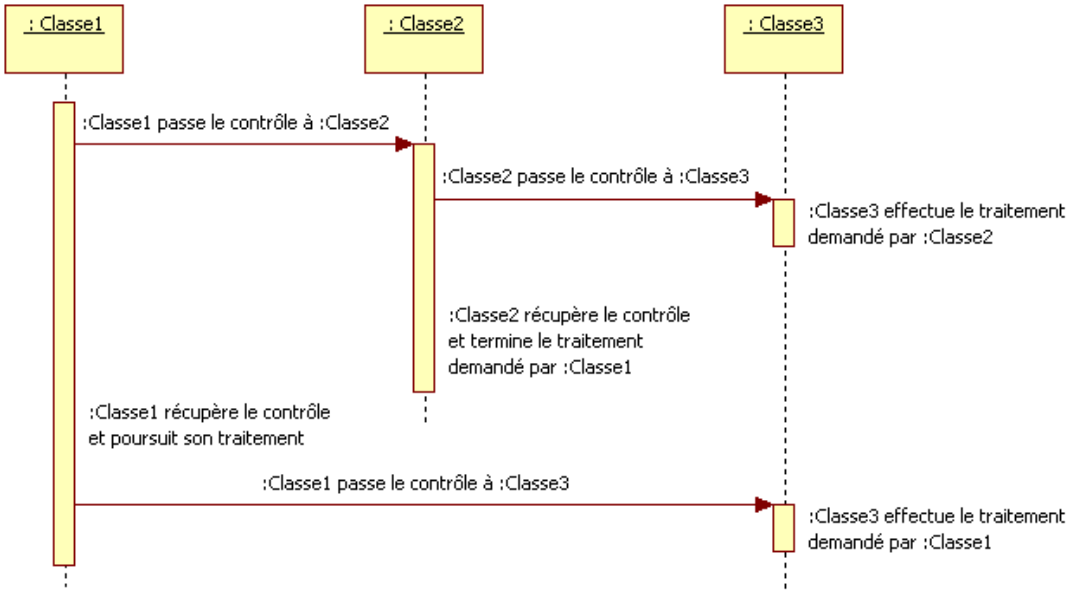

Figure 3.11 : représentation d'une période d'activité

3.4.4 Messages de création et de destruction d'instance

Les objets présentés sur un diagramme de séquence ont généralement une ligne de vie qui s'étend sur toute la hauteur diagramme. Néanmoins, les interactions peuvent désigner explicitement la création ou la destruction d'instance lors du déroulement du diagramme de séquence.

La **création** d'instance se représente par un message pointant sur le rectangle du nom de l'objet, et non sur sa ligne de vie, signifiant ainsi que le message crée l'objet. Éventuellement, un message de création d'instance peut être représenté comme un message précisé du stéréotype `<<create>>`, pointant sur la ligne de vie de l'objet.

La **destruction** d'instance se représente par un message à la suite duquel la ligne de vie est interrompue et marquée d'une croix ; le message peut être précisé par le prototype `<<destroy>>`.

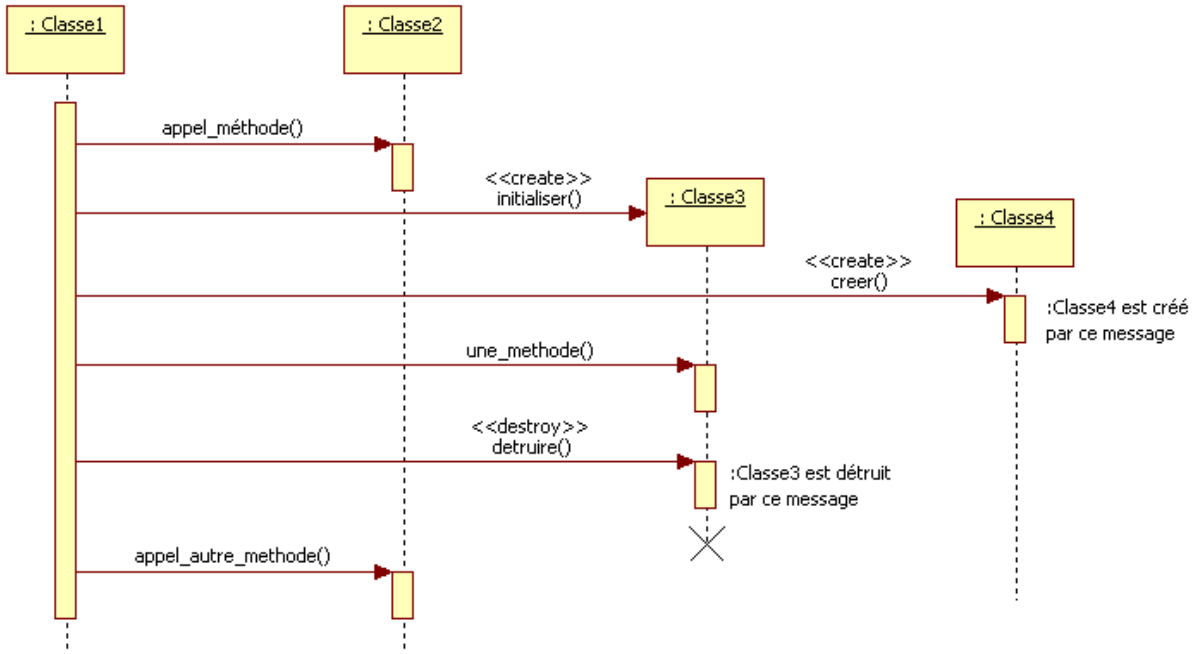

Figure 3.12 : représentation des messages de construction et de destruction d'instance

3.4.5 Messages synchrones et asynchrones

Les interactions peuvent être **synchrones** ou **asynchrones** en fonction du message qu'elles véhiculent :

- interaction synchrone (appel de méthode/procédure) : l'objet émetteur doit attendre la réalisation complète par l'objet destinataire de l'opération spécifiée par le message ; l'objet émetteur peut ainsi superviser la réalisation de l'opération ;
- interaction asynchrone (envoi de message) : l'objet émetteur n'est pas bloqué durant la réalisation par l'objet destinataire de l'opération spécifiée par le message ; de fait, l'objet émetteur n'a pas de moyen de superviser la réalisation de cette opération.

De plus, une interaction peut désigner spécifiquement la fin de la réalisation d'une opération en indiquant explicitement le retour d'un appel de méthode. Le retour de méthode/procédure est généralement implicite à la fin d'une période d'activité, mais il peut être utile de le mentionner explicitement pour apporter des informations sur le retour de l'appel de méthode (paramètre ou valeur de retour, par exemple), ou bien lors de l'utilisation de messages asynchrones afin de permettre la synchronisation entre objets.

Un message synchrone est représenté par une flèche à tête triangulaire et trait plein. Un message asynchrone est représenté par une flèche horizontale à tête classique et trait plein ; il peut être aussi éventuellement représenté par une demi-flèche [1]).

Un retour d'appel de procédure explicite est représenté par une flèche classique à trait pointillé.

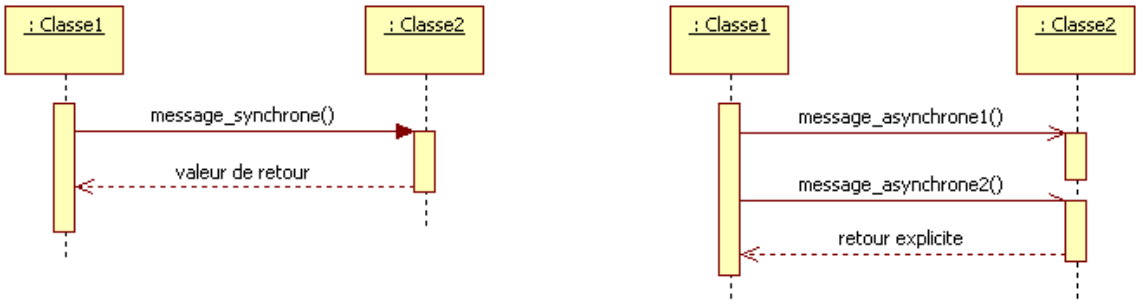

Figure 3.13 : représentation des messages synchrones et asynchrones

3.4.6 Représentation des comportements non séquentiels : tests et boucles

Les **comportements non séquentiels**, non linéaires dans le temps, peuvent être représentés par des annotations en marge du diagramme de séquence. On peut ainsi décrire facilement des structures conditionnelles (tests) et des structures itératives (boucles).

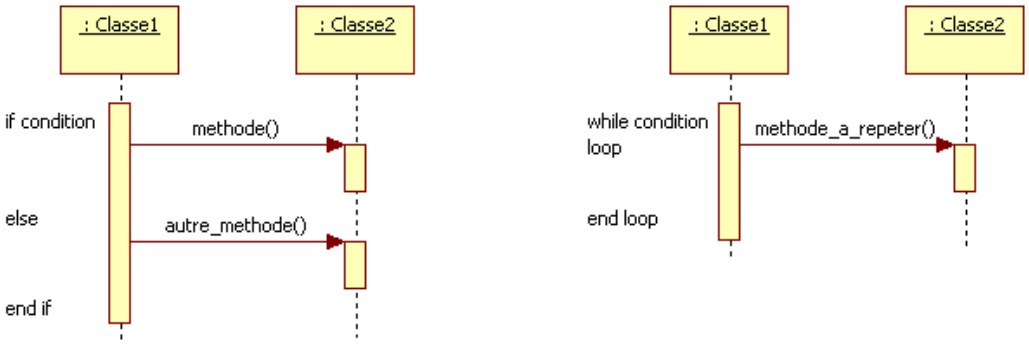

Figure 3.14 : représentation des comportements non séquentiels

Nb : UML v2 a introduit la notion de *cadres d'interaction* et de *fragments* qui permettent notamment de représenter de manière standard les structures conditionnelles et itératives.

[1] Avant UML v1.4.

3.4.7 Syntaxe complète des messages

Les interactions peuvent être précisées par des indicateurs associés aux messages.

La syntaxe complète est la suivante :
`précédents / [conditions] numéro *[répétition] : variable = nomMessage(paramètres).`

- `précédents` : liste des messages devant être envoyés avant celui-ci ;
- `[conditions]` : conditions nécessaires à l'envoi du message, ex : `[X = 1]`, `[t < 5s]` ;
- `numéro` : numéro du message parmi l'ensemble des messages ;
- `*[répétition]` : conditions de répétition du message, ex. : `*[i := 1..2]` ;
- `variable =` : récupération de la valeur de retour renvoyée par le message, ex. : `val = message()` (valeur de retour de la méthode) ;
- `paramètres` : valeur des paramètres avec lesquels le message doit être envoyé (valeur des paramètres d'entrée de la méthode).

3.5 COMPLÉMENTS

3.6 DIAGRAMMES COMPLÉMENTAIRES

3.6.1 Les diagrammes de collaboration

3.6.2 Les diagrammes d'activités

3.6.3 Les diagrammes d'états-transitions

Le rôle fondamental des diagrammes de séquence est donc de faire émerger les classes, ainsi que leurs méthodes.

4 LE DIAGRAMME DE CLASSES

4.1 DÉFINITION

Le **diagramme de classes** représente, de manière statique, les classes qui composent le système, ainsi que les relations existant entre elles.

Le but est de décrire :
- la structure statique interne précise de chacune des classes (attributs et méthodes) ;
- les relations entre les classes mises en œuvre.

Le diagramme de classes, qui est unique, se construit en partie à l'aide des informations issues des différents diagrammes de séquence. Il permet d'obtenir, à l'aide d'un outil logiciel approprié [1], le squelette du code par génération automatique de code [2] ; il s'agit donc de la dernière étape d'analyse juste avant le codage proprement dit. Par conséquent, il fait partie de la *conception détaillée* du système.

Une ébauche de diagramme de classes peut aussi être produite en phase de *spécification*, après l'analyse du cahier des charges et la production du diagramme de cas d'utilisation afin d'orienter les phases de *conception* ; on parle alors de *diagramme de classes d'analyse*. Cette ébauche est par la suite complétée lors de la phase de *conception détaillée*.

4.2 CONSTRUCTION DU DIAGRAMME

L'architecture statique d'un système fait état de :
- classes ;
- interfaces : classes non-instanciables [3] servant de modèle pour d'autres classes ;
- paquetages : regroupement de classes et de leurs relations.

Les relations pouvant exister entre les classes se départagent en plusieurs catégories :
- généralisation : héritage entre deux classes ;
- réalisation : héritage dans le cas d'une classe de base type interface ;
- association : utilisation d'une instance d'une classe à partir d'une instance d'une autre classe en tant qu'attribut-objet ;
- agrégation : association avec une notion d'appartenance et création de l'instance de l'attribut-objet ;
- composition : agrégation avec création + destruction de l'instance de l'attribut-objet ;
- classe d'association : écolution d'une association vers le concept de classe ;
- dépendance : instanciation ou utilisation d'une instance d'une classe par une autre n'étant pas déterminante pour leur structure interne.

4.2.1 Les classes

4.2.1.1 Définition

Une **classe** définit la description abstraite, servant de modèle, d'un ensemble d'objets ayant un *état* similaire, un *comportement* identique, mais une *identité* différente.

[1] AGL (StarUML, BoUML, Entreprise Architect, Objeteering, Rational Rose, …).
[2] Code engineering (eng).
[3] Évolution POO de la notion de classe abstraite.

Les étapes précédentes de l'analyse, et principalement celles réalisées lors de la conception préliminaire [1], guident pour la détermination des classes.

En effet, tous les objets mis en évidence précédemment (diagrammes de séquence) sont susceptibles d'être considérés comme des instances de classe ; on en déduit donc les classes. D'autres classes, notamment des classes de généralisation, peuvent être rajoutées si leur nécessité apparaît.

Les messages reçus par un objet sont eux susceptibles d'être considérés comme des méthodes, l'émetteur du message réalisant alors un appel de l'une des méthodes du destinataire ; la méthode exacte est celle correspondant au contenu du message.

La définition d'une classe comprend impérativement les 3 éléments suivants :

- nom de la classe : description du modèle d'objets ;
- attributs : informations qualifiant l'objet, variables associées à la classe ;
- méthodes : compétences, opérations, actions et réactions de l'objet, fonctions associées à la classe.

4.2.1.2 Représentation graphique

Une classe se représente par un cadre à plusieurs niveaux : nom, attributs et méthodes.

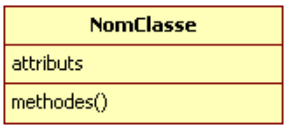

Figure 4.1 : représentation graphique d'une classe

Outre le nom de la classe, on peut préciser :

- l'indication `NomClasse : class` pour ôter tout doute ;
- classe abstraite : nom de la classe en italique.

Pour chaque membre sont précisées toutes les informations nécessaires à la définition exacte de l'architecture interne de la classe.

Pour un attribut, la syntaxe est :
`visibilité nomAttribut : modificateurs type multiplicité = valeur_initiale.`

- visibilité : + (public) / # (protégé) / – (privé) ;
- type : `boolean` / `short` / `int` / `char` / `float` / `double` (types primitifs) / autre classe (type complexe) ;
- multiplicité : `[]` (tableau) / `*` (pointeur) / `&` (référence) ;
- modificateurs : `const` (constant) / `static` ou nom de l'attribut souligné (statique).

Pour une méthode, la syntaxe est :
`visibilité nomMéthode(paramètres) : modificateurs type multiplicité.`

- visibilité : + (public) / # (protégé) / – (privé) ;
- type : `void` / `boolean` / `short` / `int` / `char` / `float` / `double` (types primitifs) / autre classe (type complexe) ;
- paramètres : type primitif ou complexe de chaque paramètre, séparés par une virgule, éventuellement avec leur nom ;
- multiplicité : `[]` (tableau) / `*` (pointeur) / `&` (référence) ;
- modificateurs : `const` (constant) / `static` ou nom de la méthode soulignée (statique) / `virtual` (virtuelle) / `abstract` ou nom de la méthode en italique (abstraite / virtuelle pure).

Nb : Lorsqu'un attribut correspond à une instance d'une classe spécifique au système, généralement on ne l'intègre pas dans la liste des attributs, mais on utilise une association, afin de mettre en avant une relation avec une autre classe.

Ex. : La classe `BorneAutomatique` d'un système de borne automatique permettant d'acheter un billet de train.

[1] Les diagrammes de séquence détaillés.

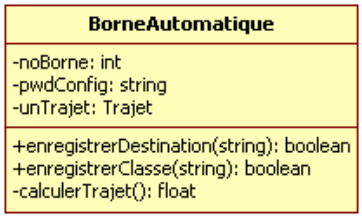

Figure 4.2 : exemple de représentation graphique d'une classe

Le nom de la classe peut être éventuellement précédé par la précision d'un stéréotype (`<<class>>`, `<<interface>>`, `<<entity>>`, `<<control>>`, etc.).

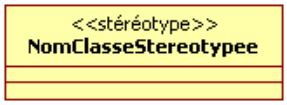

Figure 4.3 : représentation graphique d'une classe stéréotypée

4.2.2 Les interfaces

4.2.2.1 Définition

Une **interface** est un type de classe spécifique, qui peut être définie comme étant une classe abstraite dont toutes les méthodes sont abstraites [1] et qui ne possède aucun attribut [2]. Une interface représente ainsi une classe-« concept ».

On peut assimiler une interface à une classe de services, c'est-à-dire une classe qui est utilisée par d'autres classes pour leur donner un cadre d'implémentation des fonctionnalités (/services) qu'elles doivent proposer.
En d'autres termes, une interface ne fait que représenter, sans l'implémenter, un comportement (quoi) que doivent suivre certaines classes, laissant ainsi la liberté à chaque classe de l'implémenter selon ses besoins (comment).

Nb : Le principe d'interface n'est pas implémenté dans tous les langages orientés objet : il est par exemple présent en Java et en C#, mais absent en C++ [3].

4.2.2.2 Représentation graphique

Une interface peut se représenter de deux manières :

- comme une classe, excepté qu'on précise le stéréotype `<<interface>>`, ce qui permet ainsi de donner le détail des attributs et méthodes ;
- cercle nommé.

Figure 4.4 : représentation graphique d'une interface

4.2.3 Les paquetages

4.2.3.1 Définition

Un **paquetage** désigne un regroupement de classes ou interfaces, ainsi que leurs relations. L'idée est de rassembler des classes ayant le même type d'utilité, appartenant à une même thématique ou qui collaborent dans un même but ; on rassemblera par exemple une classe de base avec ses classes dérivées – surtout si la classe de base est abstraite.

4.2.3.2 Représentation graphique

Un paquetage est représenté par un cadre avec « onglet » entourant les classes qui en font partie. Le nom peut être précisé dans le haut du cadre ou bien dans l'onglet.

[1] Abstraite ≡ virtuelle pure.
[2] Dans certains langages, le concept d'interface a quelque peu dérivé et peut posséder des attributs. Ex : en Java, une interface a des attributs obligatoirement constants.
[3] Rien n'empêche cependant de définir une classe abstraite dont toutes les méthodes sont virtuelles pures.

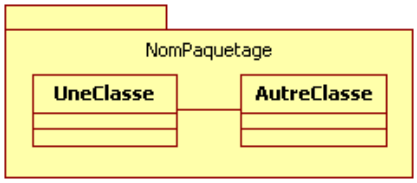

Figure 4.5 : représentation graphique d'un paquetage

4.2.4 Les généralisations et les spécialisations

4.2.4.1 Définition

La **généralisation** dans un ensemble de classes désigne la mise en commun d'attributs et de méthodes au sein d'une classe-mère ; chaque classe possède les attributs et méthodes de sa classe-mère auxquels viennent s'ajouter les membres propres à la classe.

Inversement, la **spécialisation** désigne la propagation des attributs et des méthodes d'une classe dans une classe-fille, laquelle peut intégrer des membres supplémentaires.

La généralisation/spécialisation est mise en œuvre par le mécanisme d'héritage (/dérivation) de la POO.

4.2.4.2 Représentation graphique

La généralisation d'une classe vers une autre se représente par une flèche à tête triangulaire et trait plein reliant les 2 classes dans le sens *classe spécialisée → classe générale*.

Figure 4.6 : représentation graphique d'une généralisation

4.2.5 Les réalisations

4.2.5.1 Définition

La **réalisation** dans un ensemble de classes correspond au principe de généralisation pour laquelle la classe généralisée (la super-classe) est une interface. On dit alors que la classe spécialisée *implémente* l'interface.

4.2.5.2 Représentation graphique

La réalisation d'une interface par une classe peut se représenter de deux manières :

- cas d'une interface représentée comme une classe stéréotypée : par une flèche à tête triangulaire et trait pointillé reliant les 2 entités dans le sens *classe réalisatrice → interface* ;
- cas d'une interface représentée selon un cercle nommé : par un simple trait plein.

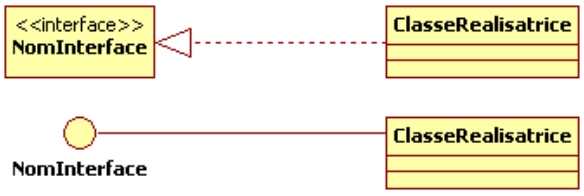

Figure 4.7 : représentation graphique d'une réalisation

4.2.6 Les associations simples

4.2.6.1 Définition

Une **association** désigne le cas où une instance d'une classe utilise un objet d'une autre classe en tant qu'attribut-objet. Cependant, il n'est pas de la responsabilité de la classe utilisatrice de créer ou de détruire l'instance utilisée ; elle doit se servir d'un objet instancié par une autre classe afin d'initialiser son attribut-objet.

Le couplage entre les classes est faible, et les deux classes restent relativement peu dépendantes l'une de l'autre.

4.2.6.2 Représentation graphique

Une association se représente par une flèche à tête classique (non-triangulaire) et trait plein, orientée dans le sens de la navigabilité *classe utilisatrice → classe utilisée*.

L'association peut être précisée par diverses informations textuelles :

- le nom : désigne l'utilité de l'association, précisée éventuellement du sens de lecture ('<' / '>') ;
- le rôle de chacune des classes : décrit comment chaque classe est vue par l'autre à travers l'association ;
- la cardinalité de chacune des classes : précise le nombre d'instances impliquées dans l'association :
  - 1 ou rien : exactement 1 ;
  - X : exactement X ;
  - 0..1 : 0 ou 1 ;
  - 0..* ou * : plusieurs ;
  - X..Y : X à Y ;
  - X..* : X à plusieurs ;
  - X..Y, U..V : X à Y plus U à V.

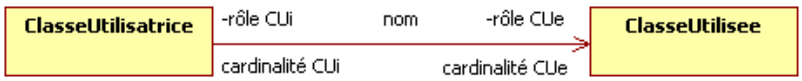

Figure 4.8 : représentation graphique d'une association

Ex. : La borne automatique utilise le serveur bancaire. La classe `BorneAutomatique` du système utilise un objet de la classe `ServeurBancaire` (`BorneAutomatique` appelle la méthode `enregistrer_transaction()` de `ServeurBancaire` ; `BorneAutomatique` utilise donc une instance de `ServeurBancaire`).

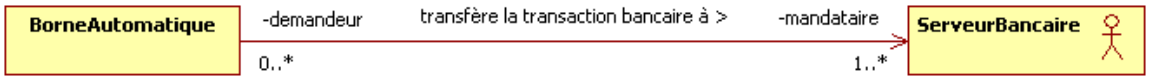

Figure 4.9 : exemple d'association

Dans le cadre de l'association « `BorneAutomatique` transfère la transaction bancaire à `ServeurBancaire` » :

Un serveur bancaire est susceptible d'être contacté par plusieurs bornes automatiques différentes ; en revanche, on peut imaginer que certains serveurs bancaires ne seront jamais contactés par une borne automatique (0..*).

Une borne automatique fait toujours appel à 1 serveur bancaire pour finaliser la transaction ; la complexité des partenariats inter-bancaires induit que la borne peut nécessiter de faire appel à plusieurs serveurs bancaires (1..*).

Nb : Le rôle est généralement utilisé comme nom d'instance lorsque l'association est implémentée dans un programme en langage orienté objet. De fait, le rôle peut éventuellement être précisé par un indicateur de visibilité (ici '–' devant `demandeur` et `mandataire` précisant une visibilité privée).

On peut mentionner les cas particuliers suivants :

- association bidirectionnelle : chacune des classes utilise une instance de l'autre classe ; l'association est alors représentée non-fléchée, indiquant qu'elle est navigable dans les 2 sens ;
- association réflexive : la classe utilise une instance d'elle-même ; l'association est alors représentée par un lien qui revient sur la même classe ;
- association multiple : les classes ont plusieurs relations distinctes entre elles ; auquel cas, on prend alors soin de préciser le nom de chaque association et pour chacune d'entre elles le rôle de chaque classe, afin de distinguer l'utilité des différentes relations ;
- association n-aire : l'association relie plus de 2 classes distinctes ; l'association est représentée par une étoile à n branches dont chacune d'entre elles est reliée à l'une des classes de l'association, et dont le centre est un losange ; les associations n-aires peuvent être très facilement transformées en classes d'association [1].

[1] cf. 4.2.9.

4.2.7 Les agrégations

4.2.7.1 Définition

Une **agrégation** désigne un type particulier d'association à laquelle s'ajoute un concept d'appartenance ; l'objet utilisé fait partie de la classe utilisatrice, et est indispensable à sa création complète. La classe utilisatrice est donc responsable de la création de l'instance utilisée ; en revanche, elle n'est pas responsable de sa destruction, même si cela est fortement conseillée. Malgré tout, l'instance reste partageable entre plusieurs instances de la classe utilisatrice, tout comme des instances d'autres classes, et celles-ci peuvent donc l'utiliser.

Dans le cadre de cette agrégation, la classe utilisatrice est alors appelée *agrégat* et l'objet utilisé *agrégé*.

Le couplage entre les deux classes est donc supérieur à celui d'une simple association ; il a néanmoins ses limites car les instances de l'agrégat et de l'agrégé ont chacune une durée de vie propre : l'agrégé est créé en même temps que l'agrégat, mais il peut continuer d'exister même si l'agrégat est détruit, surtout s'il est utilisé par d'autres instances.

Nb : L'agrégation peut aussi être appelée *agrégation par référence*, dû à la manière dont on l'implémente généralement en langage orienté objet.

4.2.7.2 Représentation graphique

Une agrégation se représente par une association orientée dans le sens *classe agrégat → classe agrégée* possédant un losange vide du côté de l'agrégat.

Là aussi, l'agrégation peut être précisée par le nom, les rôles et les cardinalités.

Figure 4.10 : représentation graphique d'une agrégation

Ex. : La borne automatique est faite d'un écran, elle est incomplète si elle n'en dispose pas d'un ; si la borne automatique est démontée, l'écran peut être ré-utilisé. La classe `BorneAutomatique` du système instancie un objet de la classe `Ecran` (`BorneAutomatique` possède une instance de `Ecran` parmi ses attributs).

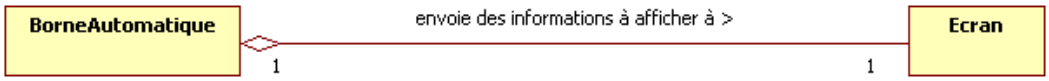

Figure 4.11 : exemple d'agrégation

Dans le cadre de l'association « `BorneAutomatique` envoie des informations à afficher à `Ecran` » :

Un écran affiche des informations qui lui sont nécessairement envoyées par une borne automatique ; les informations affichées ne peuvent cependant jamais provenir que d'une seule et même borne automatique (1).

Une borne automatique fait toujours usage d'un écran afin d'afficher les différents messages d'informations ; en revanche un seul écran suffit (1).

On peut mentionner les cas particuliers suivants :

- agrégation navigable : en plus des propriétés de l'agrégation, l'agrégé possède parmi ses attributs un objet de la classe de l'agrégat et peut l'utiliser (comme s'il existait une association orientée dans le sens agrégé → agrégat) ;
- agrégation réflexive : la classe possède une instance d'elle-même ; l'agrégation est alors représentée par un lien d'agrégation pointant sur la même classe.

4.2.8 Les compositions

4.2.8.1 Définition

Une **composition** désigne un type particulier d'agrégation à laquelle s'ajoute un concept de « partie d'un tout ». L'objet agrégé fait partie intégrante de la classe agrégat, est indispensable à sa création complète, et n'existe que parce qu'il est impérativement nécessaire à l'agrégat. L'agrégat est donc responsable de la création de l'agrégé, mais aussi de sa destruction. De fait, l'instance ne peut pas être partagée entre plusieurs instances, qu'il s'agisse d'instances de l'agrégat ou bien d'instances d'autres classes, et une seule peut donc l'utiliser.

Dans le cadre de cette composition, l'agrégat est alors appelé *composite* ou *conteneur* et l'agrégé *composant* ou *composé*.

Le couplage entre les deux classes est très important, et encore supérieur à celui d'une agrégation ; les instances du composite et du composant ont une durée de vie commune : le composant est créé et détruit en même temps que le composite, renforçant le principe que le composant n'est pas partageable.

Nb : La composition peut aussi être appelée *agrégation par valeur*, dû à la manière dont on l'implémente généralement en langage orienté objet.

4.2.8.2 Représentation graphique

Une composition se représente par une agrégation orientée dans le sens *classe composite → classe composant* possédant un losange plein du côté du composite.
La composition peut être bien évidemment précisée par le nom, les rôles et les cardinalités.

Figure 4.12 : représentation graphique d'une composition

Nb : Comme le composant n'est pas partageable entre plusieurs instances du composite, la cardinalité du côté de celui-ci ne peut donc prendre que la valeur 0 ou 1 (cardinalités : 1 / 0..1).

Une autre représentation possible de la composition insère le composant dans le composite sous forme de classe imbriquée. En ce cas, non seulement l'instance n'est pas partageable entre plusieurs instances (classe composite ou autre classe), mais de plus la classe composant ne peut être utilisée par aucune autre classe.

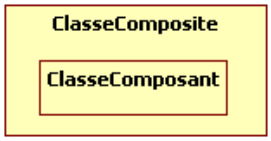

Figure 4.13 : représentation graphique d'une composition sous forme de classe imbriquée

Ex. : La borne automatique nécessite la notion de trajet pour fonctionner ; si la borne automatique est mise hors-service ou démontée, les trajets sont détruits. La classe `BorneAutomatique` du système instancie un objet de la classe `Trajet` (`BorneAutomatique` possède une instance de `Trajet` parmi ses attributs).

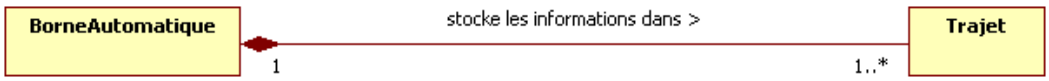

Figure 4.14 : exemple de composition

Dans le cadre de l'association « `BorneAutomatique` stocke les informations dans `Trajet` » :

Un trajet ne peut être associé qu'à une et une seule borne automatique (1).

Une borne automatique nécessite toujours un trajet afin de stocker les informations ; en revanche, elle peut avoir à gérer plusieurs trajets en même temps (1..*).

On peut mentionner les cas particuliers suivants :

- composition navigable : en plus des propriétés de la composition, le composant possède parmi ses attributs un objet de la classe du composite et peut l'utiliser (comme s'il existait une association orientée dans le sens composite → composant) ;
- composition réflexive : la classe possède une instance d'elle-même ; la composition est alors représentée par un lien de composition pointant sur la même classe.

4.2.9 Les classes d'association

4.2.9.1 Définition

Une **classe d'association** désigne l'évolution d'une association vers le concept de classe. Il s'agit donc de la description d'une relation dans laquelle une instance d'une classe utilise un objet d'une autre classe en tant qu'attribut-objet, relation qui a été transformée en une modélisation abstraite de famille d'objets, rassemblant ainsi des variables et des fonctions dédiées à la classe [1]. Parmi ses attributs, la classe d'association possède un attribut-objet pour chacune des 2 classes de l'association ; généralement, elle propose aussi des méthodes permettant d'accéder à ses attributs-objets de l'extérieur de la classe.

Le couplage entre les classes est variable puisqu'une classe d'association peut être constuite sur la base d'une association simple, d'une agrégation ou d'une composition.

L'association décrite par la classe d'association est généralement bidirectionnelle, mais peut éventuellement n'être navigable que dans un seul sens.

Nb : La classe d'association peut aussi être appelée *classe-association* ou *classe associative*. Lorsque la classe d'association n'est connectée qu'à l'association entre les 2 classes et n'a de relation avec aucune autre classe, on parle alors d'*association attribuée*.

4.2.9.2 Représentation graphique

Une classe d'association se représente en mélangeant les représentations d'une association simple classique et d'une classe classique. Les deux classes de l'association sont reliées entre elles, généralement par une association bidirectionnelle ; la classe d'association est raccordée à l'association par un trait pointillé connecté au trait plein.
L'association peut être précisée par le nom, les rôles et les cardinalités ; la classe d'association peut être précisée par des attributs autres que les références sur chacune des classes associées, et des méthodes autres que les accesseurs sur les références des classes associées.

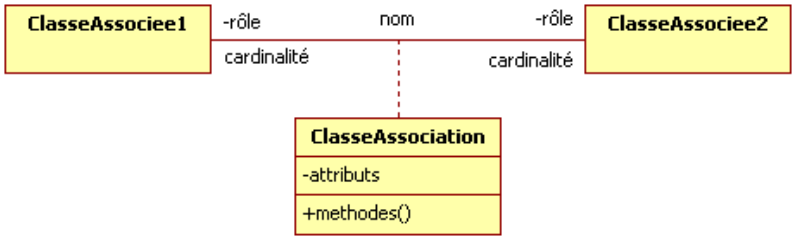

Figure 4.15 : représentation graphique d'une classe d'association

4.2.10 Les dépendances

4.2.10.1 Définition

Une **dépendance** désigne le cas où une entité utilise de manière quelconque une autre entité, sachant que ces entités peuvent être indifféremment classe, interface, paquetage, instance, méthode, attribut, … Il s'agit d'un concept très général pouvant signifier différents types de relations entre deux entités qui ne déterminent aucunement ni ne donnent d'information sur l'architecture interne des entités.

Pour préciser le type de dépendance, on dispose de différents types de stéréotypes normalisés regroupés en 4 types de relations, pouvant représenter la description de (*avec une dépendance de A → B…*) :

- **utilisation** (stéréotype `<<use>>`) :
  - l'instanciation d'un objet de la classe B dans une méthode de la classe A autre que le constructeur de A (stéréotype `<<instanciate>>`) ;
  - l'utilisation d'une référence sur une instance de la classe B comme paramètre d'entrée d'une méthode de la classe A (stéréotype `<<create>>`) ;
  - l'utilisation par une méthode de la classe A d'une méthode statique de la classe B (stéréotype `<<call>>`) ;
  - l'envoi d'un signal B par une méthode de la classe A (stéréotype `<<send>>`).

[1] Exemple d'application en modélisation de la maxime « le tout vaut plus que la somme des parties ».

- **abstraction** :
  - la définition d'une entité A à partir d'une entité B (stéréotype `<<derive>>`) ;
  - l'implémentation de l'entité B par une entité A (stéréotype `<<realize>>`) ;
  - la spécialisation d'une entité B en une entité A lors du passage de l'analyse à la conception (stéréotype `<<refine>>`) ;
  - le suivi des modifications d'une entité B ayant été remaniée en une entité A entre deux versions d'une même modélisation (stéréotype `<<trace>>`).
- **liaison** (stéréotype `<<bind>>`) :
  - l'« instanciation » d'une classe générique B avec définition complète des types des paramètres pour créer une classe A.
- **permission** (stéréotype `<<access>>`) :
  - la déclaration de l'entité A comme amie de l'entité B, permettant ainsi à A d'accéder à B quelque soit sa visibilité (stéréotype `<<friend>>`).

4.2.10.2 Représentation graphique

Une dépendance se représente par une flèche à tête classique (non-triangulaire) et trait pointillé orientée dans le sens *classe dépendante → classe utilisée*, qui est précisée par un stéréotype, parmi les stéréotypes normalisés ou bien des stéréotypes dédiés.

L'indication du stéréotype n'est pas obligatoire mais est fortement conseillée pour renseigner sur le type de dépendance.

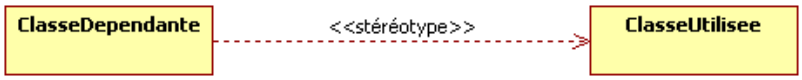

Figure 4.16 : représentation graphique d'une dépendance

4.2.11 Le diagramme

4.2.11.1 Définition

Au final, le diagramme de classes doit faire apparaître :

- toutes les classes et interfaces composant le système ;
- l'architecture interne complète de chacune des classes ;
- toutes les relations déterminées entre toutes les différentes classes ;
- les associations « transformées » après étude en agrégation ou composition ;
- les regroupements éventuels de classes, interfaces et relations en paquetages.

Ce diagramme est unique ; cependant, pour des commodités de lecture il peut être scindé en plusieurs parties, en paquetages par exemple, puis faire état des relations entre paquetages ; on peut aussi mentionner toutes les classes mais sans leur architecture interne qui est détaillée à part, etc.

4.2.11.2 Représentation graphique

Ce diagramme est la synthèse graphique de tous les éléments qu'il doit représenter.

Pour s'assurer de sa cohérence, plusieurs règles doivent être observées :

- Chaque classe ne doit être représentée qu'une seule fois ;
- Une classe a obligatoirement une relation avec au moins une autre classe ;
- On ne doit pas avoir plusieurs groupes de classes indépendants, à moins que l'on considère que l'application soit constituée de plusieurs processus différents s'exécutant de manière concurrente (environnement multi-tâches).

Ex. : Diagramme de classes de la borne automatique.

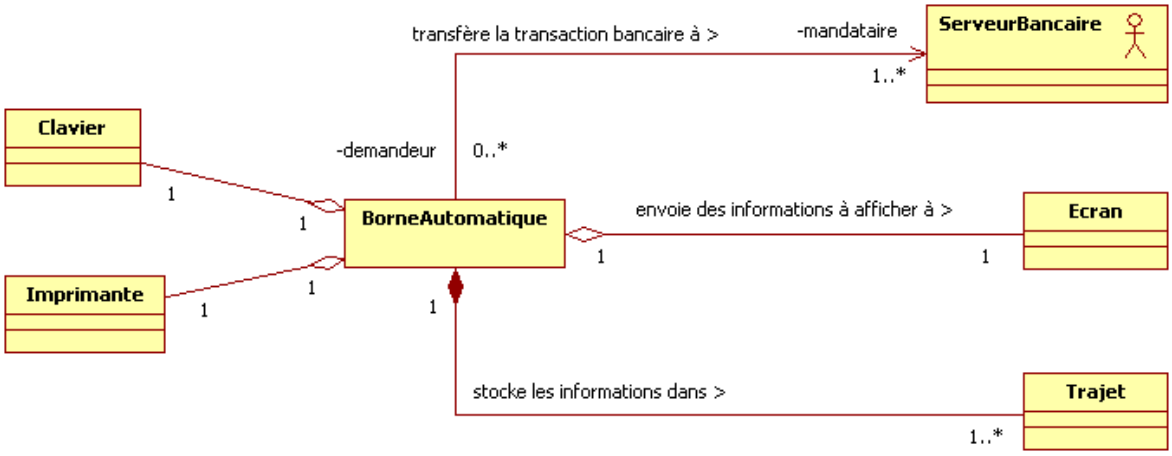

Figure 4.17 : exemple d'un diagramme de classes

4.3 DIAGRAMMES COMPLÉMENTAIRES

4.3.1 Les diagrammes d'objet

4.3.2 Les diagrammes de composants

4.3.3 Les diagrammes de déploiement

A ORIGINES DU LANGAGE

A.1 HISTORIQUE

1991 : Développement de OMT-1 (Objet Modeling Technique) par James Rumbaugh, aidé de M. Blaha, W. Premerlani, F. Eddy et W. Lorensen.
Développement de Booch'91 par Grady Booch.
1992 : Développement de OOSE (Objet Oriented Software Engineering) par Ivar Jacobson à Objectory AB.
1994 : Évolution de OMT-1 vers OMT-2.
Évolution de Booch'91 vers Booch'93.
1994 : Début de la collaboration entre Rumbaugh, Jacobson, et Booch au sein de la société Rational Software.
1995 : Travail collaboratif entre Booch, Jacobson et Rumbaugh amenant à la description publique d'une *méthode unifiée* dans sa version 0.8.
1996 : Baptême de la méthode unifiée sous le nom d'*UML* (Unified Modeling Language) dans sa version 0.9.
1997 : Publication par Rational de UML v1.0, version proposée à l'OMG pour standardisation.
Standardisation par l'OMG de UML v1.1.
1999 : Engouement exponentiel et consensuel autour d'UML dans sa version 1.3.
2001 : Publication de UML v1.4.
2003 : Publication de UML v1.5.
2005 : Publication de UML v2.0.
Publication de UML v2.1.
2009 : Publication de UML v2.2.

A.2 CONTRIBUTIONS

Nonobstant les 3 méthodes principales Booch, OMT et OOSE, un certain nombre de méthodes *objet* ont participé à l'élaboration d'UML.

OOSE (Jacobson, Ivar)	Cas d'utilisation, stéréotype de classes.
OMT (Rumbaugh, James)	Classes, instances et objets, attributs et méthodes, associations.
Booch (Booch, Grady)	Classes, objets, attributs et méthodes, sous-systèmes.
Meyer, Bertrand	Théorie du contrat, pré- et post-conditions.
Harel, David	Diagrammes états-transitions.
Wirfs-Brock, Rebecca	Responsabilités d'une classe.
Beck, Kent / Cunningham, Ward	CRC (Classe, Responsabilité, Collaboration).
Fusion (EROOS : Steegmans, Eric / Boydens Jeroen / Delanote Geert)	Description des méthodes, numérotation des messages dans les diagrammes d'interactions.
Gamma, Erich / Helm, Richard / Johnson Ralph / Vlissides, John M.	Patrons de design, frameworks, notes.
Shlaer, Sally / Mellor, Stephen J.	Cycle de vie des objets dans les diagrammes de séquence.
Odell, James J.	Classification dynamique, évènements.
Embley, David W.	Objets composites et classes singletons [1].

[1] Une classe « singleton » ne peut offrir qu'une seule instance en même temps ; autrement dit, toute nouvelle instanciation renvoie vers l'instance existante.

B LEXIQUE GRAPHIQUE

B.1 STRUCTURE STATIQUE

B.1.1 Classes

B.1.1.1 Classe

Cadre à 3 encarts :

- 1er encart : nom de la classe ;
- 2nd encart : liste des attributs ;
- 3ème encart : liste des méthodes.

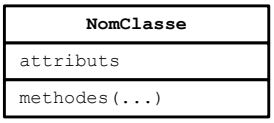

Figure B.1 : représentation UML d'une classe

Représentation simplifiée (détails des attributs et des méthodes omis) :

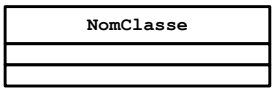

Figure B.2 : représentation UML simplifiée d'une classe (1)

Représentation simplifiée (encarts « attributs » et « méthodes » omis) :

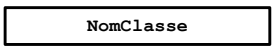

Figure B.3 : représentation UML simplifiée d'une classe (2)

B.1.1.2 Classe abstraite

`NomClasseItalique` : classe abstraite (`abstract`).

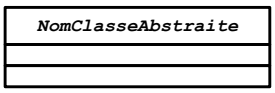

Figure B.4 : représentation UML d'une classe abstraite

B.1.1.3 Classe stéréotypée

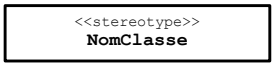

Figure B.5 : représentation UML d'une classe stéréotypée

B.1.1.4 Classe paramétrable

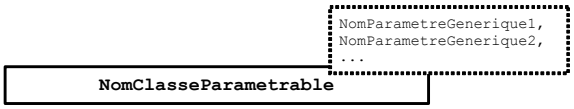

Figure B.6 : représentation UML d'une classe paramétrable

B.1.1.5 Classe paramétrée

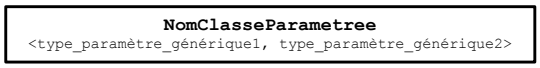

Figure B.7 : représentation UML d'une classe paramétrée

B.1.1.6 Interface

Figure B.8 : représentation UML d'une interface

B.1.2 Objets

B.1.2.1 Objet

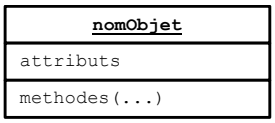

Figure B.9 : représentation UML d'un objet

Représentation simplifiée (encarts « attributs » et « méthodes » omis) :

Figure B.10 : représentation UML simplifiée d'un objet

B.1.2.2 Objet comme instance de classe

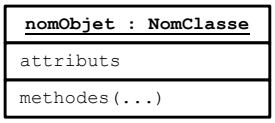

Figure B.11 : représentation UML d'un objet comme instance de classe

B.1.2.3 Objet anonyme

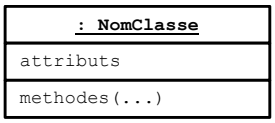

Figure B.12 : représentation UML d'un objet anonyme

B.1.3 Paquetage

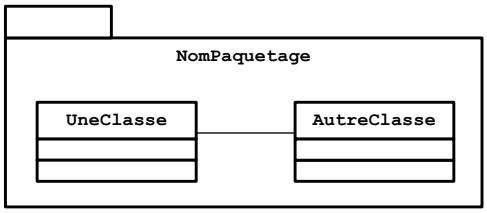

Figure B.13 : représentation UML d'un paquetage

B.1.4 Membres

B.1.4.1 Membres

`nomAttribut` : déclaration d'une variable associée à la classe ;
`nomMethode(...)` : déclaration d'une fonction associée à la classe.

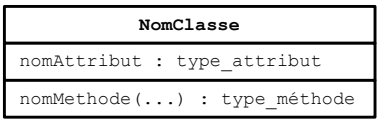

Figure B.14 : représentation UML d'une classe et de ses membres

Attribut de type *instance de classe* / *objet* : cf. associations (B.2).

B.1.4.2 Visibilité

− : privé (private) ;
: protégé (protected) ;
+ : public (public).

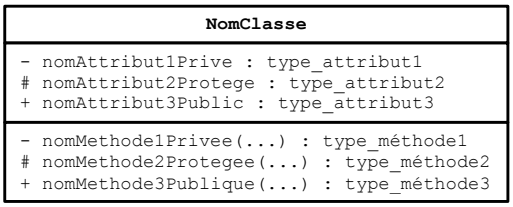

Figure B.15 : représentation UML d'une classe, de ses membres et de leur visibilité

B.1.4.3 Membre statique

`nomMembreSouligné` : membre statique / de classe (`static`).

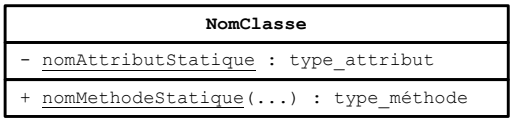

Figure B.16 : représentation UML d'une classe et des membres statiques

B.1.4.4 Méthode abstraite

`nomMethodeItalique` : abstraite / virtuelle pure (`virtual`).

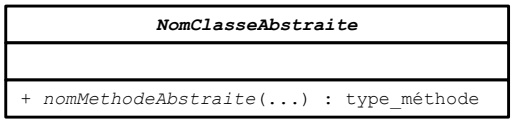

Figure B.17 : représentation UML d'une classe abstraite et d'une méthode abstraite

B.2 ASSOCIATIONS

Utilisation d'une instance d'une classe par une autre classe en tant qu'attribut-objet.
Plusieurs cas possibles : association simple, agrégation et composition.

B.2.1 Association « simple »

Utilisation d'une instance d'une classe par une autre classe (*a besoin de* / *nécessite*) en tant qu'attribut-objet.
Aucune responsabilité vis-à-vis de la création de l'instance ou de la destruction de l'instance.

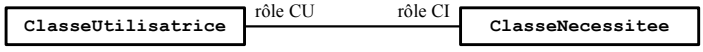

Figure B.18 : représentation UML d'une association

B.2.1.1 Association unidirectionnelle 1-1

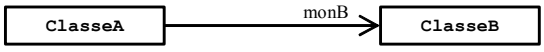

Figure B.19 : représentation UML d'une association unidirectionnelle 1-1

B.2.1.2 Association unidirectionnelle 1-plusieurs

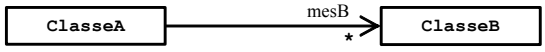

Figure B.20 : représentation UML d'une association unidirectionnelle 1-plusieurs

B.2.1.3 Association bidirectionnelle

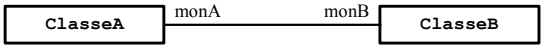

Figure B.21 : représentation UML d'une association bidirectionnelle

B.2.1.4 Association réflexive

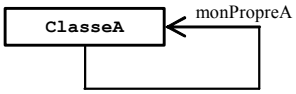

Figure B.22 : représentation UML d'une association réflexive

B.2.2 Agrégation

Utilisation d'une instance d'une classe par une autre classe en tant qu'attribut-objet avec un couplage fort et des durées de vies distinctes (*est constitué de* / *est fait de* / *fait partie de*) ; appelée aussi *agrégation par référence*.

Responsabilité vis-à-vis de la création de l'instance, même si celle-ci reste partageable avec d'autres instances (même classe ou autre classe) ; responsabilité vis-à-vis de la destruction de l'instance pas obligatoire mais fortement conseillée.

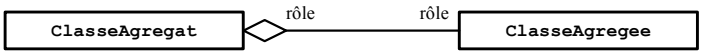

Figure B.23 : représentation UML d'une agrégation

B.2.2.1 Agrégation 1-1

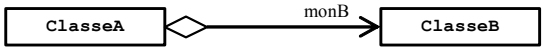

Figure B.24 : représentation UML d'une agrégation 1-1

B.2.2.2 Agrégation 1-plusieurs

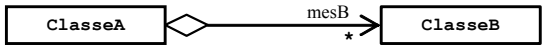

Figure B.25 : représentation UML d'une agrégation 1-plusieurs

B.2.2.3 Agrégation navigable

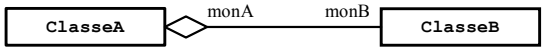

Figure B.26 : représentation UML d'une agrégation navigable

B.2.2.4 Agrégation réflexive

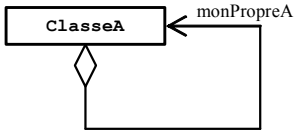

Figure B.27 : représentation UML d'une agrégation réflexive

B.2.3 Composition

Utilisation d'une instance d'une classe par une autre classe en tant qu'attribut-objet avec un couplage fort et des durées de vies identiques (*est composé de*) ; appelée aussi *agrégation par valeur*.

Responsabilité complète vis-à-vis de la création de l'instance ainsi que de la destruction de l'instance ; instance partageable avec aucune autre instance (même classe ou autre classe).

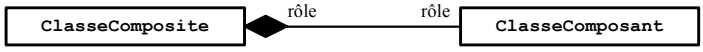

Figure B.28 : représentation UML d'une composition

Représentation alternative : classe imbriquée → instance partageable avec aucune autre instance (même classe ou autre classe) + classe utilisable par aucune autre classe.

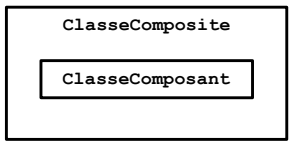

Figure B.29 : représentation UML d'une composition sous forme de classe imbriquée

B.2.3.1 Composition 1-1

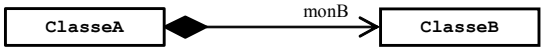

Figure B.30 : représentation UML d'une composition 1-1

B.2.3.2 Composition 1-plusieurs

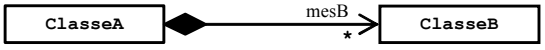

Figure B.31 : représentation UML d'une composition 1-plusieurs

B.2.3.3 Composition navigable

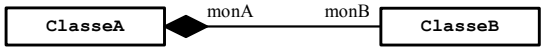

Figure B.32 : représentation UML d'une composition navigable

B.2.3.4 Composition réflexive

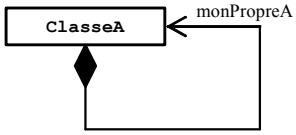

Figure B.33 : représentation UML d'une composition réflexive

B.3 SPÉCIALISATION / GÉNÉRALISATION

B.3.1 Spécialisation

Spécialisation : d'une super-classe vers une sous-classe ;
Généralisation : d'une sous-classe vers une super-classe.

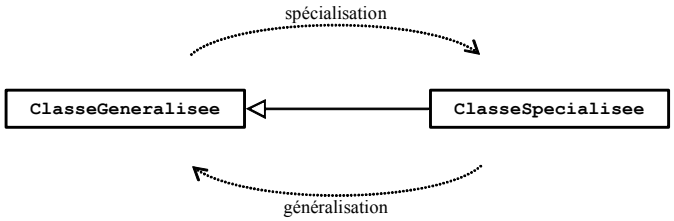

Figure B.34 : représentation UML d'une généralisation / spécialisation

B.3.2 Réalisation

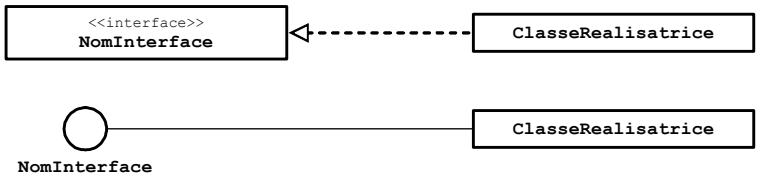

Figure B.35 : représentation UML d'une réalisation

B.4 CLASSE D'ASSOCIATION

Évolution d'une association vers le concept de classe, possédant à la fois les caractéristiques d'une association et celles d'une classe.

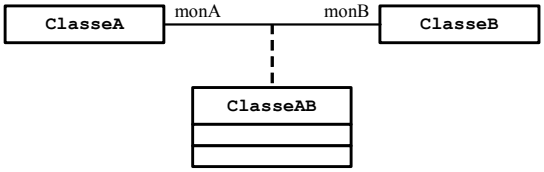

Figure B.36 : représentation UML d'une classe d'association

B.5 DÉPENDANCES

Instanciation, utilisation ou tout autre besoin sémantique entre deux entités (classe, interface, paquetage, instance, méthode, attribut, …) ne conditionnant ni n'apportant de signification quant à la structure interne de ces entité (*utilise* / *est abstraction de* / *est lié à* / *a permission sur*).

Concept très général pouvant signifier différents types de relations entre une entité A et une entité B, généralement caractérisé par l'usage d'un stéréotype ; 4 types de relations de dépendance :

- utilisation (stéréotype `<<use>>`) : stéréotypes `<<instanciate>>`, `<<create>>`, `<<call>>`, `<<send>>` ;
- abstraction : stéréotypes `<<derive>>`, `<<realize>>`, `<<refine>>`, `<<trace>>` ;
- liaison (stéréotype `<<bind>>`) ;
- permission (stéréotype `<<access>>`) : stéréotype `<<friend>>`.

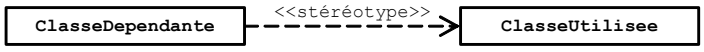

Figure B.37 : représentation UML d'une dépendance

C BIBLIOGRAPHIE

Alonso Stéphane, *Cours UML*, TS IRIS – LEGT Louis Modeste-Leroy – Évreux, 2004 ;

Dumas « Alex » Jean-Pierre, *Cours UML*, UFTI – IUFM Toulouse, 1998 ;

Muller Pierre-Alain, **Gaertner Nathalie**, *Modélisation objet avec UML*, Eyrolles, 2000 ;

Roques Pascal, *UML par la pratique*, Eyrolles, 2003 ;

Dumas « Alex » Jean-Pierre, *UML – Gestion d'une ferme*, UFTI – IUFM Toulouse, 1997 ;

Hérauville Stéphane, *Cours UML*, CFAI Marcel Sembat – Sotteville-lès-Rouen, 2002 ;

Courseille Annie, *Introduction au langage UML*, UFTI – IUFM Toulouse, 1998 ;

Roy Yves, *La notation UML*, 2002 ;

Charroux Benoît, *Cours UML*, http://perso.efrei.fr/~charroux/, EFREI – Villejuif, 1999 ;

Bouzy Bruno, *Notes de cours UML*, http://www.math-info.univ-paris5.fr/~bouzy/enseign.html, Centre de Recherche en Informatique de Paris 5 – UFR mathématiques et informatique – Université Paris Descartes, 2001 ;

Bichon Jean, *Cours de Système d'Information – Approche objet*,
Licence informatique – UFR Mathématiques et Informatique – Université Bordeaux 1, 2004 ;

Essanaa S., *Support de cours CSI*, http://www.esct.rnu.tn/site/etud.html,
École Supérieure de Commerce de Tunis – Université de La Manouba, 2007 ;

CommentÇaMarche.net, http://www.commentcamarche.net/, 2006 ;

Développez.com, *Langage UML*, http://uml.developpez.com/, 2006 ;

Wikipédia – l'encyclopédie libre, http://fr.wikipedia.org/, 2008 ;

Schoen D., *Module Génie Logiciel – Introduction*, FIUPSO – Université Paris-Sud Orsay, 1997.