INTRODUCTION AU LANGAGE UML

Table des matières

[Ce document présente les concepts et les principaux diagrammes du langage UML]

Introduction

L'évolution des techniques de programmation, a toujours été dictée par le besoin de concevoir et de maintenir des applications toujours plus complexes. La programmation par cartes perforées, a ainsi fait place à des techniques plus évoluées, comme l'assembleur (1947) ; puis il y a eu des langages plus évolués comme le Fortran où jusque là, les techniques de programmation étaient basées sur le branchement conditionnel et inconditionnel (goto), rendant les programmes importants extrêmement difficiles à développer, à maîtriser et à maintenir.
La programmation structurée (Pascal, C, …) a alors vu le jour et a permis de développer et de maintenir des applications toujours plus ambitieuses. L'algorithmique ne se suffisant plus à elle seule, à la fin des années 1970, le génie logiciel est venu placer la méthodologie au cœur du développement logiciel. Des méthodes comme Merise (1978) se sont alors imposées. La taille des applications ne cessant de croître, la programmation structurée a également rencontré ses limites, faisant alors place à la programmation orientée objet. La technologie objet est donc la conséquence ultime de la modularisation, dictée par la maîtrise de la conception et de la maintenance d'applications toujours plus complexes. Cette nouvelle technique de programmation a nécessité la conception de nouvelles méthodes de modélisation. UML (Unified Modeling Language en anglais, soit langage de modélisation unifié) est né de la fusion des trois méthodes qui s'imposaient dans le domaine de la modélisation obje : OMT, Booch et OOSE. D'importants acteurs industriels (IBM, Microsoft, Oracle, DEC, HP, Rational, Unisys etc.) s'associent alors à l'effort et proposent UML 1.0 à l'OMG (Object Management Group) qui l'accepte en novembre 1997 dans sa version 1.1. La version d'UML en cours est UML 2.1.2 qui s'impose plus que jamais en tant que langage de modélisation standardisé pour la modélisation des logiciels.
Ce document constitue le support du cours d'UML dispensé aux étudiants de niveau Master du département d'informatique de l'Université de Yaoundé I.

1. Introduction à la modélisation objet

1.1 Pourquoi et comment modéliser ?

Qu'est-ce qu'un modèle ?

Un modèle est une représentation abstraite et simplifiée (i.e. qui exclut certains détails), d'une entité (phénomène, processus, système, etc.) du monde réel en vue de le décrire, de l'expliquer ou de le prévoir.
Un modèle permet de réduire la complexité d'un phénomène en éliminant les détails qui n'influencent pas son comportement de manière significative. Il représente ce que le concepteur croit important pour la compréhension du phénomène modélisé.

Pourquoi modéliser ?

Modéliser un système avant sa réalisation permet de mieux comprendre le fonctionnement du système. C'est également un bon moyen de maîtriser sa complexité et d'assurer sa cohérence.
Un modèle est un langage commun, précis, qui est connu par tous les membres de l'équipe et il est donc, à ce titre, un vecteur privilégié pour communiquer. Dans le domaine de l'ingénierie du logiciel, le modèle permet de mieux répartir les tâches et d'automatiser certaines d'entre elles. C'est également un facteur de réduction des coûts et des délais. Par exemple, les plateformes de modélisation savent maintenant exploiter les modèles pour faire de la génération de code (au moins au niveau du squelette) voire des aller retours entre le code et le modèle sans perte d'information.
Le modèle est enfin indispensable pour assurer une bonne qualité et une maintenance efficace. En effet, une fois mise en production, l'application va devoir être maintenue, probablement par une autre équipe et, qui plus est, pas nécessairement de la même société que celle ayant créée l'application. Le choix du modèle a donc une influence capitale sur les solutions obtenues.

1.2 Méthodes d'analyse et de conception

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

La distinction entre composition et décomposition : Elle met en opposition d'une part les méthodes ascendantes qui consistent à construire un logiciel par composition à partir de modules existants et,

d'autre part, les méthodes descendantes qui décomposent récursivement le système jusqu'à arriver à des modules programmables simplement.

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

1.3 De la programmation structurée à l'approche orientée objet

1.3.1 Méthodes fonctionnelles

Egalement qualifiées de méthodes structurées, elles trouvent leur origine dans les langages procéduraux. Elles mettent en évidence les fonctions à assurer et proposent une approche hiérarchique descendante et modulaire. Ces méthodes utilisent les raffinements successifs pour produire des spécifications dont l'essentiel est sous forme de notation graphique en diagrammes de flots de données. Le plus haut niveau représente l'ensemble du problème (sous forme d'activité, de données ou de processus, selon la méthode). Chaque niveau est ensuite décomposé en respectant les entrées/sorties du niveau supérieur. La décomposition se poursuit jusqu'à arriver à des composants maîtrisables.
L'approche fonctionnelle dissocie le problème de la représentation des données, du problème du traitement de ces données. L'architecture du système est dictée par la réponse au problème (i.e. la fonction du système).

1.3.2 L'approche orientée objet

L'approche orientée objet considère le logiciel comme une collection d'objets dissociés, identifiés et possédant des caractéristiques. Une caractéristique est soit un attribut (donnée caractérisant l'état de l'objet), soit une entité comportementale de l'objet (fonction). La fonctionnalité du logiciel émerge alors de l'interaction entre les différents objets qui le constituent. L'une des particularités de cette approche est qu'elle rapproche les données et leurs traitements associés au sein d'un unique objet. Un objet est caractérisé par :
Une identité : permet de le distinguer des autres objets, indépendamment de son état. On construit généralement cette identité grâce à un identifiant découlant naturellement du problème. (Par exemple le code d'un produit, le numéro de série d'une voiture, etc.)
Les attributs : données caractérisant l'objet. Ce sont des variables stockant des informations sur l'état de l'objet.
Les méthodes : Caractérisent son comportement, c'est-à-dire l'ensemble des actions (appelées opérations) que l'objet est à même de réaliser. Ces opérations permettent de faire réagir l'objet aux sollicitations extérieures (ou d'agir sur les autres objets).
La Conception Orientée Objet (COO) est la méthode qui conduit à des architectures logicielles fondées sur les objets du système, plutôt que sur la fonction qu'il est censé réaliser.
L'architecture du système est dictée par la structure du problème.

1.4 Concepts importants de l'approche objet

Nous avons dit que l'approche objet rapproche les données et leurs traitements. Mais d'autres concepts importants sont spécifiques à cette approche et participent à la qualité du logiciel. Nous présentons les concepts de classe, d'encapsulation, d'héritage, d'agrégation et de polymorphisme.

Notion de classe

Une classe est un type de données abstrait qui précise des caractéristiques (attributs et méthodes) communes à toute une famille d'objets et qui permet de créer (instancier) des objets possédant ces caractéristiques.

Encapsulation

L'encapsulation consiste à masquer les détails d'implémentation d'un objet, en définissant une interface. L'interface est la vue externe d'un objet, elle définit les services accessibles (offerts) aux utilisateurs de l'objet. L'encapsulation facilite l'évolution d'une application car elle stabilise l'utilisation des objets : on peut modifier l'implémentation des attributs d'un objet sans modifier son interface, et

donc la façon dont l'objet est utilisé. L'encapsulation garantit l'intégrité des données, car elle permet d'interdire, ou de restreindre, l'accès direct aux attributs des objets.

Héritage, Spécialisation, Généralisation

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

Polymorphisme

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

Agrégation

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

1.5 Historique des modélisations par objets

Lorsque la programmation par objets prend de l'importance, la nécessité d'une méthode qui lui soit adaptée devient évidente. Plus de cinquante méthodes apparaissent entre 1990 et 1995 parmi lesquels : Booch, Classe-Relation, Fusion, HOOD, OMT, OOA, OOD, OOM, OOSE, etc. mais aucune ne parvient à s'imposer. En 1994, le consensus se fait autour de trois méthodes :

– OMT de James Rumbaugh (General Electric) fournit une représentation graphique des aspects statique, dynamique et fonctionnel d'un système ;

– OOD de Grady Booch, définie pour le DOD (département de la défense), introduit le concept de paquetage (package) ;

– OOSE d'Ivar Jacobson (Ericsson) fonde l'analyse sur la description des besoins des utilisateurs (cas d'utilisation, ou use cases).

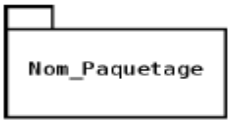
L'unification de ces 3 donne naissance à UML (Unified Modeling Language).

1.6 Notions générales du langage UML

Nous présentons ici quelques notions utilisées dans la modélisation objet avec UML.

1.6.1 Paquetage

Un paquetage est un regroupement d'éléments du modèle. Il permet ainsi d'organiser des éléments de modélisation en groupes. Il peut contenir tout type d'élément de modèle : des classes, des cas d'utilisation, des interfaces, etc. et même des paquetages imbriqués (décomposition hiérarchique). Un paquetage se représente comme un dossier avec son nom inscrit dedans ou dans l'onglet. Les éléments contenus dans un paquetage doivent représenter un ensemble fortement cohérent.



1.6.2 Espace de noms

Les espaces de noms sont des paquetages, des classeurs, etc. On peut déterminer un élément nommé de façon unique par son nom qualifié, qui est constitué de la série des noms des paquetages ou des autres espaces de noms depuis la racine jusqu'à l'élément en question. Dans un nom qualifié, chaque espace de nom est séparé par deux doubles points (::). Par exemple, si un paquetage B est inclus dans un paquetage A et contient une classe X, il faut écrire A::B::X pour pouvoir utiliser la classe X en dehors du contexte du paquetage B.

1.6.3 Stéréotype

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

rectangles pour représenter les classes, des use case ou des acteurs. La notation n'est cependant pas ambiguë grâce à la présence du stéréotype « use case ».

1.7 Les diagrammes d'UML

UML n'est pas une méthode, mais un langage graphique qui permet de représenter et de communiquer les divers aspects d'un système d'information. Aux graphiques sont bien sûr associés des textes qui expliquent leur contenu. UML est donc un métalangage car il fournit les éléments permettant de construire le modèle qui, lui, sera le langage du projet.

Il est impossible de donner une représentation graphique complète d'un logiciel, ou de tout autre système complexe, de même qu'il est impossible de représenter entièrement une statue (à trois dimensions) par des photographies (à deux dimensions). Mais il est possible de donner sur un tel système des vues partielles, analogues chacune à une photographie d'une statue, et dont la conjonction donnera une idée utilisable en pratique sans risque d'erreur grave. UML 2.0 comporte treize types de diagrammes représentant autant de vues distinctes pour représenter des concepts particuliers du système d'information. Ils se répartissent en deux grands groupes :

Les diagrammes structurels ou diagrammes statiques (UML Structure)
– diagramme de classes (Class diagram)
– diagramme d'objets (Object diagram)
– diagramme de composants (Component diagram)
– diagramme de déploiement (Deployment diagram)
– diagramme de paquetages (Package diagram)
– diagramme de structures composites (Composite structure diagram)
Les diagrammes comportementaux ou diagrammes dynamiques (UML Behavior)
– diagramme de cas d'utilisation (Use case diagram)
– diagramme d'activités (Activity diagram)
– diagramme d'états-transitions (State machine diagram)
– **Diagrammes d'interaction (Interaction diagram)**
  – diagramme de séquence (Sequence diagram)
  – diagramme de communication (Communication diagram)
  – diagramme global d'interaction (Interaction overview diagram)
  – diagramme de temps (Timing diagram)

Pour une modélisation, ces diagrammes ne sont pas nécessairement tous produits. Dans le cadre de ce cours nous étudierons 9 des 13 diagrammes : les diagrammes de cas d'utilisation, de classes, d'objets, d'états transition, d'activités, de collaboration, de séquence, de composants, et de déploiement.

2. Diagramme de cas d'utilisation (Use Case Diagram)

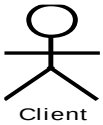
C'est le premier diagramme du modèle UML, celui où s'assure la relation entre l'utilisateur et les objets que le système met en œuvre. Il permet de recueillir, d'analyser et d'organiser les besoins, et de recenser les grandes fonctionnalités d'un système. Il s'agit donc de la première étape UML d'analyse d'un système. Il capture le comportement d'un système tel qu'un utilisateur extérieur le voit. Il scinde la fonctionnalité du système en unités cohérentes : les cas d'utilisation, ayant un sens pour les acteurs. Les cas d'utilisation permettent d'exprimer les besoins des utilisateurs d'un système. Pour élaborer les cas d'utilisation, il faut des entretiens avec les utilisateurs.

2.1 les éléments d'un diagramme de cas d'utilisation

On distingue 2 principaux concepts dans la construction d'un diagramme de cas d'utilisation : les acteurs et les cas d'utilisation.

2.1.1 Acteur

Un acteur est l'idéalisation d'un rôle joué par une personne externe, un processus ou une chose qui interagit avec un système. Il se représente par un petit bonhomme avec son nom (son rôle) inscrit dessous. Il est également possible de représenter un acteur sous la forme d'un classeur.

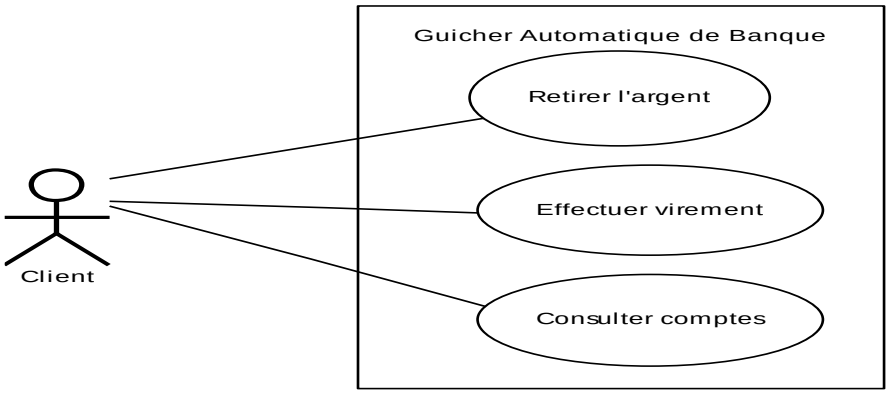


2.1.2 Cas d'utilisation

Un cas d'utilisation est une unité cohérente représentant une fonctionnalité visible de l'extérieur. Il réalise un service de bout en bout, avec un déclenchement, un déroulement et une fin, pour l'acteur qui l'initie. Il modélise donc un service rendu par le système, sans imposer le mode de réalisation de ce service. Il se représente par une ellipse contenant le nom du cas (verbe à l'infinitif), et optionnellement, au-dessus du nom, un stéréotype.

2.1.3 Représentation d'un diagramme de cas d'utilisation

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



2.2 Relations dans les diagrammes de cas d'utilisation

On distingue les relations entre acteurs et cas d'utilisation, entre acteurs et entre cas d'utilisation.

2.2.1 Relation entre acteurs et cas d'utilisation

Entre un acteur et un cas d'utilisation on a une relation d'association, qui est un chemin de communication entre l'acteur et ce cas d'utilisation. Il est représenté par un trait continu.

Acteurs principaux et secondaires

Un acteur peut être principal ou secondaire. Il est qualifié de principal pour un cas d'utilisation lorsque ce cas rend service à cet acteur. Les autres acteurs sont alors qualifiés de secondaires. Un cas d'utilisation a au plus un acteur principal. Un acteur principal obtient un résultat observable du système tandis qu'un acteur secondaire est sollicité pour des informations complémentaires. En général, l'acteur principal initie le cas d'utilisation par ses sollicitations. On utilise le stéréotype « primary » ou « secondary ».

Cas d'utilisation interne

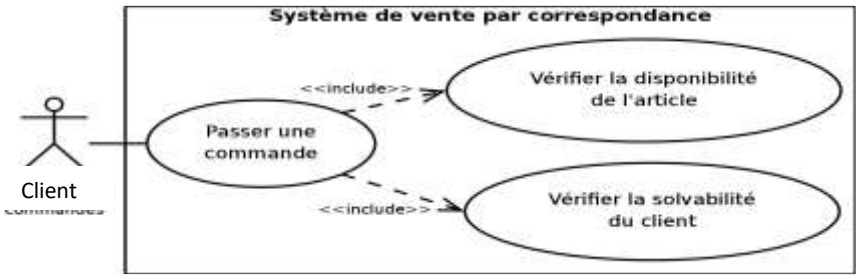
Quand un cas n'est pas directement relié à un acteur, il est qualifié de cas d'utilisation interne.

2.2.2 Relations entre cas d'utilisation

On peut définir une relation entre deux cas d'utilisation selon les cas.
On distingue les relations d'inclusion, d'extension et de généralisation/spécialisation.

Relation d'inclusion

Un cas A inclut un cas B si le comportement décrit par le cas A inclut le comportement du cas B : le cas A dépend de B. Lorsque A est sollicité, B l'est obligatoirement, comme une partie de A. Cette dépendance est symbolisée par le stéréotype « *include* ». Par exemple, l'accès aux informations d'un compte bancaire inclut nécessairement une phase d'authentification avec un identifiant et un mot de passe.
Les inclusions permettent essentiellement de factoriser une partie de la description d'un cas d'utilisation qui serait commune à d'autres cas d'utilisation.



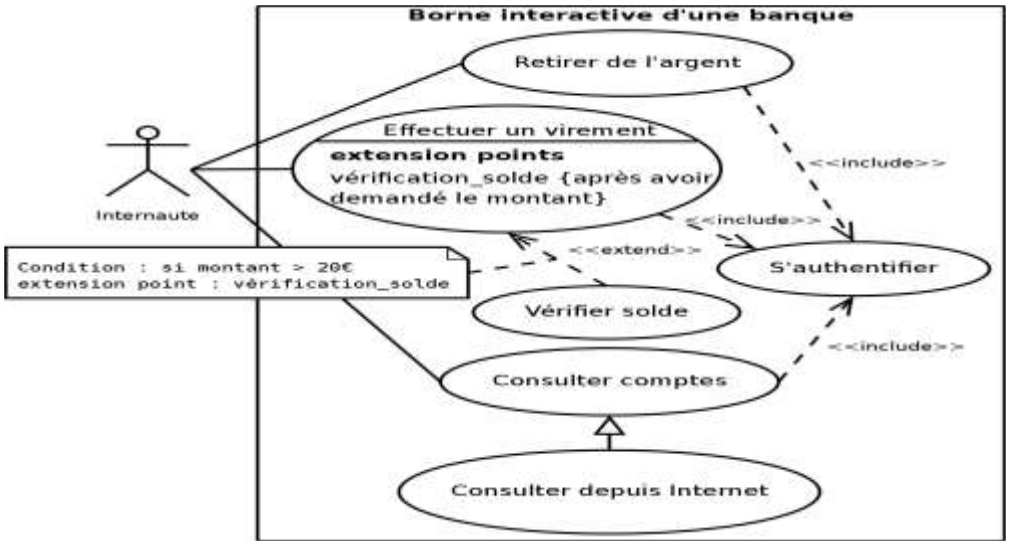
Relation d'extension

On dit qu'un cas d'utilisation A étend un cas d'utilisation B lorsque le cas d'utilisation A peut être appelé au cours de l'exécution du cas d'utilisation B. Exécuter B peut éventuellement entraîner l'exécution de A : contrairement à l'inclusion, l'extension est optionnelle. Cette dépendance est symbolisée par le stéréotype « *extend* ». Une extension est souvent soumise à condition. Graphiquement, la condition est exprimée sous la forme d'une note.

Relation de généralisation

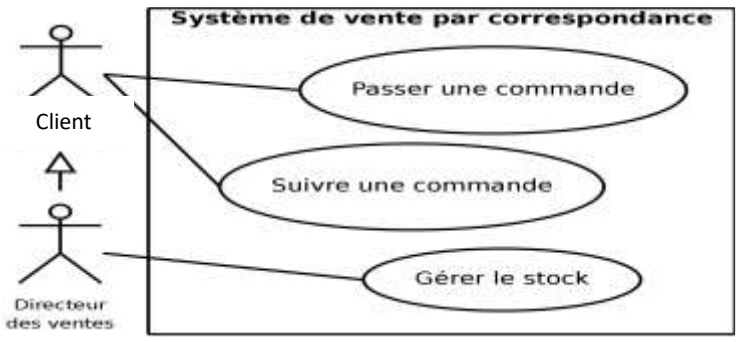
Un cas A est une généralisation d'un cas B si B est un cas particulier de A. Dans la figure ci-dessous, la consultation d'un compte via Internet est un cas particulier de la consultation.

Par exemple, la figure ci-dessous montre que le directeur de vente est un internaute avec un pouvoir supplémentaire : en plus de pouvoir effectuer les opérations en ligne, il peut gérer les stocks.



2.2.3 Relations entre acteurs

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



2.3 Description textuelle des cas d'utilisation

Le diagramme de cas d'utilisation décrit les grandes fonctions d'un système du point de vue des acteurs, mais n'expose pas de façon détaillée le dialogue entre les acteurs et les cas d'utilisation. Il est

donc recommandé de rédiger une description textuelle des use case, qui en général se compose de trois parties.

- La première partie permet d'identifier le cas :

Nom : Utiliser une tournure à l'infinitif (ex : Réceptionner un colis).
Objectif : Une description résumée permettant de comprendre l'intention principale du cas d'utilisation. Cette partie est souvent renseignée au début du projet dans la phase de découverte des cas d'utilisation.
Acteurs principaux : Ceux qui vont réaliser le cas d'utilisation (la relation avec le cas d'utilisation est illustrée par le trait liant le cas d'utilisation et l'acteur dans un diagramme de cas d'utilisation)
Acteurs secondaires : Ceux qui ne font que recevoir des informations à l'issue de la réalisation du cas d'utilisation
Dates : Les dates de créations et de mise à jour de la description courante.
Responsable : Le nom des responsables.
Version : Le numéro de version.

- La deuxième partie contient la description du fonctionnement du cas sous la forme d'une séquence de messages échangés entre les acteurs et le système.

On a toujours une séquence nominale décrivant le déroulement normal du cas, à laquelle s'ajoutent fréquemment des séquences alternatives (des embranchements dans la séquence nominale) et des séquences d'exceptions.
Les préconditions : elles décrivent dans quel état doit être le système (l'application) avant que ce cas d'utilisation puisse être déclenché.
Des scénarii : décrits sous la forme d'échanges d'évènements entre l'acteur et le système. On distingue le scénario nominal, qui se déroule quand il n'y a pas d'erreur, des scénarii alternatifs qui sont les variantes du scénario nominal et enfin les scénarii d'exception qui décrivent les cas d'erreurs.
Des postconditions : Elles décrivent l'état du système à l'issue des différents scénarii.

- La troisième partie de la description d'un cas d'utilisation est une rubrique optionnelle.

Elle contient généralement des spécifications non fonctionnelles (spécifications techniques,
. . .).

3. Diagramme de classes

Le diagramme de classes est considéré comme le plus important de la modélisation orientée objet, il est le seul obligatoire lors d'une telle modélisation. Alors que le diagramme de cas d'utilisation montre un système du point de vue des acteurs, le diagramme de classes en montre la structure interne. Il permet de fournir une représentation abstraite des objets du système qui vont interagir ensemble pour réaliser les cas d'utilisation. Un même objet peut très bien intervenir dans la réalisation de plusieurs cas d'utilisation. C'est une vue statique car on ne tient pas compte du facteur temporel dans le comportement du système. Le diagramme de classes modélise les concepts du domaine d'application ainsi que les concepts internes créés de toutes pièces dans le cadre de l'implémentation d'une application.
Les principaux éléments de cette vue statique sont les classes et leurs relations : association, généralisation et plusieurs types de dépendances, telles que la réalisation et l'utilisation.

3.1 Les classes

3.1.1 Instance de classe

Une instance est une concrétisation d'un concept abstrait. Par exemple :
– la Mercedes compressor est une instance du concept abstrait ***voiture*** ;
– l'amitié qui lie Jean et Marie est une instance du concept abstrait ***Amitié*** ;
Une classe est un concept abstrait représentant des éléments variés comme des :

- éléments concrets (ex : étudiant, avions, voiture,…),
- éléments abstraits (ex : commandes de marchandises ou services),
- Composants d'une application (ex : les boutons des boîtes de dialogue), etc.

Tout système orienté objet est organisé autour des classes.
Une classe est la description formelle d'un ensemble d'objets ayant une sémantique et des caractéristiques communes.
Un objet est une instance d'une classe. C'est une entité discrète dotée d'une identité, d'un état et d'un comportement que l'on peut invoquer. Par exemple, si l'on considère que ***Homme*** est une classe, on pourra dire que la personne ***francky*** est une instance de Homme, c'est-à-dire un objet.

3.1.2 Caractéristiques d'une classe

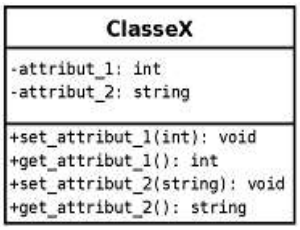
Une classe définit des objets ayant des caractéristiques communes. Les caractéristiques d'un objet permettent de spécifier son état et son comportement.

- **État d'un objet** : Ce sont les attributs et généralement les terminaisons d'associations, tous deux réunis sous le terme de propriétés structurelles, ou tout simplement propriétés, qui décrivent l'état d'un objet. Les associations sont utilisées pour connecter les classes du diagramme de classe; dans ce cas, la terminaison de l'association (du côté de la classe cible) est généralement une propriété de la classe de base. Les propriétés décrites par les attributs prennent des valeurs lorsque la classe est instanciée. L'instance d'une association est appelée un lien.
- **Comportement d'un objet** : Les opérations décrivent les éléments individuels d'un comportement que l'on peut invoquer. Ce sont des fonctions qui peuvent prendre des valeurs en entrée et modifier les attributs ou produire des résultats. Une opération est la spécification (déclaration) d'une méthode.

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

3.1.3 Représentation graphique

Une classe est représentée par un rectangle divisé en général en trois compartiments.
Le premier indique le nom de la classe, le deuxième ses attributs et le troisième ses opérations.



3.1.4 Encapsulation, visibilité, interface

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

- ***Public*** ou + : tout élément qui peut voir le conteneur peut également voir l'élément indiqué.
- ***Protected*** ou # : seul un élément situé dans le conteneur ou un de ses descendants peut voir l'élément indiqué.
- ***Private*** ou - : seul un élément situé dans le conteneur peut voir l'élément.
- ***Package*** ou ~ ou rien : seul un élément déclaré dans le même paquetage peut voir l'élément.

Dans une classe, le marqueur de visibilité se situe au niveau de chacune de ses caractéristiques (attributs, terminaisons d'association et opération). Il permet d'indiquer si une autre classe peut y accéder. Dans un paquetage, le marqueur de visibilité se situe sur des éléments contenus directement dans le paquetage, comme les classes, les paquetages imbriqués, etc. Il indique si un autre paquetage susceptible d'accéder au premier paquetage peut voir les éléments.

3.1.5 Nom d'une classe

Le nom de la classe doit évoquer le concept décrit par la classe. Il commence par une majuscule. On peut ajouter d'autres informations comme le nom de l'auteur de la modélisation, la date, etc.
La syntaxe de base de la déclaration d'un nom d'une classe est la suivante :
[<Nom_du_paquetage_1>::...::<Nom_du_paquetage_N>] <Nom_de_la_classe> [{ [abstract], [<auteur>], [<date>], ... }]

3.1.6 Les attributs

Attributs de la classe

Les attributs définissent des informations qu'une classe ou un objet doivent connaître. Ils représentent les données encapsulées dans les objets de cette classe. Chacune de ces informations est définie par un nom, un type de données, une visibilité et peut être initialisé.

Attributs de classe

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

Attributs dérivés

Les attributs dérivés peuvent être calculés à partir d'autres attributs et de formules de calcul. Les attributs dérivés sont symbolisés par l'ajout d'un « / » devant leur nom.

3.1.7 Les méthodes

Méthode de la classe

Dans une classe, une opération (même nom et même types de paramètres) doit être unique.
Quand le nom d'une opération apparaît plusieurs fois avec des paramètres différents, on dit que l'opération est surchargée. En revanche, il est impossible que deux opérations ne se distinguent que par leur valeur retournée.
La déclaration d'une opération contient les types des paramètres et le type de la valeur de retour, sa syntaxe est la suivante :
<visibilité> <nom_méthode> ([<paramètre_1>, ... , <paramètre_N>]) : [<type_renvoyé>] [{<propriétés>}]
La syntaxe de définition d'un paramètre (<paramètre>) est la suivante :
[<direction>] <nom_paramètre>:<type> ['['<multiplicité>']'] [=<valeur_par_défaut>]
La direction peut prendre l'une des valeurs suivante :
in : Paramètre d'entrée passé par valeur. Les modifications du paramètre ne sont pas disponibles pour l'appelant. C'est le comportement par défaut.
out : Paramètre de sortie uniquement. Il n'y a pas de valeur d'entrée et la valeur finale est disponible pour l'appelant.
inout : Paramètre d'entrée/sortie. La valeur finale est disponible pour l'appelant.
Le type du paramètre (<type>) peut être un nom de classe, un nom d'interface ou un type de donné prédéfini. Les propriétés (<propriétés>) correspondent à des contraintes ou à des informations complémentaires comme les exceptions, les préconditions, les postconditions ou encore l'indication qu'une méthode est abstraite (mot-clef abstract), etc.

Méthode de classe

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

Méthodes et classes abstraites

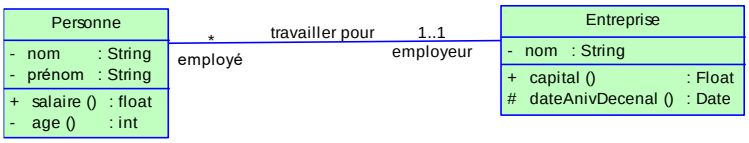
Une méthode est dite abstraite lorsqu'on connaît son entête mais pas la manière dont elle peut être réalisée (on connaît sa déclaration mais pas sa définition).
Une classe est dite abstraite lorsqu'elle définit au moins une méthode abstraite ou lorsqu'une classe parent contient une méthode abstraite non encore réalisée.
On ne peut instancier une classe abstraite : elle est vouée à se spécialiser.
Une classe abstraite peut très bien contenir des méthodes concrètes.

Une classe abstraite pure ne comporte que des méthodes abstraites. En programmation orientée objet, une telle classe est appelée une interface. On utilise le steriotype <abstract>.

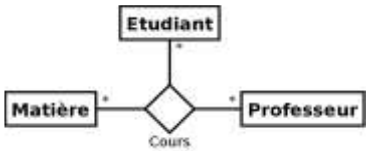
3.2 Relations entre classes

3.2.1 Association

Une association est une relation entre deux classes (association binaire) ou plus (association n-aire), qui décrit les connexions structurelles entre leurs instances. Elle indique qu'il peut y avoir des liens entre des instances des classes associées.
Une association binaire est matérialisée par un trait plein entre les classes associées. Elle peut être ornée d'un nom, avec éventuellement une précision du sens de lecture.



Quand les deux extrémités de l'association pointent vers la même classe, l'association est dite réflexive.
Une association n-aire lie plus de deux classes. La ligne pointillée d'une classe-association peut être reliée au losange par une ligne discontinue pour représenter une association n-aire dotée d'attributs, d'opérations ou d'associations. On représente une association n-aire par un grand losange avec un chemin partant vers chaque classe participante. Si l'association a un nom, on le met à proximité du losange.



3.2.2 Multiplicité ou cardinalité

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

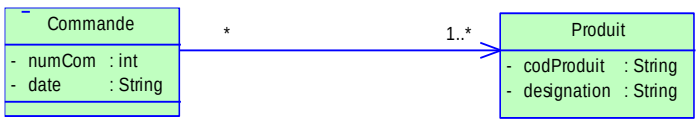
- exactement un : 1 ou 1..1
- plusieurs : * ou 0..*
- au moins un : 1..*
- de un à six : 1..6

Dans une association binaire, la multiplicité sur la terminaison cible défini le nombre d'objets de la classe cible pouvant être associés à un seul objet donné de la classe source. Dans une association n-aire, la multiplicité apparaissant sur le lien de chaque classe s'applique sur une instance de chacune des classes, à l'exclusion de la classe-association et de la classe considérée. Par exemple, si on prend une association ternaire entre les classes (A, B, C), la multiplicité de la terminaison C indique le nombre d'objets C qui peuvent apparaître dans l'association avec une paire particulière d'objets A et B.

Remarque 1 : Pour une association n-aire, la multiplicité minimale doit en principe, mais pas nécessairement, être 0. En effet, une multiplicité minimale de 1 (ou plus) sur une extrémité implique qu'il doit exister un lien (ou plus) pour TOUTES les combinaisons possibles des instances des classes situées aux autres extrémités de l'association n-aire !
Remarque 2 : Pour les habitués du modèle entité/relation, les multiplicités sont « à l'envers » en UML (par référence à Merise) pour les associations binaires et « à l'endroit » pour les n-aires avec n > 2.

3.2.3 Navigabilité

La navigabilité indique s'il est possible de traverser une association. On représente graphiquement la navigabilité par une flèche du côté de la terminaison navigable. Par défaut, une association est navigable dans les deux sens.



Dans cet exemple les instances de la classe Produit ne stockent pas de liste d'objets du type Commande. Inversement, chaque objet commande contient une liste de produits (terminaison navigable du côté de la classe Produit)

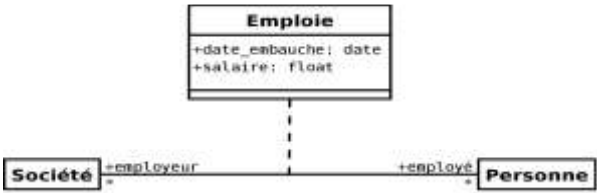
3.2.4 Qualification

Quand une classe est liée à une autre classe par une association, il est parfois préférable de restreindre la portée de l'association à quelques éléments ciblés (comme un ou plusieurs attributs) de la classe. Ces éléments ciblés sont appelés un qualificatif. L'objet sélectionné par la valeur du qualificatif est appelé objet cible. L'association est appelée association qualifiée. Un qualificatif agit toujours sur une association dont la multiplicité est plusieurs du côté cible.

Un objet qualifié et une valeur de qualificatif génèrent un objet cible lié unique. En considérant un objet qualifié, chaque valeur de qualificatif désigne un objet cible unique.
Par exemple le diagramme ci-dessus nous dit que :
– Un compte dans une banque appartient à au plus deux personnes. Autrement dit, une instance du couple {Banque, compte} est en association avec zéro à deux instances de la classe Personne.
– Mais une personne peut posséder plusieurs comptes dans plusieurs banques. C'est-à dire qu'une instance de la classe Personne peut être associée à plusieurs (zéro compris) instances du couple {Banque, compte}.
– Bien entendu, et dans tous les cas, une instance du couple {Personne, compte} est en association avec une instance unique de la classe Banque.

3.2.5 Classe-association

Une classe-association possède les caractéristiques des associations et des classes, et est utilisée lorsqu'une association doit posséder des propriétés. Par exemple, l'association ***Emploie*** entre une société et une personne possède comme propriétés le salaire et la date d'embauche. Elle est caractérisée par un trait discontinu entre la classe et l'association qu'elle représente.



NB : Il n'est pas possible de rattacher une classe-association à plus d'une association.

Agrégation et composition

Agrégation

Association représentant une relation d'inclusion structurelle ou comportementale d'un élément dans un ensemble.
Dans une association, aucune des deux classes n'est plus importante que l'autre. Si l'on souhaite modéliser une relation *tout/partie* où une classe constitue un élément plus grand (tout) composé d'éléments plus petit (partie), il faut utiliser une agrégation.
Graphiquement, on ajoute un losange vide du côté de l'agrégat. Contrairement à une association simple, l'agrégation est transitive. La signification de cette forme simple d'agrégation est uniquement conceptuelle. Elle n'entraîne pas de contrainte sur la durée de vie des parties par rapport au tout.

Composition

La composition, également appelée agrégation composite, décrit une contenance structurelle entre instances. Ainsi, la destruction de l'objet composite implique la destruction de ses composants. Une

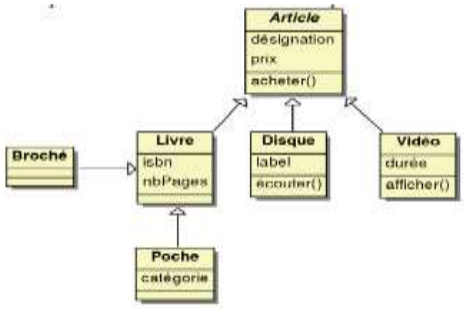
instance de la partie appartient toujours à au plus une instance de l'élément composite : la multiplicité du côté composite ne doit pas être supérieure à 1 (i.e. 1 ou 0..1).
Graphiquement, on ajoute un losange plein du côté de l'agrégat.

Généralisation et Héritage

L'héritage permet la classification des objets. La généralisation décrit une relation entre une classe générale (classe de base ou classe parent) et une classe spécialisée (sous-classe). La classe spécialisée possède intégralement les caractéristiques de la classe de base, mais comporte des informations supplémentaires (attributs, opérations, associations).
Les propriétés principales de l'héritage sont :

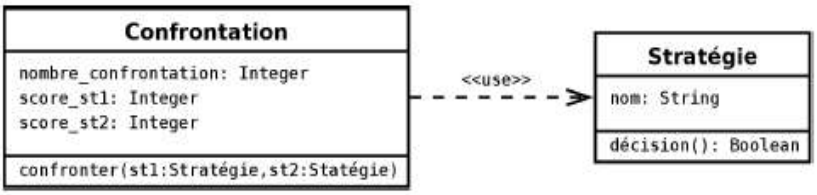
– La classe enfant possède toutes les caractéristiques des ses classes parents.
– Une classe enfant peut redéfinir (même signature) une ou plusieurs méthodes de la classe parent.
– Toutes les associations de la classe parent s'appliquent aux classes dérivées.
– Une instance d'une classe peut être utilisée partout où une instance de sa classe parent est attendue.
– Une classe peut avoir plusieurs parents, on parle alors d'héritage multiple

NB : Le langage C++ permet son implémentation, mais pas le langage Java.



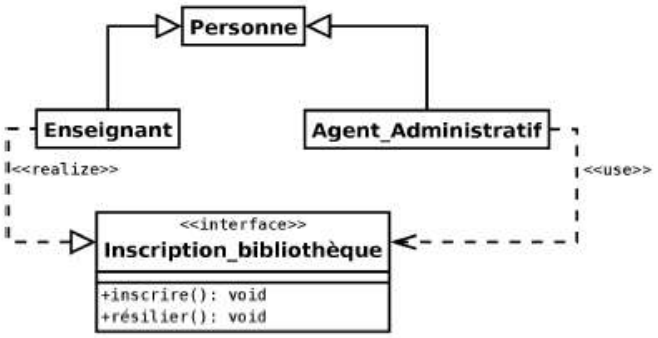
Dépendance

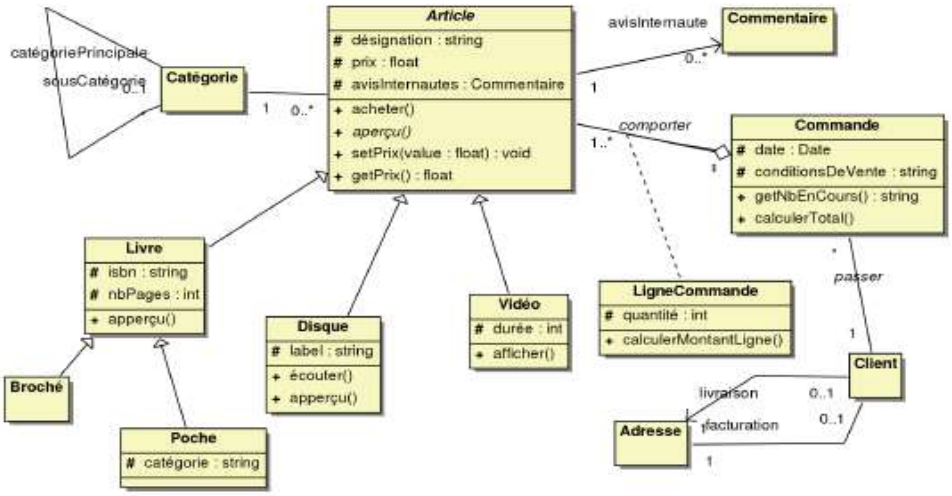
Une dépendance est une relation unidirectionnelle exprimant une dépendance sémantique entre des éléments du modèle. Elle est représentée par un trait discontinu orienté. Elle indique que la modification de la cible peut impliquer une modification de la source. La dépendance est souvent stéréotypée. On utilise souvent une dépendance quand une classe utilise une autre comme argument dans la signature d'une opération.
Par exemple, le diagramme ci-dessous montre que la classe Confrontation utilise la classe Stratégie car la classe Confrontation possède une méthode ***confronter*** dont deux paramètre sont du type Stratégie.



Interfaces

C'est un classeur, stéréotypé « *interface* », et dont le rôle est de regrouper un ensemble de propriétés et d'opérations assurant un service cohérent. Une interface est représentée comme une classe excepté l'ajout du stéréotype « interface».
Une interface doit être réalisée par au moins une classe et peut l'être par plusieurs. Graphiquement, cela est représenté par un trait discontinu terminé par une flèche triangulaire et le stéréotype « *realize* ». Une classe peut très bien réaliser plusieurs interfaces. Une classe (classe cliente de l'interface) peut dépendre d'une interface (interface requise). On représente cela par une relation de dépendance et le stéréotype « *use* ». Attention aux problèmes de conflits si une classe dépend d'une interface réalisée par plusieurs autres classes.





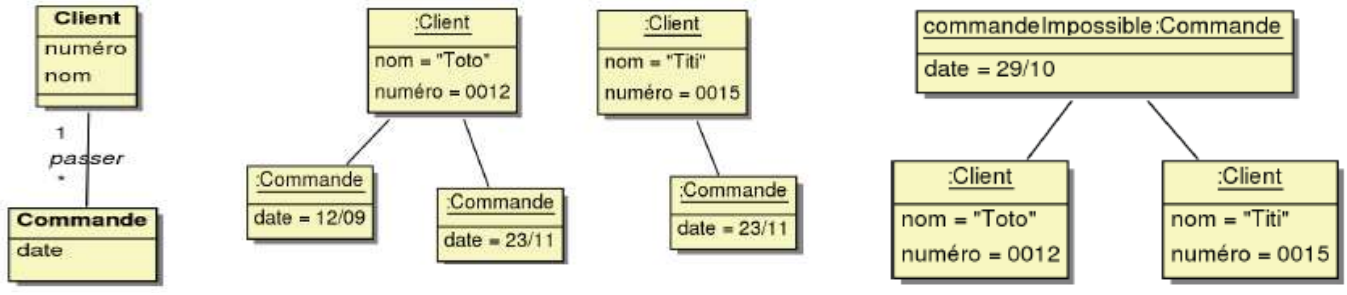
4. Diagramme d'objets (object diagram)

4.1 Présentation

Un diagramme d'objets représente des objets (instances de classes) et leurs liens (instances de relations) pour donner une vue figée de l'état d'un système à un instant donné. Il peut être utilisé pour :
– illustrer le modèle de classes en montrant un exemple qui explique le modèle ;
– préciser certains aspects du système en mettant en évidence des détails imperceptibles dans le diagramme de classes ;
– exprimer une exception en modélisant des cas particuliers ou des connaissances non généralisables qui ne sont pas modélisés dans un diagramme de classe ;
– prendre une image (snapshot) d'un système à un moment donné.
Le diagramme de classes modélise les règles et le diagramme d'objets modélise des faits.
Un diagramme d'objets ne montre pas l'évolution du système dans le temps. Pour représenter une interaction, il faut utiliser un diagramme de communication ou de séquence.

4.2 Représentation

La figure ci-dessous montre : (A) un diagramme de classe ; (B) un diagramme d'objets cohérent avec ce diagramme de classes ; (C) un diagramme d'objets incohérent avec le diagramme de classes.

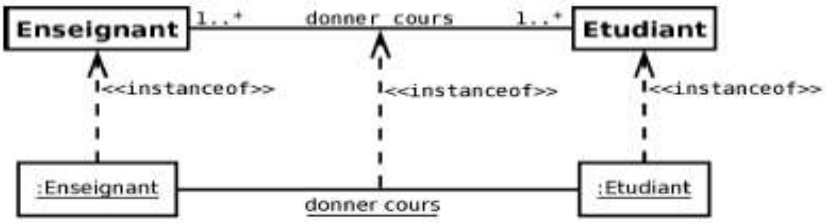


A B C

Graphiquement, un objet se représente comme une classe. Cependant, le compartiment des opérations n'est pas utile. De plus, le nom de la classe dont l'objet est une instance est précédé d'un « : » et est souligné. Pour différencier les objets d'une même classe, leur identifiant peut être ajouté devant le nom de la classe. Enfin les attributs reçoivent des valeurs. Quand certaines valeurs d'attribut d'un objet ne sont pas renseignées, on dit que l'objet est partiellement défini.
La relation de généralisation ne possède pas d'instance, elle n'est donc jamais représentée dans un diagramme d'objets. Graphiquement, un lien se représente comme une relation, mais, s'il y a un nom, il est souligné. Naturellement, on ne représente pas les multiplicités.

4.3 Relation de dépendance d'instanciation

Elle est stéréotypée « instanceof » et décrit la relation entre un classeur et ses instances. Elle relie, en particulier, les liens aux associations et les objets aux classes.



5. Diagramme d'états-transitions

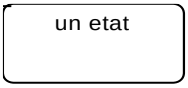
5.1 Présentation

Les diagrammes d'états-transitions d'UML décrivent le comportement interne d'un objet à l'aide d'un automate à états finis. Ils présentent les séquences possibles d'états et d'actions qu'une instance de classe peut traiter au cours de son cycle de vie en réaction à des événements. Ils spécifient habituellement le comportement d'une instance de classe. Mais parfois aussi le comportement interne d'autres éléments tels que les cas d'utilisation, les sous-systèmes, les méthodes.
Ce diagramme rassemble et organise les états et les transitions d'une classe donnée. Il est souhaitable de construire un diagramme d'états-transitions pour chaque classe possédant un comportement dynamique important. Il ne peut être associé qu'à une seule classe.

5.2 Eléments d'un diagramme d'états-transitions

5.2.1 État

Un état représente une période dans la vie d'un objet pendant laquelle ce dernier attend un événement ou accomplit une activité. Un objet peut passer par une série d'états pendant sa durée de vie. On le représente par un rectangle aux coins arrondis.



Certains états, dits composites, peuvent contenir (envelopper) des sous états.
Le nom de l'état peut être spécifié dans le rectangle et doit être unique dans le diagramme d'états-transitions, ou dans l'état enveloppant. Un état peut être anonyme.

5.2.2 État initial et final

Ce sont des pseudos états. Le premier indique l'état de départ, par défaut, lorsque le diagramme d'états-transitions est invoqué. Lorsqu'un objet est créé, il entre dans l'état initial.
Le second indique que le diagramme d'états-transitions, ou l'état enveloppant, est terminé.

5.2.3 Événement

Un événement est quelque chose qui se produit pendant l'exécution d'un système et qui doit être modélisé. Les transitions d'un diagramme d'état transition sont déclenchées par des évènements déclencheurs. Il se produit à un instant précis et est dépourvu de durée. Quand un événement est reçu, une transition peut être déclenchée et faire basculer l'objet dans un nouvel état. On en distingue plusieurs types : signal, appel, changement et temporel.

a. Événement de type signal (signal) : Un signal est un type d'évènement destiné à véhiculer une communication asynchrone à sens unique entre deux objets. L'objet expéditeur crée et initialise explicitement une instance de signal et l'envoi à un objet ou groupe d'objets. La réception d'un signal est un événement pour l'objet destinataire. La syntaxe d'un signal est la suivante :
nom_événement ([paramettre : type])

b. Événement d'appel (call) : Un événement d'appel représente la réception de l'appel d'une opération par un objet. Ce sont des opérations déclarées au niveau du diagramme de cladsses.Les paramètres de l'opération sont ceux de l'événement d'appel. La syntaxe d'un événement d'appel est la même que celle d'un signal. Par contre, les événements d'appel sont des méthodes déclarées au niveau du diagramme de classes. **nom_événement ([paramettre : type])**

c. Événement de changement (change) : il est généré par la satisfaction (passage de faux à vrai) d'une expression booléenne sur des valeurs d'attributs. Il s'agit d'une manière déclarative d'attendre qu'une condition soit satisfaite. La syntaxe d'un événement de changement est la suivante : **when (condition_booléenne)**

d. Événement temporel (after) : ils sont générés par le passage du temps. Ils sont spécifiés soit de manière absolue (date précise), soit de manière relative (temps écoulé). Par défaut, le temps commence à s'écouler dès l'entrée dans l'état courant.

La syntaxe d'un événement temporel spécifié de manière relative est la suivante :
after (durée)
Un événement temporel spécifié de manière absolue est défini en utilisant un événement de changement : **when (date = <date>)**

5.2.4 Transition

Une transition définit la réponse d'un objet à l'occurrence d'un événement. Elle lie, généralement, deux états E1 et E2 et indique qu'un objet dans un état E1 peut entrer dans l'état E2 et exécuter certaines activités, si un événement déclencheur se produit et que la condition de garde est vérifiée.
La syntaxe d'une transition est la suivante : **nomEvenement (params) [garde] / activite**
Le même événement peut être le déclencheur de plusieurs transitions quittant un même état.
Chaque transition avec le même événement doit avoir une condition de garde différente.

a. Condition de garde

La garde désigne une condition qui doit être remplie pour pouvoir déclencher la transition. L'activité désigne des instructions à effectuer au moment du tir.

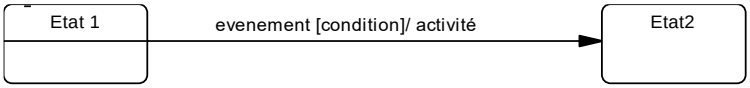
C'est une expression logique sur les attributs de l'objet, associé au diagramme d'états-transitions, ainsi que sur les paramètres de l'événement déclencheur. Elle est évaluée uniquement lorsque l'événement déclencheur se produit. Si l'expression est fausse, la transition ne se déclenche pas, sinon, la transition se déclenche et ses effets se produisent.

b. Effet d'une transition

Lorsqu'une transition se déclenche, son effet (spécifié par '/' <activité> dans la syntaxe) s'exécute. Il s'agit généralement d'une activité qui peut être
– une opération primitive comme une instruction d'assignation ;
– l'envoi d'un signal ;
– l'appel d'une opération ;
– une liste d'activités, etc.
La façon de spécifier l'activité à réaliser est laissée libre (langage naturel ou pseudo-code).
Lorsque l'exécution de l'effet est terminée, l'état cible de la transition devient actif.

c. Transition externe

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



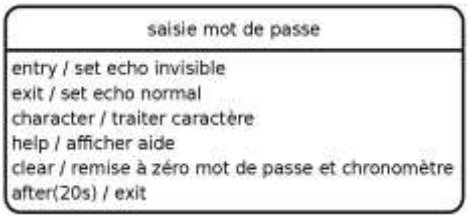
d. Transition d'achèvement

Une transition dépourvue d'événement déclencheur explicite se déclenche à la fin de l'activité contenue dans l'état source (y compris les états imbriqués). Elle peut contenir une condition de garde qui est évaluée au moment où l'activité contenue dans l'état s'achève. Ces transitions sont, par exemple, utilisées pour connecter les états initiaux avec leur état successeurs.

e. Transition interne

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

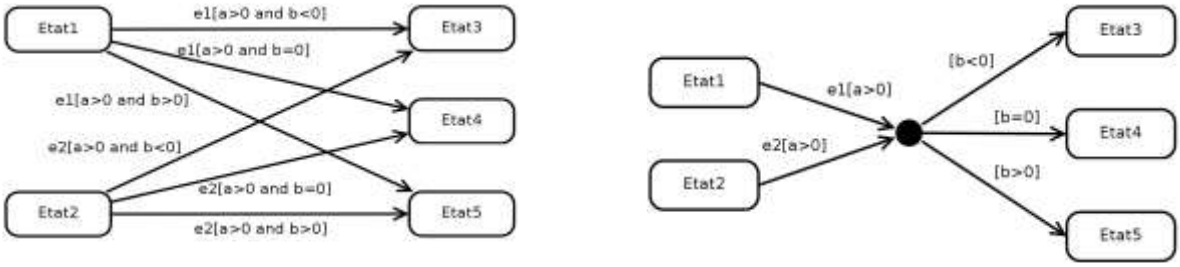
Les transitions internes possèdent des noms d'événement prédéfinis correspondant à des déclencheurs particuliers : *entry, exit, do* et *include*. Ces mots clefs réservés viennent prendre la place du nom de l'événement dans la syntaxe d'une transition interne.
entry : permet de spécifier une activité qui s'accomplit quand on entre dans l'état.
exit : permet de spécifier une activité qui s'accomplit quand on sort de l'état.
do : Une activité do commence dès que l'activité entry est terminée. Lorsque cette activité est terminée, une transition d'achèvement peut être déclenchée, après l'exécution de l'activité exit bien entendu.
include : permet d'invoquer un sous-diagramme d'états-transitions.



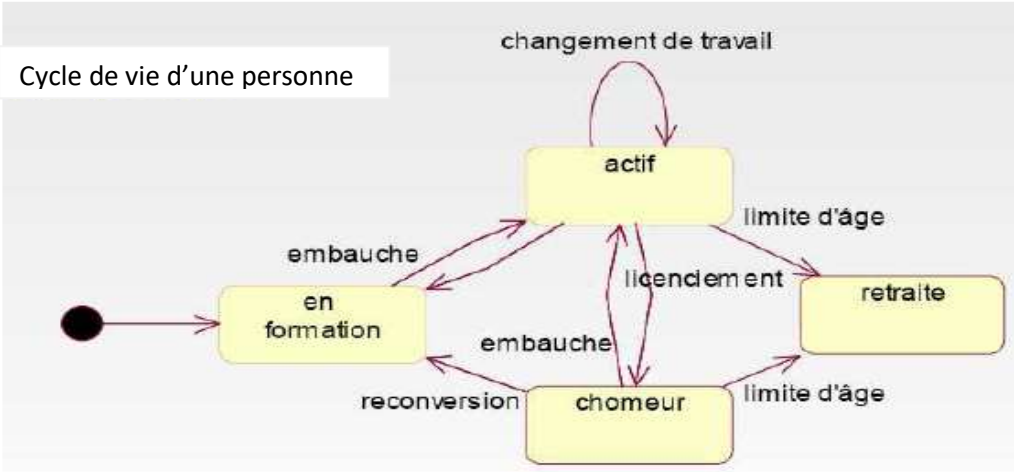
5.2.5 Point de choix

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

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



Un point de décision possède une entrée et au moins deux sorties. Les gardes situées après le point de décision sont évaluées au moment où il est atteint. Cela permet de baser le choix sur des résultats obtenus en franchissant le segment avant le point de choix. Il est possible d'utiliser une garde particulière, notée [else], sur un des segments en aval d'un point de choix. Ce segment n'est franchissable que si les gardes des autres segments sont toutes fausses.



6. Diagramme d'activités

6.1 Présentation

Les diagrammes d'activités permettent de représenter graphiquement le comportement d'une méthode ou le déroulement d'un cas d'utilisation. Ils mettent l'accent sur les traitements. Contrairement au diagramme d'états-transitions, les diagrammes d'activités ne sont pas spécifiquement rattachés à un classeur particulier. Dans la phase de conception, les diagrammes d'activités sont particulièrement adaptés à la description des cas d'utilisation.

6.2 Concepts

6.2.1 Action

La notion d'action est à rapprocher de la notion d'instruction élémentaire d'un langage de programmation (C++, Java). Une action est le plus petit traitement qui puisse être exprimé en UML. Une action peut être, par exemple :
– une affectation de valeur à des attributs ;
– la création d'un nouvel objet ou lien ;
– un calcul arithmétique simple ;
– l'émission d'un signal ;
– la réception d'un signal ;

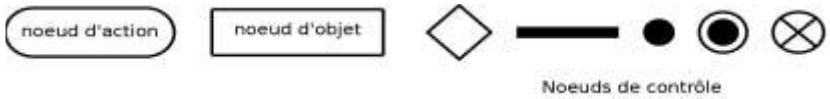
6.2.2 Activité

Une activité définit un comportement décrit par un séquencement organisé d'unités dont les éléments simples sont les actions. Le flot d'exécution est modélisé par des nœuds reliés par des arcs (transitions).

6.2.3 Nœud d'activité

Un nœud d'activité est un élément permettant de représenter les étapes d'une activité. Il existe trois familles de nœuds d'activités :

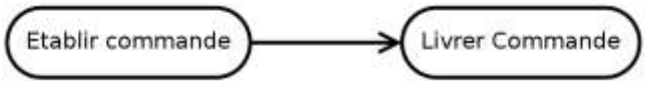
- les nœuds d'exécutions ou exécutable : nœud d'activité qu'on peut exécuter.
- Les nœuds d'objets : permet de définir un flot d'objet (i.e. un flot de données) dans un diagramme d'activités. Ce nœud représente l'existence d'un objet généré par une action dans une activité et utilisé par d'autres actions.
- les nœuds de contrôle :



De la gauche vers la droite, on trouve :
Le nœud représentant une action, qui est une variété de nœud d'exécution, un nœud objet, un nœud de décision ou de fusion, un nœud de bifurcation ou d'union, un nœud initial, un nœud final et un nœud final de flot.

6.2.4 Transition

Elle matérialise le passage d'un nœud d'activité vers un autre. Graphiquement les transitions sont représentées par des flèches en traits pleins qui connectent les activités (nœuds) entre elles. Elles sont déclenchées dès que l'activité source est terminée et provoquent automatiquement et immédiatement le début de la prochaine activité à déclencher (l'activité cible).

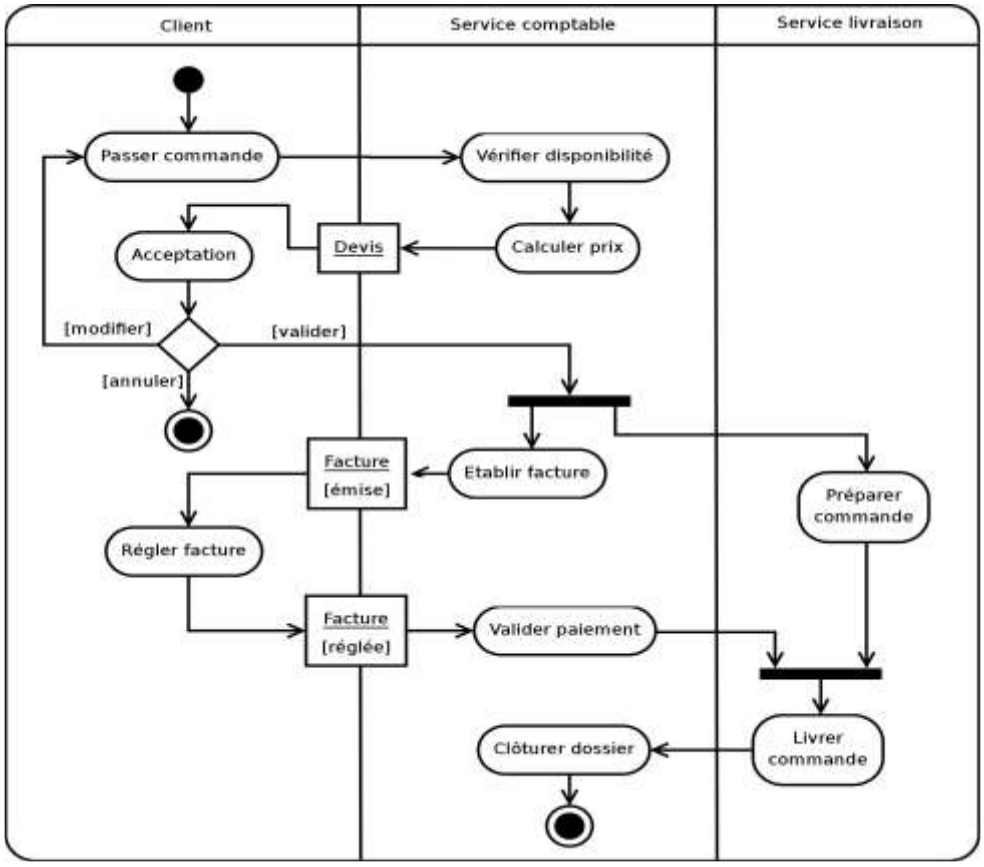


6.2.5 Partitions

Les partitions, souvent appelées couloirs d'activités ou lignes d'eau du fait de leur notation, permettent d'organiser les nœuds d'activités dans un diagramme d'activités en opérant des

regroupements. On peut, par exemple, les utiliser pour spécifier la classe responsable de la mise en œuvre d'un ensemble de tâches. Dans ce cas, la classe en question est responsable de l'implémentation du comportement des nœuds inclus dans ladite partition.
Elles sont représentées par des lignes continues. Il s'agit généralement de lignes verticales, mais elles peuvent être horizontales ou même courbes. Les transitions peuvent, bien entendu, traverser les frontières des partitions.

6.3 Représentation



7. Diagrammes d'interaction

Un objet interagit pour implémenter un comportement. On peut décrire cette interaction de deux manières complémentaires : l'une est centrée sur des objets individuels (diagramme d'états-transitions) et l'autre sur une collection d'objets qui coopèrent (diagrammes d'interaction). Le diagramme d'interaction permet d'offrir une vue plus global du comportement d'un jeu d'objets.
On distingue : Le *diagramme de communication* qui met l'accent sur l'organisation structurelle des objets qui envoient et reçoivent des messages, et le *diagramme de séquence* qui met l'accent sur la chronologie de l'envoi des messages. Ils permettent d'établir un lien entre les diagrammes de cas

d'utilisation et les diagrammes de classes : ils montrent comment des objets communiquent pour réaliser une certaine fonctionnalité. Pour produire un diagramme d'interaction, il faut focaliser son attention sur un sous ensemble d'éléments du système et étudier leur façon d'interagir pour décrire un comportement particulier.

7.1 Diagramme de collaboration ou de communication

Une collaboration permet de décrire la mise en œuvre d'une fonctionnalité par un jeu de participants. Un rôle est la description d'un participant. Par exemple, pour implémenter un cas d'utilisation, il faut utiliser un ensemble de classes, et d'autres éléments, fonctionnant ensemble pour réaliser le comportement de ce cas d'utilisation. Cet ensemble d'éléments, comportant à la fois une structure statique et dynamique, est modélisé en UML par une collaboration. Ce diagramme est souvent utilisé pour illustrer un cas d'utilisation ou pour décrire une opération. Le diagramme de communication aide à valider les associations du diagramme de classe en les utilisant comme support de transmission des messages.

7.1.1 Les participants

Les participants sont représentés par des rectangles contenant une étiquette dont la syntaxe est : **[<nom_du_rôle>] : [<Nom_du_type>].** Au moins un des deux noms doit être spécifié dans l'étiquette, les deux points (:) sont obligatoire.

7.1.2 Les connecteurs

Les relations entre les représentants sont appelées connecteurs et se définissent par un trait plein.

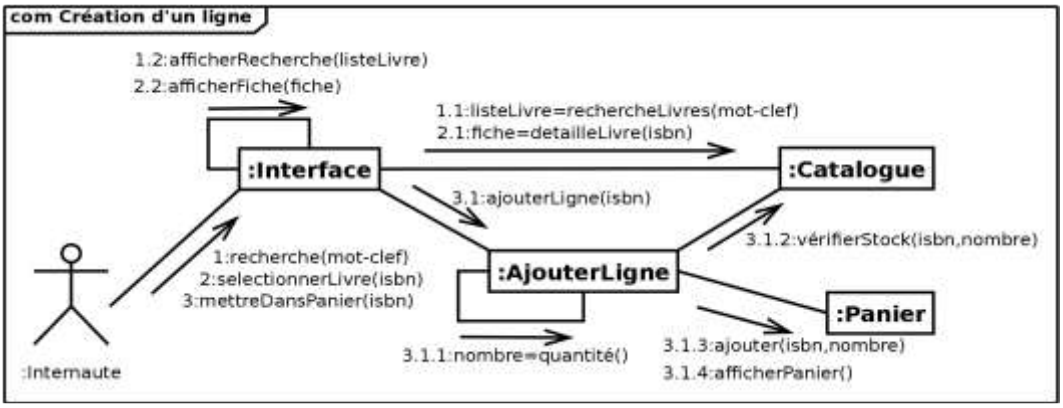
7.1.3 Les messages

Dans un diagramme de communication, les messages sont généralement ordonnés selon un numéro de séquence croissant. Un message est, habituellement, spécifié sous la forme suivante :

[['[<cond>]' [<séq>] [*[||] ['[<iter>']']] :] [<var> :=] <msg>([<par>])

<***cond***> est une condition sous forme d'expression booléenne entre crochets.
<***séq***> est le numéro de séquence du message. On numérote les messages par envoi et sous envoi désignés par des chiffres séparés par des points : ainsi l'envoi du message 1.4.4 est postérieur à celui du message 1.4.3, tous deux étant des conséquences (des sous-envois) de la réception d'un message 1.4. La simultanéité d'un envoi est désignée par une lettre : les messages 1.6a et 1.6b sont envoyés en même temps.
<***iter***> spécifie (en langage naturel, entre crochets) l'envoi séquentiel (ou en parallèle, avec ||) de plusieurs message. On peut omettre cette spécification et ne garder que le caractère * pour désigner un message récurrent envoyé un certain nombre de fois.
<***var***> est la valeur de retour du message, qui sera par exemple transmise en paramètre à un autre message.
<***msg***> est le nom du message.
<***par***> désigne les paramètres (optionnels) du message.

Le schéma ci-dessous donne le diagramme de communication illustrant la recherche puis l'ajout, dans son panier virtuel, d'un livre lors d'une commande sur Internet.



7.2 Diagramme de séquence

Les principales informations contenues dans un diagramme de séquence sont les messages échangés entre les lignes de vie, présentés dans un ordre chronologique. Ainsi, contrairement au diagramme de communication, le temps y est représenté explicitement par une dimension
(La dimension verticale) et s'écoule de haut en bas.

7.2.1 Représentation des lignes de vie

Une ligne de vie se représente par un rectangle, auquel est accrochée une ligne verticale pointillée, contenant une étiquette dont la syntaxe est : **[<nom_du_rôle>] : [<Nom_du_type>].**

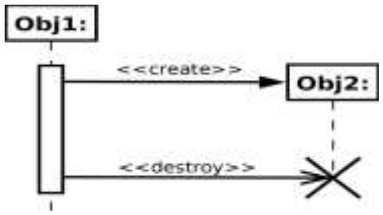
7.2.2 Représentation des messages

Un message définit une communication particulière entre des lignes de vie. Plusieurs types de messages existent, les plus communs sont :
– l'envoi d'un signal ;
– l'invocation d'une opération;
– la création ou la destruction d'une instance.

Messages asynchrones : Ils n'attendent pas de réponse et ne bloquent pas l'émetteur qui ne sait pas si le message arrivera à destination, le cas échéant quand il arrivera et s'il serra traité par le destinataire. Un signal est, par définition, un message asynchrone.
Graphiquement, un message asynchrone se représente par une flèche en traits pleins et à l'extrémité allant de la ligne de vie d'un objet vers celle d'un autre.

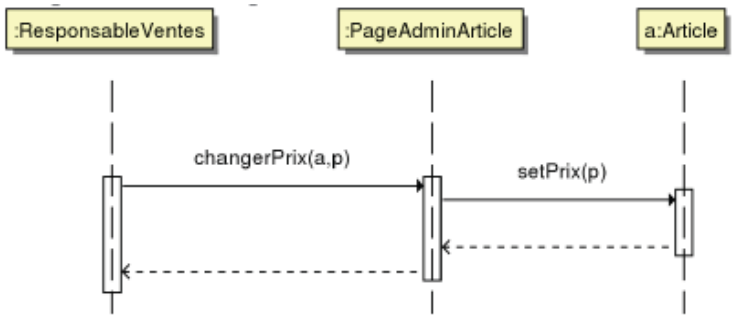
Messages synchrones : l'émetteur reste bloqué le temps que dure le traitement du message.
Graphiquement, un message synchrone se représente par une flèche en traits pleins et à l'extrémité pleine partant de la ligne de vie d'un objet vers celle d'un autre. Ce message peut être suivi d'une réponse qui se représente par une flèche en pointillé.

Messages de création et destruction d'instance : La création d'un objet est matérialisée par une flèche qui pointe sur le sommet d'une ligne de vie. La destruction d'un objet est matérialisée par une croix qui marque la fin de la ligne de vie de l'objet. La destruction d'un objet n'est pas nécessairement consécutive à la réception d'un message.



Événements et messages: UML permet de séparer clairement l'envoi du message, sa réception, ainsi que le début de l'exécution de la réaction et sa fin.

Syntaxe des messages et des réponses : Dans la plupart des cas, la réception d'un message est suivie de l'exécution d'une méthode d'une classe. Cette méthode peut recevoir des arguments et la syntaxe des messages permet de transmettre ces arguments.
La syntaxe de réponse à un message est la suivante :
[<attribut> =] message [: <valeur_de_retour>]
Où message représente le message d'envoi.



Exécution de méthode et objet actif : Un objet actif initie et contrôle le flux d'activités. Graphiquement, la ligne pointillée vertical d'un objet actif est remplacée par un double trait vertical. Un objet passif, au contraire, a besoin qu'on lui donne le flux d'activité pour pouvoir exécuter une méthode. La spécification de l'exécution d'une réaction sur un objet passif se représente par un rectangle blanc ou gris placé sur la ligne de vie en pointillée. Le rectangle peut éventuellement porter un label.
Les exécutions simultanées sur une même ligne de vie sont représentées par un rectangle chevauchant.

7.2.3 Fragments d'interaction combinés

Un fragment combiné représente des articulations d'interactions, défini par un opérateur et des opérandes. L'opérateur conditionne la signification du fragment combine. Un fragment est représenté dans un rectangle dont le coin supérieur gauche contient un pentagone. Dans le pentagone figure le type de la combinaison, appelé opérateur d'interaction. Les opérandes d'un opérateur d'interaction sont séparés par une ligne pointillée. Les conditions de choix des opérandes sont données par des expressions booléennes entre crochets.
Il existe 12 operateurs d'interaction, parmi lesquels les opérateurs de choix et de boucle : **alternative**, **option**, **loop,** et **break.**

Opérateur alt

L'opérateur alternative, ou ***alt***, est un opérateur conditionnel possédant plusieurs opérandes. C'est un peu l'équivalent d'une exécution à choix multiple (condition switch en C++). Chaque opérande détient une condition de garde. L'absence de condition de garde implique une condition vraie. La condition ***else*** est vraie si aucune autre condition n'est vraie. Exactement un opérande dont la condition est vraie est exécuté.

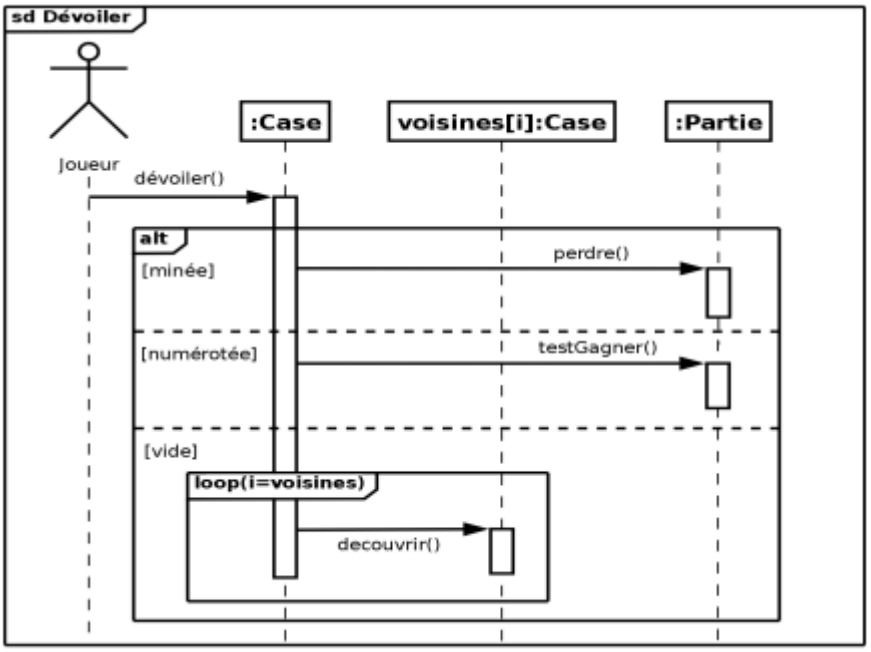
Opérateurs opt

L'opérateur option, ou ***opt***, comporte un opérande et une condition de garde associée.
Le sous-fragment s'exécute si la condition de garde est vraie et ne s'exécute pas dans le cas contraire.

Opérateur loop

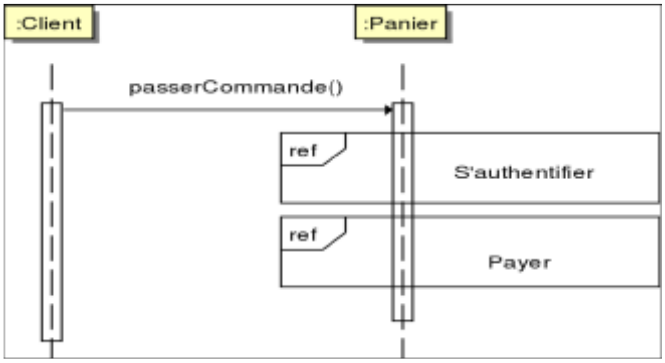
Un fragment combiné de type ***loop*** possède un sous-fragment et spécifie un compte minimum et maximum (boucle) ainsi qu'une condition de garde.
La syntaxe de la boucle est la suivante : **loop['('<minInt> [',' <maxInt>] ')']**
La condition de garde est placée entre crochets sur la ligne de vie. La boucle est répétée au moins ***minInt*** fois avant qu'une éventuelle condition de garde booléenne ne soit testée. Tant que la condition est vraie, la boucle continue, au plus ***maxInt*** fois.

Exemple



Opérateur ref

L'opérateur ***ref***, permet de réutiliser une interaction. Ceci consiste à placer un fragment portant la référence « ref » là où l'interaction est utile.



8. Diagramme de composants

La notion de classe, de part sa faible granularité et ses connexions figées (les associations avec les autres classes matérialisent des liens structurels), ne constitue pas une réponse adaptée à la problématique de la réutilisation.

8.1 Notion de composant

Un composant est une unité autonome représentée par un classeur structuré, stéréotypé «Component», comportant une ou plusieurs interfaces requises ou offertes. Son comportement interne, généralement réalisé par un ensemble de classes, est totalement masqué : seules ses interfaces sont visibles.

Le diagramme de composant représente l'architecture logicielle du système. Ils permettent de structurer une architecture logicielle à un niveau de granularité moindre que les classes. Les composants peuvent contenir des classes. Ils permettent aussi de spécifier l'intégration de briques logicielles tierces (composants EJB, CORBA, .Net, WSDL, etc..)

8.2 Notion de port

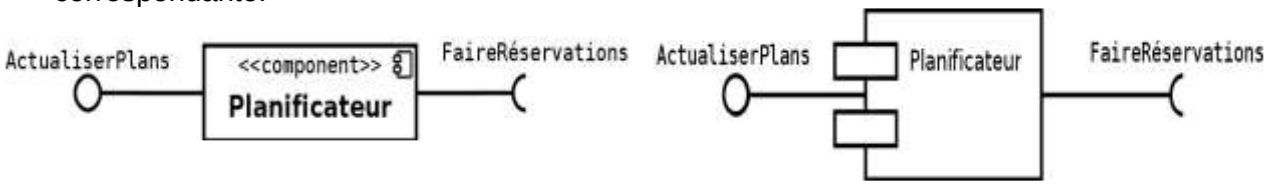
Un port est un point de connexion entre un classeur et son environnement. Graphiquement, un port est représenté par un petit carré à cheval sur la bordure du contour du classeur. On peut faire figurer le nom du port à proximité de sa représentation.

Généralement, un port est associé à une interface requise ou offerte. Parfois, il est relié directement à un autre port. L'utilisation des ports permet de modifier la structure interne d'un classeur sans affecter les clients externes.

8.3 Représentation d'un diagramme de composants

La relation de dépendance est utilisée dans les diagrammes de composants pour indiquer qu'un élément de l'implémentation d'un composant fait appel aux services offerts par les éléments d'implémentation d'un autre composant.

Lorsqu'un composant utilise l'interface d'un autre composant, on peut utiliser la représentation ci-dessous, en imbriquant le demi-cercle d'une interface requise dans le cercle de l'interface offerte correspondante.



9. Diagramme de déploiement

Un système doit s'exécuter sur des ressources matérielles dans un environnement matériel particulier. UML permet de représenter un environnement d'exécution ainsi que des ressources physiques (avec les parties du système qui s'y exécutent) grâce aux diagrammes de déploiement.

Un nœud est une ressource sur laquelle des artefacts peuvent être déployés pour être exécutés.

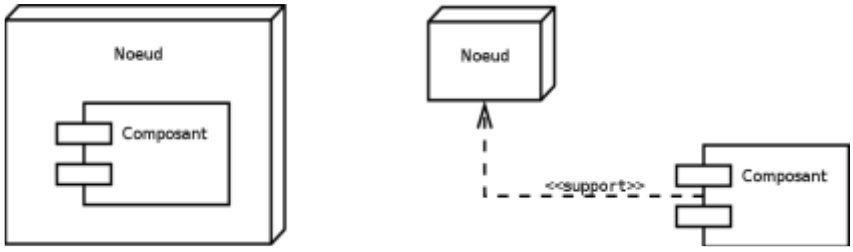
Un diagramme de déploiement décrit la disposition physique des ressources matérielles qui composent le système et montre la répartition des composants sur ces matériels. Chaque ressource étant matérialisée par un nœud, le diagramme de déploiement précise comment les composants sont répartis sur les nœuds et quelles sont les connexions entre les composants ou les nœuds.

9.1 Représentation des nœuds

Chaque ressource est matérialisée par un nœud représenté par un cube comportant un nom. Un nœud est un classeur et peut posséder des attributs (quantité de mémoire, vitesse du processeur, etc.).

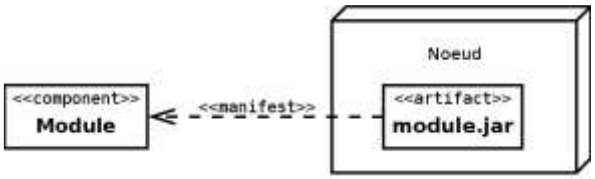
Un nœud particulier est une instance de nœud

Une fois les nœuds définis, on doit leur affecter les composants. Pour montrer qu'un composant est affecté à un nœud, il faut soit placer le composant dans le nœud, soit les relier par une relation de dépendance stéréotypée «support» orientée du composant vers le nœud.



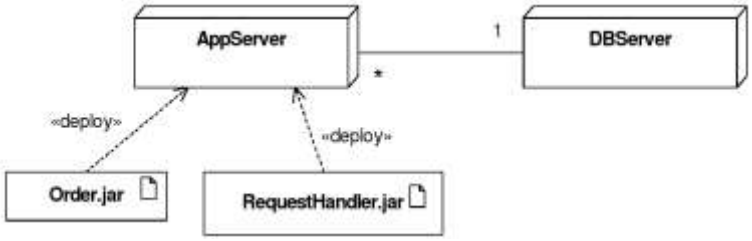
9.2 Notion d'artefact (artifact)

Un artefact correspond à un élément concret existant dans le monde réel (document, exécutable, fichier, tables de bases de données, script, etc.). Il se représente comme un classeur par un rectangle contenant le mot-clé «***artifact***» suivi du nom de l'artefact.
L'implémentation des modèles (classes, etc.) se fait sous la forme de jeu d'artefacts. On dit qu'un artefact peut manifester, c'est-à-dire implémenter, un ensemble d'éléments de modèle. On appelle manifestation la relation entre un élément de modèle et l'artefact qui l'implémente. Graphiquement, une manifestation se représente par une relation de dépendance stéréotypée «***manifest***»



Une instance d'un artefact se déploie sur une instance de nœud. Graphiquement, on utilise une relation de dépendance stéréotypée «***deploy***» pointant vers le nœud en question. L'artefact peut aussi être inclus directement dans le cube représentant le nœud. En toute rigueur, seul des artefacts doivent être déployés sur des nœuds. Un composant doit donc être manifesté par un artefact qui, lui-même, peut être déployé sur un nœud.

9.3 Exemple d'un diagramme de déploiement



Bibliographie

- **UML 2, Laurent AUDIBERT,** Département d'Informatique, IUT de villetaneuse, Edition 2007-2008
- **Object Oriented Analysis and Design, B. LAVANYA,** Dept of computer Science, University of Madras, India.
- **Modélisation objet avec UML, Pierre-Alain Muller , Nathalie Gaertner ,** Eyrolles.

Quelques Outils

Voici quelques outils utilisés pour construire des diagrammes de UML

- **Argo/UML** (http://argouml.tigris.org/index.html) de l'université de Californie UCI (www.ics.uci.edu/pub/arch/uml)
- **Prosa/om** (www.prosa.fi/prosa.html) d'Insoft Oy
- **Objecteering** (http://www.objecteering.com/) de Softeam (www.softeam.fr)
- **Paradigm Plus** (www.platinum.com/products/appdev/pplus_ps.htm) de Computer Associates
- **Rhapsody** (www.ilogix.com/fs_prod.htm) d'I-Logix (www.ilogix.com)
- **Rose 2000** (http://www.rosearchitect.com/) de Rational Software Corporation (www.rational.com)
- **StP/UML** (www.aonix.com/Products/SMS/core7.1.html) d'Aonix (www.aonix.fr)
- **Visual UML** (www.visualuml.com/products.htm) de Visual Object Modelers