Modélisation Objet : Le formalisme UML

Ecole des Mines d'Albi-Carmaux	**Option GSI**

TABLE DES MATIERES

1. Préambule

Avant d'axer ce document sur la présentation globale du formalisme UML, il est intéressant de s'intéresser d'une part à la notion de ***Modélisation***, et d'autre part aux concepts inhérents à la vision ***Orientée Objet***. Le langage UML se trouve en effet à la croisée de ces deux chemins.

1.1 La Modélisation

La modélisation est une notion indissociable du fonctionnement de l'homme. Notre appréhension du monde est entièrement basée sur cette idée : nos objectifs naturels de compréhension et d'anticipation nous amènent naturellement à construire une représentation mentale et abstraite de l'ensemble des éléments qui composent notre environnement. Ce processus d'assimilation et de représentation peut se résumer ainsi :

Modéliser, c'est représenter une entité connue ou inconnue selon des concepts et un vocabulaire connus

Dessiner un plan, prévoir les réactions des gens, faire un portrait, manœuvrer un véhicule, faire un croquis pour expliquer une idée sont des activités ayant directement trait avec la modélisation.

En termes de vision industrielle, la modélisation constitue un outil essentiel dans les domaines de l'analyse (définition du produit) de la conception (description du produit à réaliser) et du développement (réalisation du produit). De manière plus générale, la modélisation intervient tout au long de la gestion de projet, comme outil de description et de communication entre les acteurs. Le document résultant de l'activité de modélisation constitue le modèle. Un modèle permet de décrire et de figer la réponse que l'on pense apporter aux exigences :

Un modèle est une représentation abstraite et non-ambiguë du sujet dans un langage donné

Une maquette, un plan, une photo, des mensurations, un émulateur, un organigramme sont des modèles.

Ses fonctions premières d'outil de description et de communication doivent faire tendre un modèle à être compréhensif, rigoureux, évolutif, complet, convivial, raffinable… Néanmoins, un modèle n'existe que parce qu'il résulte d'un point de vue. En effet, un modèle est toujours une vision simplifiée et formalisée d'un sujet. Cette simplification et cette formalisation résultent du rôle qu'on attend du modèle : une photo d'une maison ou ses plans de masses sont deux modèles d'un même sujet destinés à des usages différents et résultant par conséquent d'axes d'observation différents (un axe purement visuel et un axe plus structurel).

Le point de vue de modélisation est dicté par l'usage qui va être fait du modèle

Considérer un sujet selon une vue de profil, selon son aspect physique, selon ses réactions thermiques, selon son organisation, selon son comportement, c'est se placer selon différents points de vue de modélisation.

La modélisation est finalement une activité de projection :

- d'un sujet réel (existant ou non),
- sur le plan d'un langage de modélisation,
- selon un angle de considération résultant de l'utilisation attendue du modèle,
- pour obtenir une vision abstraite, partielle et formalisée du sujet (le modèle).

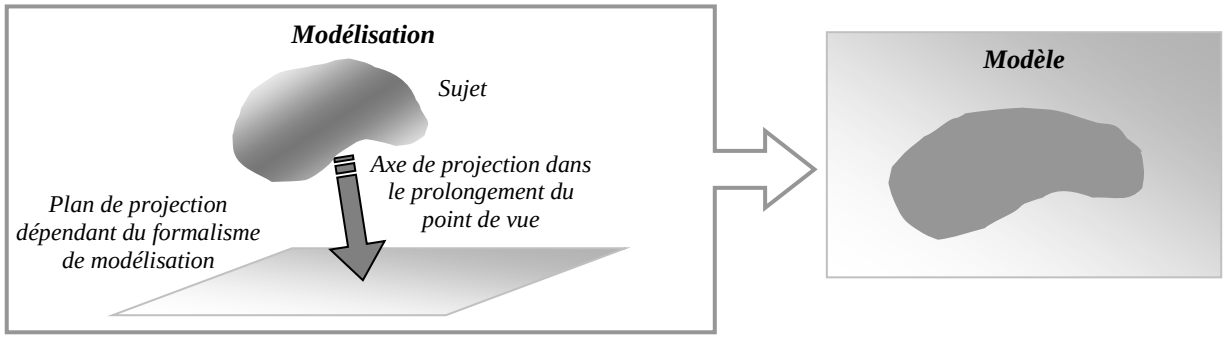


Figure n°1 : *Principe de Modélisation*

1.2 Les concepts « objet »

Comme tout domaine de réflexion et de travail, le domaine « orienté-objet » relève de concepts, d'outils, de règles et d'éléments spécifiques. La légitimité de toute ambition d'évoluer ou d'étudier dans ce type d'espace passe par l'adoption et la compréhension de l'ensemble de ces notions propres au domaine « orienté-objet ».

1.2.1 L'objet

Les activités « orientées-objet » reposent, comme leur nom l'indique, sur le concept d'objet. Dans ce contexte, l'objet constitue la brique élémentaire à partir de laquelle ces activités se construisent. Dans une activité « orientée-objet », tout est objet. La bonne manipulation de ces briques élémentaires repose en partie sur la connaissance de leur nature, de leurs caractéristiques, de leurs aptitudes et de leur structure.

Un objet est donc une entité identifiée qui possède un comportement propre (des fonctions spécifiques) dépendant de son état interne et avec laquelle on peut interagir (échange de messages)

Cette définition informelle pose plusieurs questions :

1. ***entité identifiée*** : Comment identifie-t-on et délimite-t-on les objets ?
2. ***comportement propre*** : Qu'est-ce qui constitue pratiquement le comportement de l'objet ?
3. ***état interne*** : Qu'est-ce qui caractérise l'état d'un objet ?
4. ***interaction*** : Par quel procédé interagit-on avec un objet ?

Une partie des réponses à ses interrogations se trouvent dans la représentation structurelle des objets :

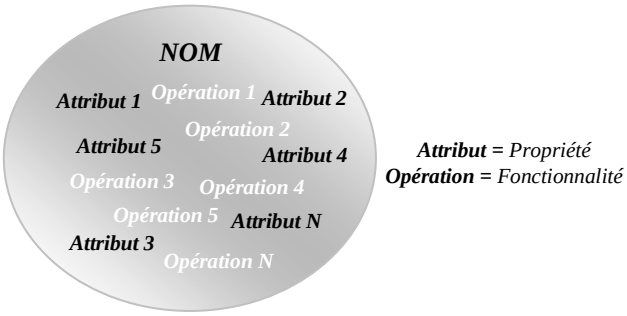


Figure n°2 : ***Structure d'un objet***

1. Si la question de la désignation des objets est évidemment résolue par la présence d'une dénomination de l'objet (le *Nom* de l'objet, qui traduit la continuité de son existence), le problème de la délimitation des objets demeure en suspens. Il s'agit d'une liberté laissée à la personne en charge de l'activité « orientée-objet » et en tant que telle, cette latitude permet d'une part d'exprimer une approche plus personnelle du sujet et d'autre part, elle accroît l'incertitude dans la façon d'aborder ce sujet. On peut néanmoins indiquer que les critères qui guident généralement la délimitation des objets relèvent du point de vue de description adopté (au sens où le point de vue a été abordé dans la partie *Modélisation*). Ainsi, pour décrire un ordinateur on pourra dire qu'il se compose d'une unité centrale, d'un écran, d'un clavier, d'une connectique (chacun de ces éléments étant lui-même décomposable) ou encore d'un système d'exploitation, d'une interface homme-machine, d'un logiciel de traitement de texte, de jeux, d'un navigateur (chacun de ces éléments étant également décomposable).

2. Le comportement d'un objet est déterminé par les aptitudes dont il dispose (i.e. ses opérations). Ces fonctionnalités pourront être « déclenchées » en interne ou appelées de l'extérieur (par d'autres objets).

3. L'état interne d'un objet est caractérisé par le vecteur de ses attributs. L'ensemble des valeurs des attributs d'un objet permet de situer l'objet dans l'ensemble des états auxquels il peut accéder.

4. Concernant l'interaction avec l'objet, elle va porter essentiellement sur l'appel d'opérations depuis l'extérieur ou plus rarement sur l'intervention directe sur les attributs (et donc l'état) de l'objet.

1.2.2 La Classe

Du point de vue d'une activité « orientée-objet », les objets résultent de l'instanciation de classes (où l'instanciation est l'action de créer une entité réelle à partir d'un concept théorique). Une classe constitue donc à la fois le domaine de définition et la description générique d'un ensemble d'objets. Chaque objet appartient ainsi à une unique classe.

Une classe est une description générique d'une collection d'objets ayant une structure similaire

On peut illustrer les relations entre classes et objets à l'aide de la figure suivante :

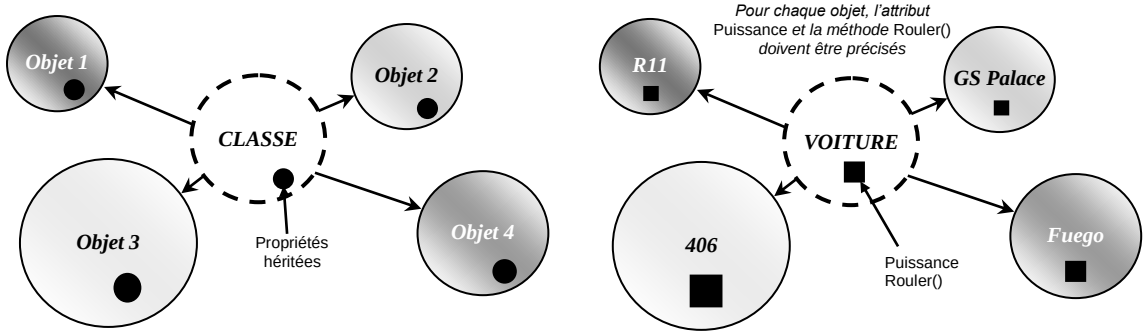


Figure n°3 : ***Objets et classes (vue de principe et exemple)***

1.2.3 L'Héritage

Une notion importante du monde de l'objet est le concept d'héritage. Cette idée permet de factoriser les éléments communs d'un ensemble de classes dans une classe plus générale. On dira par exemple que le vin, l'eau, la bière ou la vodka-orange héritent des propriétés des liquides ; et ce même s'ils disposent en plus de propriétés propres.

L'héritage permet de décrire les notions de spécialisation ou de généralisation

Il faut également noter que l'héritage peut être « multiple » (i.e. une classe hérite de plusieurs autres). Dans ce cas, le processus d'héritage n'effectue pas une union des propriétés héritées mais une somme, ce qui peut induire des conflits et des collisions de dénominations.

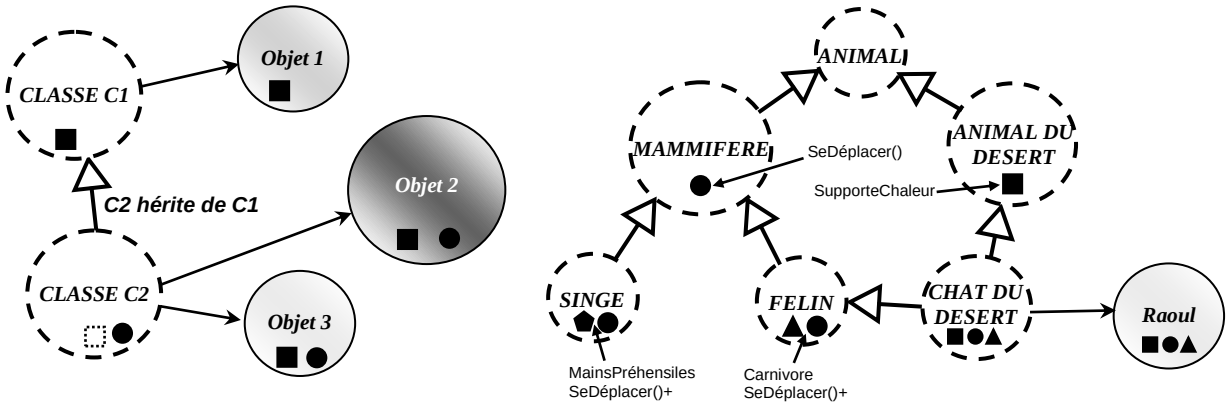


Figure n°4 : ***héritage et héritage multiple (vue de principe et exemple)***

Forte de ces considérations théoriques, la suite du document va être consacrée à l'étude et à la description du formalisme UML en tant que langage de modélisation orientée-objet.

2. Présentation d'UML

La modélisation et les formalismes existent depuis toujours (croquis, plans, maquettes…). La modélisation orientée-objet est apparue durant les années 70 et 80 suite à l'émergence de la programmation orientée-objet. En effet, il était indispensable d'associer à ces nouvelles techniques de développement de nouveaux outils de description adaptés. Entre 1989 et 1994, le nombre de méthodes et de langages de modélisation orientés objet est passé de moins de 10 à plus de 50. Parmi cet assortiment d'outils, 3 émergeaient plus particulièrement :

- La méthode Booch (du nom de Grady Booch son concepteur),
- La méthode OOSE (pour Object-Oriented Software Engineering crée par Ivar Jacobson),
- La méthode OMT (pour Object Modeling Technique conçue par James Rumbaugh).

Face à l'évolution convergente de ces méthodes, leurs auteurs (Booch, Rumbaugh et Jacobson) ont décidé de s'associer au sein de l'entreprise Rational pour créer le langage UML (Unified Modeling Language) ; l'objectif de cet outil de modélisation étant d'unir les apports de leurs travaux respectifs. Le formalisme UML est « stabilisé » depuis 1997 (avec la mouture UML 1.1, ce qui n'empêche pas les évolutions puisque actuellement la toute récente version UML 2.0 fait référence) et est suivi par l'OMG (Object Modeling Group, autorité de normalisation liée à l'objet).

UML ne constitue aucunement une méthode de conception mais bien un langage de modélisation

UML a été conçu selon les objectifs suivants :

- Objectif longitudinal : accompagner l'ensemble du cycle de conception,
- Objectif vertical : être adapté à toutes les échelles de modélisation (du plus haut niveau de considération à la prise en compte des détails les plus fins),
- Objectif tranversal : être suffisamment compréhensible et convivial pour être facilement accessible à l'homme et être dans le même temps suffisamment formel et rigoureux pour être adapté au traitement informatique.

Comme tout langage évolué, UML propose :

- Un vocabulaire : *un ensemble de symboles graphiques*
  Ce sont les éléments de modélisation propre à UML (classes, états, relations…) et leur sémantique (la signification de chaque symbole).
- Une grammaire : *un ensemble des règles d'agencement de ces symboles*
  Ce sont les lois de construction en UML (un état doit être relié à un autre état par une transition, les héritages ne doivent pas présenter de cycles, l'accès aux propriétés d'une classe par une autre classe dépend des relations qui les associent…).
- Une liberté d'expression : *un ensemble d'alternatives dépendantes du point de vue*
  Ce sont les différents diagrammes proposés par UML (ou combinaisons de diagrammes) qui peuvent être choisis selon l'angle de description (et donc de lecture) que l'on veut donner au modèle (vue structurelle : diagrammes de classes et d'objets, vue comportementale : diagrammes de séquences et d'états-transitions, vue architecturale : diagrammes de composants et de déploiement…).

Le formalisme UML peut être vu comme une boîte à outil offrant au concepteur un panel de diagrammes parmi lesquels il va choisir ceux qui sont adaptés à la finalité du modèle qu'il veut construire. Ces diagrammes sont les suivants :

- Diagramme de cas d'utilisation : *ensemble des contextes de mise en œuvre du sujet modélisé*
  Ce diagramme permet de représenter les fonctionnalités principales du système modélisé, considérées du point de vue de l'utilisateur.
- Diagrammes de classes : *description de la structure statique du sujet modélisé*
  Ce diagramme permet de décrire tout ou partie du système modélisé, d'une manière abstraite, en terme de classes, de structure et d'associations.
- Diagrammes d'objets : *description de la structure statique d'une instance du sujet modélisé*

Ce diagramme permet de décrire des exemples de configuration de tout ou partie du système modélisé, en terme d'objets, de valeurs et de liens.

- Diagrammes de séquence : *représentation temporelle du comportement du sujet modélisé*
  Ce diagramme permet de décrire des scénarios au travers du séquencement des interactions entre les acteurs et composants du modèle (description chronologique).
- Diagrammes de collaboration : *représentation spatiale et temporelle du comportement du sujet*
  Ce diagramme permet de décrire l'agencement des chaînes fonctionnelles impliquées dans les scénarios. Ce diagramme est très proche du diagramme de séquence qu'il complète de notions de répartition physique des objets.
- Diagrammes d'états-transitions : *description du comportement spécifique d'une classe*
  Ce diagramme utilise les automates à états finis (statecharts) pour décrire l'activité propre d'une classe en terme d'états, d'évènements, d'actions, de conditions et de transitions permettant le changement d'état.
- Diagrammes d'activité : *description d'opérations en terme de succession d'actions*
  Ce diagramme permet de décrire le déroulement des fonctions et des processus par le biais de la représentation de l'organisation des sous-taches et des flux entre les objets.
- Diagrammes de composants : *projection du diagramme d'objets sur le plan physico-réel*
  Ce diagramme permet d'effectuer l'allocation des composants depuis une vue logique (diagramme de classes ou d'objets) vers la représentation et l'organisation des composants physiques qui implémenteront réellement le sujet et de leurs dépendances.
- Diagrammes de déploiement : *description de la mise en place de l'architecture du sujet modélisé*
  Ce diagramme permet de représenter le déploiement des composants sur les dispositifs matériels.

Cette présentation très théorique des éléments constitutifs du formalisme UML n'est destinée qu'à fournir une vision globale du langage et de son organisation. Cette approche « haut-niveau » sera reprise à l'issue de la présentation détaillée des différents diagrammes mais elle permet d'ores et déjà de positionner les composants qui vont être présentés (symboles et conventions graphiques) dans l'organisation globale du langage UML.

Concernant la structure du formalisme UML, on peut en donner la première vision suivante :

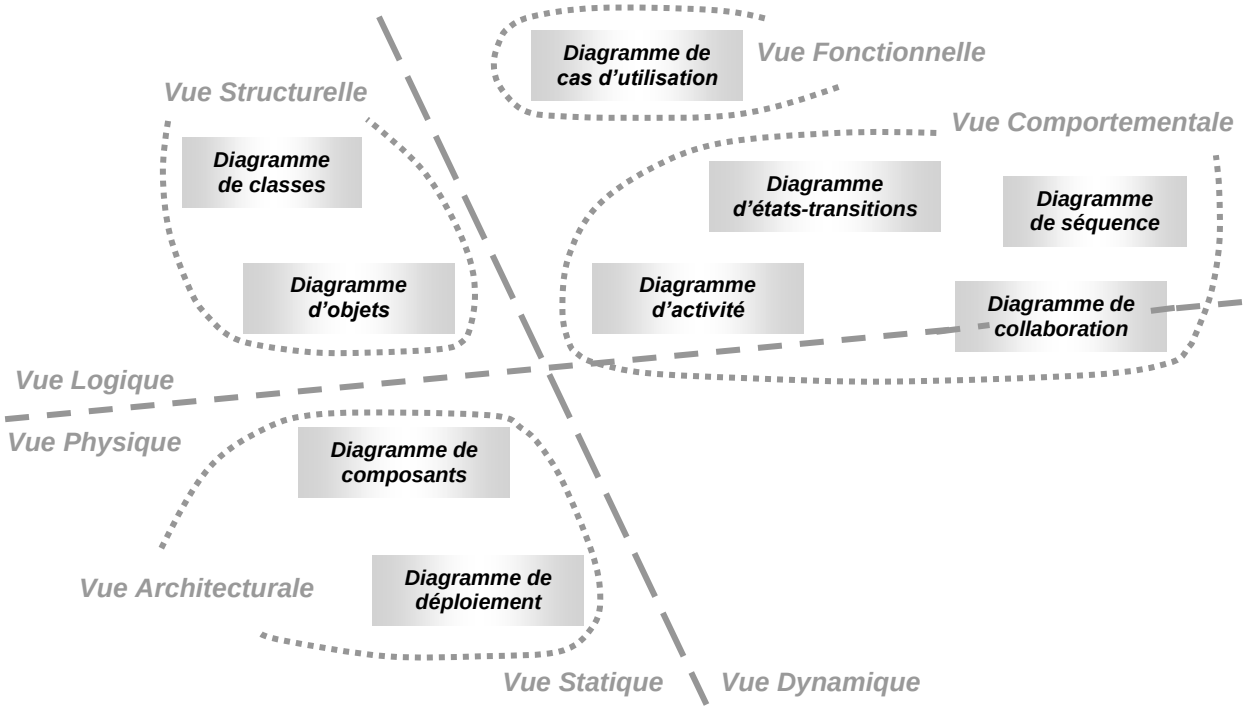


Figure n°5 : ***Une répartition des diagrammes UML***

Ces différentes « classifications » de l'espace des diagrammes UML (logique/physique, statique/dynamique, et fonctionnel/structurel/comportemental/architectural) permettent de déterminer les jeux de diagrammes à choisir en fonction du point de vue adopté. Le point de vue de modélisation tel qu'il a été présenté en préambule de ce

document est principalement supporté en UML par le choix des vues du sujet qui doivent être décrites et donc par le choix des diagrammes au sein de ses vues. Afin de mieux appréhender la signification et l'intérêt de ces différentes vues on peut en donner les définitions partielles suivantes :

- **Vue logique :** facette théorique du sujet. Cette vue, déconnectée de toute considération matérielle, fournit une description basée sur les rôles, les responsabilités et les relations associées.
- **Vue physique :** aspect organique et matériel du sujet. Cette vue fournit une description basée sur les composants physiques qui implémentent réellement le sujet.
- **Vue statique :** facette relative à l'agencement des composants. Cette vue s'apparente à une description figée du sujet fournissant une photographie de sa physionomie.
- **Vue dynamique :** facette liée à l'activité du sujet. Cette vue fournit une représentation vivante du sujet et décrit ses comportements et ses réactions.
- **Vue fonctionnelle :** facette relative aux contextes d'activité du sujet. Cette vue particulière relève d'une description externe du sujet en tant que fournisseurs de services à l'environnement et aux utilisateurs.
- **Vue structurelle :** aspect organisationnel logique du sujet. Cette vue décrit la structuration des différents constituants fonctionnels du système.
- **Vue comportementale :** facette liée au fonctionnement du sujet et de ses composants. Cette vue fournit une description des actions, processus et modes de fonctionnement sur divers niveaux de granularité.
- **Vue architecturale :** aspect organisationnel physique du sujet. Cette vue décrit la structuration des différents composants matériels qui constituent organiquement le sujet

Toujours d'un point de vue « vision générale d'UML », ce formalisme de modélisation comprend en plus des 9 diagrammes évoqués un langage de description de contraintes. Ce langage nommé OCL (pour Object Constraints Language) permet d'ajouter une composante sémantique plus riche à certains diagrammes. Il faut comprendre que le graphe obtenu lors de la modélisation d'une facette d'un sujet selon un diagramme donné peut parfois manquer d'une nuance précise qu'on souhaiterait voir apparaître. Il est alors toujours possible d'associer une note au diagramme qui explicite cette notion (et c'est souvent ce qui est fait). Cependant, si l'on revient sur l'objectif tranversal évoqué en début de cette partie, il semble clair que l'interprétation par une machine de ces notes en langage naturel (par exemple lors de la génération de code à partir de modèle UML) est totalement utopique (aujourd'hui au moins). OCL pallie ce manquement en offrant un langage de contraintes formalisé qui pourra être interprété par un ordinateur (comme un langage de programmation). La description du langage OCL et de ses applications ne fait pas partie des objectifs de ce document.

La présentation détaillée d'UML qui va suivre s'articule autour de la découpe selon les différentes vues fonctionnelle, structurelle, comportementale et architecturale. Ce sont donc les diagrammes de cas d'utilisation qui sont présentées dans un premier temps (vue fonctionnelle) avant qu'on ne s'intéresse plus spécifiquement aux diagrammes de classes et d'objets (en tant que vue structurelle), puis au groupe comportementale avec les diagrammes de séquence, de collaborations, d'activité et d'états-transitions avant de terminer par les diagrammes de composants et de déploiement pour la vue architecturale.

3. Modèle Fonctionnel

Les diagrammes de cas d'utilisation composent la vue fonctionnelle du formalisme UML. Ce sont des diagrammes cruciaux vis à vis de la cohérence et de la pertinence du modèle en cours de description. En outre, leur originalité et leur relative indépendance parmi les diagrammes UML leur confèrent une certaine spécificité qui nécessite une étude distincte.

3.1 Concepts de base

Un diagramme de cas d'utilisation fait appel à deux concepts :

- L'acteur : *entité extérieure en interaction avec le système*
- Le cas d'utilisation : *une manière d'utiliser le système*

3.1.1 Acteur

Les acteurs représentent les éléments externes aux délimitations du système qui sont amenés à interagir avec celui-ci. Ils comprennent aussi bien les acteurs humains (utilisateurs) que les acteurs d'autre type (machine, autre système, tout élément de l'environnement).

Les acteurs définissent les partenaires extérieurs interagissant avec le sujet

Les acteurs jouent un rôle important dans le travail de délimitation des frontière du sujet modélisé, ils permettent de positionner le système dans son environnement en représentant son entourage actif .

Un acteur se représente graphiquement sous la forme d'un personnage accompagné de sa désignation (visuel créé avec *Objecteering UML Modeler*) :

Figure n°6 : ***Aspect visuel d'un acteur***

3.1.2 Cas d'utilisation

Les cas d'utilisation répondent à la question « pour quoi le système va-t-il être utilisé ? » (Quels services va-t-il rendre, vu de l'extérieur ?) et décrivent les rôles du sujet. Ces symboles représentent le point de vue des utilisateurs du système.

Les cas d'utilisation décrivent les fonctionnalités externes du sujet

Il faut également noter que les cas d'utilisation couvrent implicitement l'ensemble des scénarios opérationnels associés au sujet (vu de l'extérieur en tant qu'entité globale).

Un cas d'utilisation se représente graphiquement sous la forme d'une figure ovale (éventuellement incluse dans un cadre structurant) complétée de la fonctionnalité représentée (visuel créé avec *Objecteering UML Modeler*) :

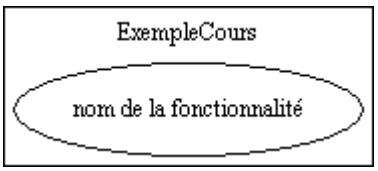


Figure n°7 : ***Aspect visuel d'un cas d'utilisation***

3.2 Diagramme de cas d'utilisation (Use-Case Diagram)

L'objet d'un diagramme de cas d'utilisation est de positionner le système dans son contexte opérationnel et de décrire les « nouvelles règles de fonctionnement » issues de sa mise en œuvre. Ce type de diagrammes se consacre à la description des interactions du sujet avec les acteurs qui l'entourent.

Les diagrammes de cas d'utilisation présentent l'intégration du sujet dans son environnement en s'intéressant à sa description fonctionnelle

Le formalisme associé est relativement simple afin de faciliter l'expression du besoin et le dialogue avec le maître d'ouvrage. La construction de diagrammes de cas d'utilisation est une excellente façon d'aborder la description et la conception d'un système.

Les relations admises au sein d'un diagramme de cas d'utilisation sont les suivantes :

- Les relations de type *déclenchement* qui relient un acteur à un cas d'utilisation : ces relations sont représentées sous la forme d'un trait simple et traduisent l'implication d'un acteur vis à vis d'un cas d'utilisation (du type *l'acteur déclenche le cas d'utilisation*),
- Les relations de type *inclusion* ou *utilisation* qui relient deux cas d'utilisation : ces relations sont représentées sous la forme d'une flèche pointillée accompagnée du terme « *include* » et traduisent le fait que le cas d'origine contient obligatoirement le cas destination,
- Les relations de type *extension* ou *raffinement* qui relient deux cas d'utilisation : ces relations sont représentées sous la forme d'une flèche pointillée accompagnée du terme « *extend* » et traduisent le fait que le cas d'origine peut optionnellement être rajouté au cas destination,
- Les relations de type *généralisation* ou *spécialisation* qui relient deux cas d'utilisation : ces relations sont représentées sous la forme d'une flèche pleine à pointe triangulaire vide et traduisent le fait que le cas d'origine peut remplacer le cas destination.

Pour illustrer ces concepts relatifs à la construction d'un diagramme de cas d'utilisation, on peut s'intéresser à la description des fonctionnalités d'un système de gestion de commande en ligne pour un organisme de VPC (visuel créé avec *Objecteering UML Modeler*) :

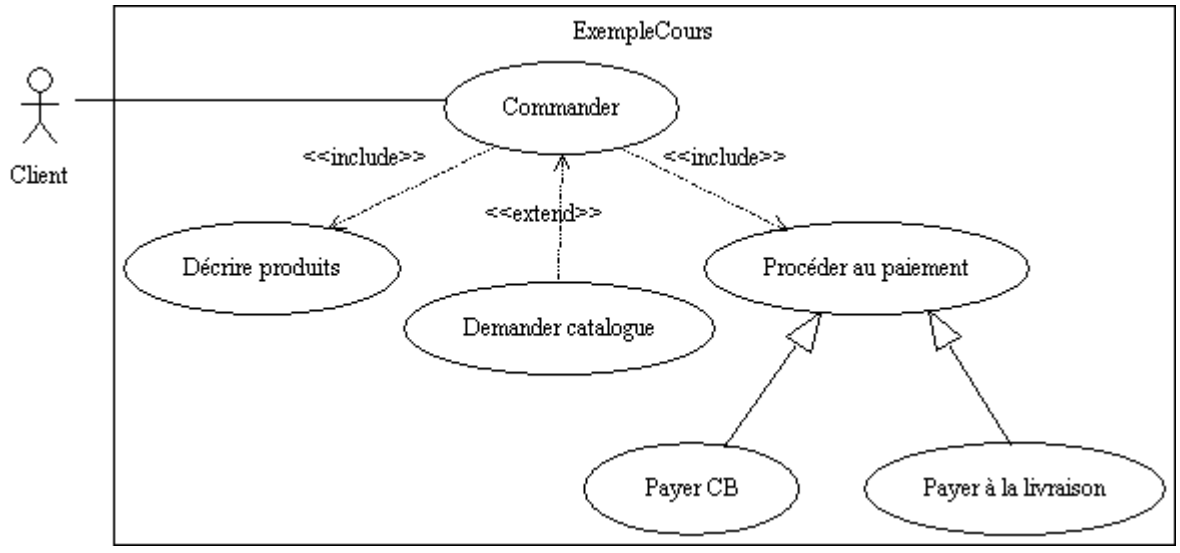


Figure n°8 : ***Exemple de diagramme de cas d'utilisation***

4. Modèle Structurel

Les diagrammes de classes et d'objets composent la vue structurelle du formalisme UML. Le diagramme de classes décrit de manière abstraite une vision générique du sujet alors que le diagramme d'objets décrit un exemple de configuration du système.

4.1 Diagramme de classes (Class Diagram)

Un diagramme de classe décrit l'agencement de classes connectées par des relations de différents types. Ce type de diagramme fait partie de la vue structurelle du sujet et participe à sa description logique et statique.

Les diagrammes de classes décrivent la structure logique du sujet sous une vision orienté-objet

Une classe décrit un groupe d'objets ayant la même structure (même ensemble d'attributs) et le même comportement (mêmes opérations). Les relations signifient la sémantique de l'organisation des classes.

4.1.1 Classe

Le formalisme UML décrit une classe en utilisant la symbolique suivante (visuel créé avec *Objecteering UML Modeler*) :

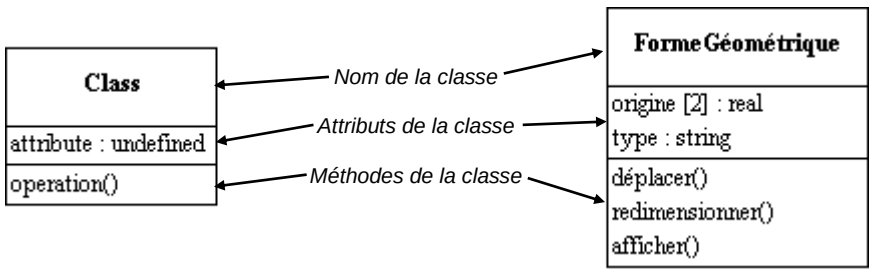


Figure n°9 : ***Représentation graphique d'une classe (vue de principe et exemple)***

Le nom d'une classe est unique. Il est dit « simple » lorsqu'il est de la forme *NomClasse* et « complet » lorsqu'il est de la forme *NomPackage::NomClasse* (où *NomPackage* désigne l'ensemble de classes dans lequel la classe *NomClasse* est incluse).

Un attribut représente une propriété de la classe modélisée (il sera donc commun à tous les objets de la classe et pour chaque objet il sera valué). La notation complète d'un attribut est la suivante :

[*visibilité***]** *nomAttribut* **[** *[multiplicité]* **] [***:type***] [***=valeur initiale***] [** *{propriétés}* **]** **[ ]** : éléments optionnels

+ - # *[0..1], [n]* *{constant, addOnly}*

exemple : *+ taille : real = 1,8* qui signifie que les objets de cette classe ont un attribut *taille* de type réel, d'accès publique et dont la valeur par défaut est 1,8 (en rajoutant *constant*, on aurait pu interdire de le modifier pour préciser par exemple que tous les objets ont définitivement une taille de 1,8).

Une méthode (opération) est l'implémentation d'un service qui peut être demandé à tous les objets de la classe modélisée. Les méthodes sont associées au comportement. La notation complète d'une méthode est la suivante :

[*visibilité***]** *nomMéthode* **[** *(paramètres:types)* **] [***:type retourné***] [** *{propriétés}* **]** **[ ]** : éléments optionnels

+ - # *(param1:type1, param2:type2)* *{query, abstract}*

exemple : *- moyenne(a,b:real):real* qui signifie que les objets de cette classe ont une méthode *moyenne* qui prend en entrée deux réels (a et b) et fournit un résultat réel. Cette méthode est privée (visible pour la classe).

En terme de lisibilité de la classe, on se contente généralement de faire apparaître sur le modèle les attributs et méthodes importants pour sa compréhension (ce qui n'empêche leur existence mais les outils informatiques UML permettent généralement de cacher certains attributs ou méthodes). Les stéréotypes (noms entre guillemets, par exemple : « caractéristiques visuelles » ou « propriétés dynamiques ») permettent de structurer l'affichage des attributs et des méthodes (plutôt que de les montrer *en vrac*).

Il est possible d'ajouter un 4ème compartiment à la représentation graphique d'une classe : la responsabilité. Ce compartiment permet de décrire les attentes vis à vis de la classe. Cette notion de responsabilité permet de revenir sur les critères de découpe d'un sujet en classe. En général, il est pertinent de se contenter d'une ou deux responsabilités par classes. Si les classes sont trop vastes (modèle peu réutilisable) ou si elles sont trop petites (modèle trop abstrait), le modèle risque d'être peu représentatif et inexploitable.

4.1.2 Relations entre classes

Le formalisme UML associe aux classes un certain nombre de relations permettant d'expliciter leurs rapports.

4.1.2.1 Généralisation / Spécialisation

Il s'agit d'une relation entre une *classe-mère* (ou super-classe) et une *classe-fille* (ou sous-classe) qui décrit la relation d'héritage entre la classe-fille et la classe-mère : la sous-classe est dérivée de la super-classe (elle hérite de l'ensemble de ses propriétés : attributs, méthodes, associations). La représentation graphique est la suivante (visuel créé avec *Objecteering UML Modeler*) :

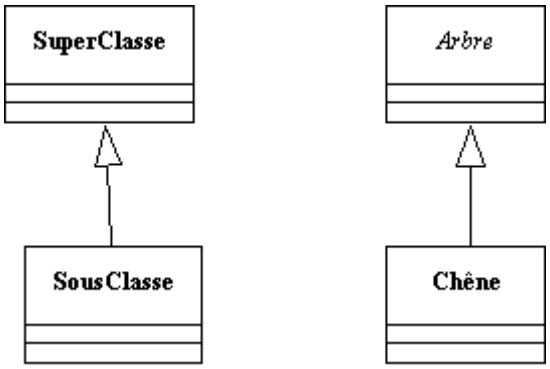


Figure n°10 : ***Représentation de la relation de généralisation / spécialisation (vue de principe et exemple)***

Ce type de relation permet l'introduction de classes dites *abstraites*. Il s'agit de classes dont le nom est indiqué en italique et qui ne peuvent pas être instanciées. *Arbre* est une classe abstraite dans la mesure où tout arbre réel instancie une espèce donnée et non la classe *Arbre* (il n'existe pas d'arbre d'aucune espèce).

L'héritage peut être multiple (une classe-fille héritant de plusieurs classes-mères) afin de préciser en particulier la nature multiple d'une classe. Ce type d'héritage peut s'avérer relativement délicat à manipuler du fait des « collisions » qui peuvent être entraînées parmi les attributs et les méthodes. En effet, la classe-fille n'hérite pas de l'union des caractéristiques de ses classes-mères mais de la somme. Si les classes-mères disposent de propriétés communes, la classe-fille hérite des deux propriétés (estampillées chacune de leur provenance afin de les différencier).

Ces considérations amènent à évoquer la notion de *polymorphisme*. Une classe-fille peut redéfinir ou simplement compléter une propriété héritée d'une classe mère (en particulier une méthode qui peut être précisé dans le cadre d'utilisation de la classe-fille). Les descendants de la classe-fille hériteront théoriquement de la nouvelle propriété (celle affinée par la définition de la classe-fille) qui remplace la propriété initiale dans la chaîne d'héritage. En pratique, les outils logiciels de modélisation orientée objet ont tendance à conserver les deux propriétés dans la chaîne d'héritage (dès la classe qui redéfinit la propriété et qui possède ainsi les deux versions) et à laisser à la phase d'utilisation de ces propriétés la responsabilité de préciser celle qui est utiliser (si la précision n'est pas donnée, c'est la version la plus récente de la propriété, i.e. celle redéfinie, qui devient prioritaire). Cette notion de redondance de signature de propriété est désignée sous le terme de *polymorphisme*.

Concernant les notions d'héritage multiple, la représentation graphique est la suivante (visuel créé avec *Objecteering UML Modeler*) :

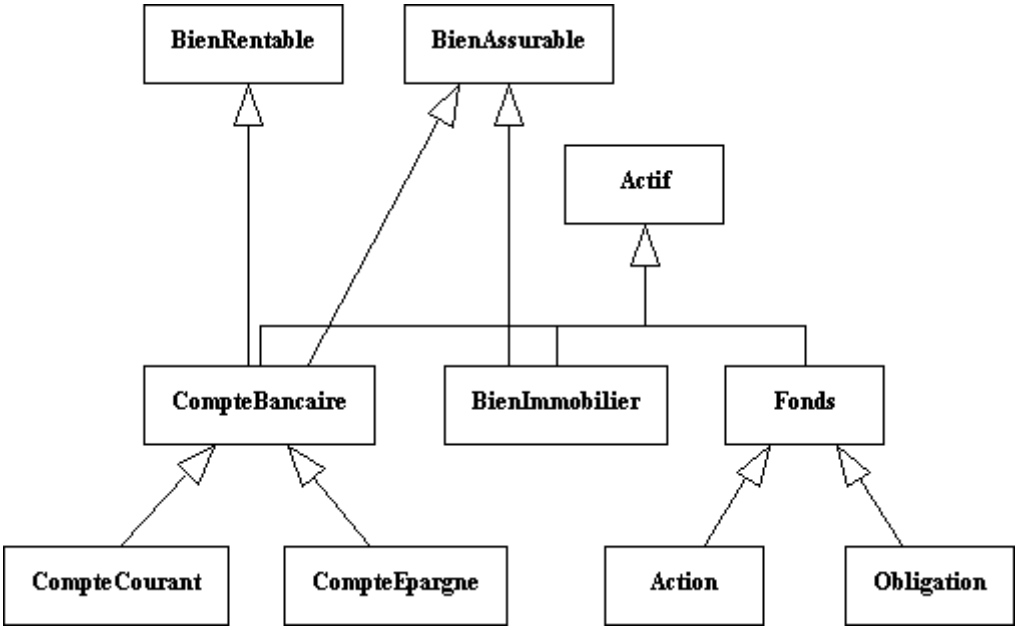


Figure n°11 : ***Représentation d'héritages multiples (exemple issu d'une application bancaire)***

4.1.2.2 Association

Il s'agit de la relation la plus générale entre classes. Cette relation permet de donner du sens (l'association est complétée d'indications sémantiques) à un lien organisationnel entre classes.

L'association permet de décrire une relation structurelle entre classes, enrichie d'un contenu sémantique

Une association peut concerner une seule classe (association unaire) ou plusieurs classes (association binaire ou n-aire). La représentation graphique est la suivante (visuel créé avec *Objecteering UML Modeler*) :

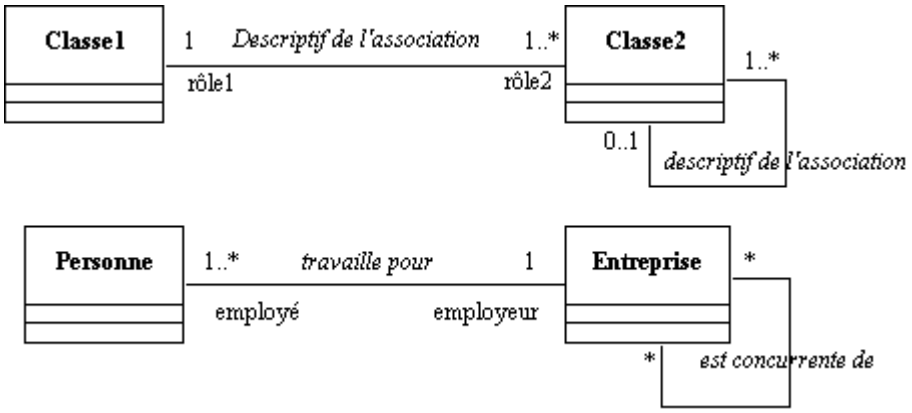


Figure n°12 : ***Représentation d'associations entre classes (vue de principe et exemple)***

Une association peut contenir un *descriptif* (qui donne du sens à l'association et justifie son existence), des *cardinalités* (1 = un et un seul, 1..* = au moins 1, n = une valeur précise, * = une valeur quelconque) qui précise dans quelles quantités l'association doit être instanciée, des *rôles* qui permettent de préciser l'implication de chaque classe dans l'association.

Il faut noter qu'il est également possible d'enrichir la description d'une association en complétant sa description à l'aide d'une classe qui est dédiée à l'association. Cette classe s'appelle alors une *Classe-association* et présente

les caractéristiques d'une classe (héritage, relations, attributs, méthodes…) et celles d'une association (descriptif, cardinalités, rôles…) :

- Ce type de relation traduit le fait que la classe-association n'est liée aux deux classes composant l'association que du fait de l'existence d'une relation entre elles (et non pas liée à chacune d'elle indépendamment).
- Ce type d'entité correspond globalement à un besoin de décrire les propriétés et les caractéristiques de l'association entre les deux classes (c'est une amplification de la représentation de l'association).

Une classe-association permet à la fois de compléter la description d'une association à l'aide de concepts orientés-objet et de faire apparaître un composant du sujet (classe) spécifique à la relation

La représentation graphique est la suivante (visuel créé avec *Objecteering UML Modeler*) :

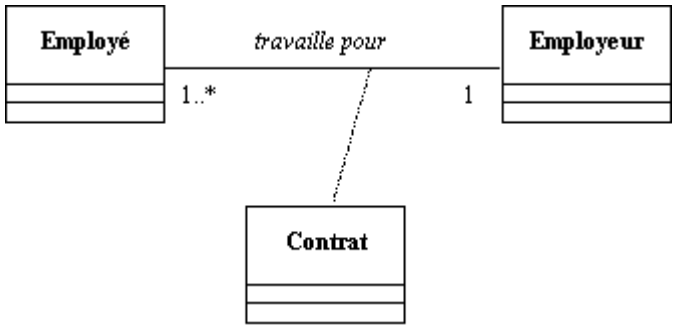


Figure n°13 : ***Représentation d'une classe-association***

Il faut être vigilant, ce type de notation ne correspond pas à la représentation d'une association entre trois classes mais bien à la description d'une relation entre la classe-association d'une part et le binôme composé des deux autres classes d'autre part.

4.1.2.3 Agrégation

Cette relation décrit les relations de constitution entre les classes. Elle peut concerner une ou plusieurs classes.

L'agrégation permet de décrire les notions de contenance ou d'appartenance entre classes

Un cas particulier d'agrégation est la composition (qui implique que le composant n'existe pas sans le composé). La représentation graphique est la suivante (visuel créé avec *Objecteering UML Modeler*) :

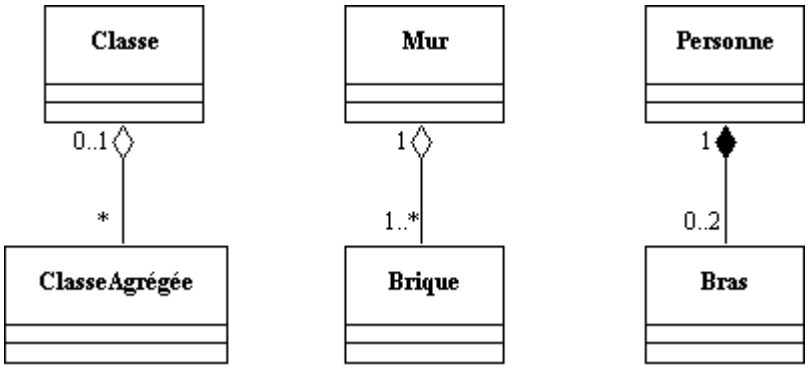


Figure n°14 : ***Représentation d'agrégations et de compositions entre classes (vue de principe et exemples)***

Le sens de cette relation est dans un sens « est élément de » ou « appartient à » et dans l'autre sens « contient » ou « est constitué de ». La relation de composition traduit le fait que le *Bras* ne peut exister sans la *Personne* à qui il appartient.

4.1.2.4 Dépendance

Sur le plan sémantique, toutes les relations (agrégation, association, généralisation) sont des relations de dépendance. C'est « l'utilisation quotidienne » qui a amené certaines sous-catégories de dépendances à être identifiées comme suffisamment fréquentes et suffisamment importantes pour donner lieu à un type de relation spécifique. Néanmoins la notion de dépendance conserve son importance en UML en général et dans la construction des diagrammes de classes en particulier.

La dépendance traduit le fait que l'existence d'un élément (en particulier son comportement) nécessite la présence d'un autre

Dans le cas des diagrammes de classes, cette relation traduit généralement la notion d'utilisation d'une classe par une autre. D'un point de vue pratique, une relation de dépendance correspond souvent au positionnement d'une classe comme paramètre d'une méthode d'une autre classe. La représentation graphique est la suivante (visuel créé avec *Objecteering UML Modeler*) :

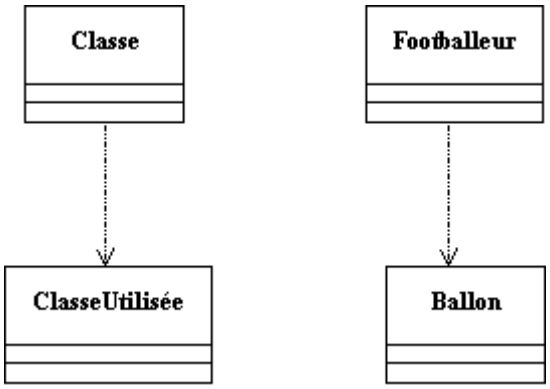


Figure n°15 : ***Représentation de dépendance entre classes (vue de principe et exemples)***

Dans l'exemple précédent, la classe *Footballeur* dispose d'une méthode *shooter* dont un paramètre est le *Ballon* (la nature du ballon influencera cette fonction).

Compte tenu du caractère « englobant » et « originel » de la relation de dépendance, il existe un certain nombre de stéréotypes permettant de nuancer le sens de cette relation (les stéréotypes liés aux notions d'association, de généralisation et d'agrégation ont disparu pour laisser place à ces relations elles-mêmes). En théorie, il existe 17 stéréotypes accessibles pour nuancer ou catégoriser une relation de dépendance (*bind*, *derive*, *friend*, *instanceOf*, *instanciate*, *powertype*, *refine*, *use*, *access*, *import*, *extend*, *include*, *become*, *call*, *copy*, *send*, *trace*).

En pratique ces stéréotypes ne sont que rarement utilisés dans les diagrammes de classes (beaucoup plus couramment dans d'autres diagrammes : *include* et *extend* sont très présents dans les diagrammes de cas d'utilisation) à moins qu'on ne cherche à obtenir un modèle destiné à l'implémentation extrêmement détaillé.

4.1.3 Notion de Paquetage (Package)

Les paquetages permettent la structuration verticale des composants très « à plat » en UML par rassemblement des éléments sous une dénomination commune. Ils permettent d'organiser et de hiérarchiser bien d'autres diagrammes que les diagrammes de classes mais ils sont néanmoins très utilisés dans cette vue structurelle.

Les paquetages constituent des outils de regroupement et d'organisation des composants UML

Un paquetage (appelé aussi *package*) regroupe des classes présentant un lien sémantique ou structurel fort et les cache complètement au reste de l'environnement. Les classes contenues dans le paquetage pourront alors être *publiques* (+ *public*), *protégées* (# *protected*) ou *privées* (- *private*). Or, les paquetages peuvent être connectés par des relations d'héritage ou de dépendance. Cette relation de dépendance est généralement stéréotypée « *import* » (ou plus rarement « *access* ») et signifie que les classes du paquetage qui fait l'import pourront accéder librement aux classes publiques du paquetage importé. Les classes privées demeurent quant à elles inaccessibles et les classes protégées ne sont visibles que pour les paquetages descendant du paquetage initial.

Cette notion de paquetage permet de décomposer et d'organiser un diagramme (de classes en particulier) et de ne pas être obligé de le construire « à plat », elle permet également de répartir la tâche de modélisation en spécifiant les paquetages et leur contenu exportable (i.e. les classes publiques).

Les paquetages se représentent et s'utilisent, en terme d'import en particulier, comme présenté sur la figure suivante (visuel créé avec *Objecteering UML Modeler*) :

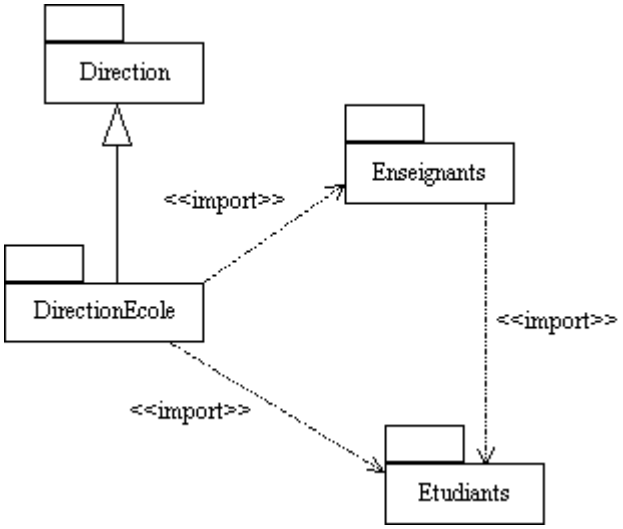


Figure n°16 : ***Représentation de paquetages (packages)***

4.1.4 Exemple de diagramme de classes

Si l'on considère la représentation simplifiée d'un logiciel de dessin basique, on peut considérer qu'il s'agit d'une *interface* offrant deux zones à l'écran, l'une pour le *choix des fonctions-outils* (gomme, crayon, coloriage…) et l'autre pour la *feuille de dessin*. C'est la partie représentant la feuille de dessin qui appliquera aux mouvements de la souris une *fonction-outil* parmi celles proposées dans la partie choix. Un diagramme de classes décrivant cet outil peut être le suivant (visuel créé avec *Objecteering UML Modeler*) :

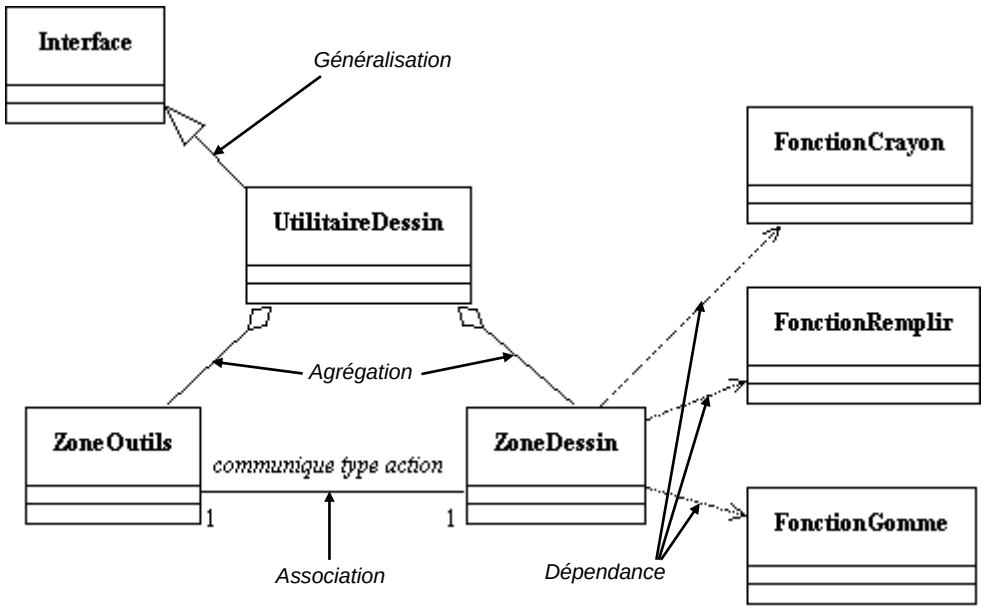


Figure n°17 : ***Exemple de diagramme de classes (utilitaire de dessin simplifié)***

Comme toute expression dans un langage évolué, ce modèle peut être contesté (il faut alors pouvoir le justifier d'une manière rigoureuse et indiscutable) ou reformulé (un autre diagramme peut être tout aussi cohérent avec le sujet bien que différent).

4.2 Diagramme d'objets (Object Diagram)

Un diagramme d'objets représente une vision réelle du schéma théorique décrit par le diagramme de classes. On va retrouver de nombreuses analogies et similitude entre le diagramme d'objets et le diagramme de classes.

4.2.1 Objet

Le formalisme UML décrit un objet en utilisant la symbolique suivante (visuel créé avec *Objecteering UML Modeler*) :

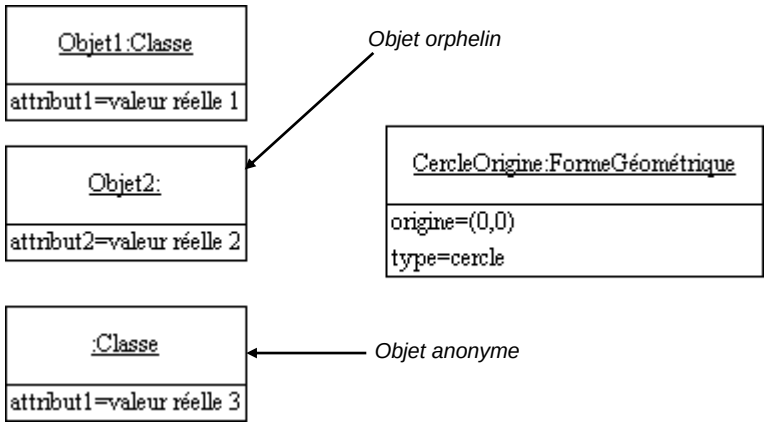


Figure n°18 : ***Représentation d'objets***

L'objet qui porte le nom *Objet1* instancie la classe *Classe* et l'attribut *attribut1* de cette classe *Classe* est valué pour cet objet à *valeur réelle 1*. L'objet *Objet2* est dit « orphelin » car on ne sait pas quelle classe il instancie, son attribut *attribut2*est valué à *valeur réelle 2*. L'objet *:Classe* est dit « anonyme » car on sait qu'il instancie la classe *Classe* mais on ne sait pas son nom, par contre, son attribut *attribut1* est valué à *valeur réelle 3*. L'objet *CercleOrigine* instancie la classe *FormeGéométrique* (cf. Figure n°9) et ses attributs *origine* et *type* sont valués en accord avec les spécifications de la classe. On peut remarquer que sur cette représentation, il a été choisi de ne pas faire figurer les méthodes des objets (issues de la classe).

4.2.2 Exemple de diagramme d'objets

Si l'on se base sur l'exemple de logiciel de dessin vu lors de la présentation du diagramme de classes, on peut prendre comme illustration le cas de *mon* logiciel de dessin, nommé *Dessinlog* et présenter le diagramme suivant (visuel créé avec *Objecteering UML Modeler*) :

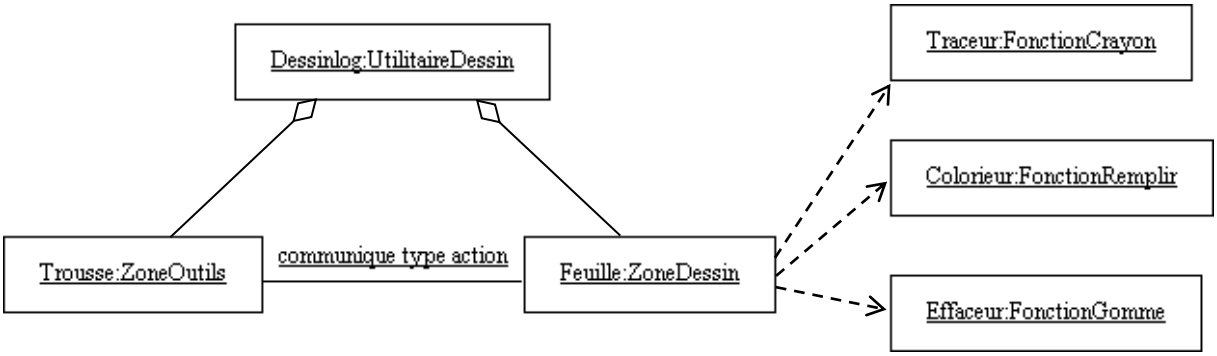


Figure n°19 : ***Exemple de diagramme d'objets (un utilitaire de dessin simplifié)***

On peut noter que le contenu des objets n'est pas représenté. Le diagramme en est plus lisible et la présence de ces contenus ne s'imposait pas dans ce schéma.

La représentation des relations est similaires à celle utilisée pour les diagrammes de classes à la différence près que les *associations* sont instanciées en *liens* (notation identique aux associations avec le descriptif souligné). Cependant, usuellement on se contente généralement dans un diagramme d'objet d'utiliser les instances d'associations (i.e. les *liens*) qui traduisent la sémantique du diagramme (et apporte sa spécificité) alors que les autres relations (agrégation, généralisation, dépendance) reprennent des caractéristiques structurelles inhérentes au diagramme de classes originel.

5. Modèle Comportemental

Les diagrammes de séquence, de collaboration, d'activité et d'états-transitions composent la vue comportementale du formalisme UML. Ces diagrammes permettent de décrire la dynamique interne du sujet. En pratique, on choisit généralement de représenter le comportement du système en utilisant un sous-ensemble de ces quatre diagrammes car s'ils décrivent des aspects complémentaires du fonctionnement, les recouvrements et redondances sont nombreux. Il s'agit donc de choisir le ou les diagrammes les plus pertinents vis à vis de l'objectif de modélisation.

5.1 Diagramme de séquence (Sequence Diagram)

Ce diagramme, qui fait partie de la description logique et dynamique, permet de représenter le déroulement de scénarios au travers d'une vision séquentielle et chronologique des échanges et interactions entre les éléments intervenant (acteurs ou composants du modèle).

Les diagrammes de séquence présentent le déroulement d'une phase d'activité du système en le caractérisant par l'enchaînement temporel des échanges entre les éléments y participant

Un diagramme de séquence regroupe donc les objets et acteurs concernés par un même scénario et décrit leurs échanges au moyen d'actions et de messages.

5.1.1 Structure d'un diagramme de séquence

Un diagramme de séquence regroupe l'ensemble des acteurs et objets concernés par le déroulement du scénario décrit. Chacun de ces éléments dispose de sa *ligne de vie* qui va permettre de représenter ses *périodes d'activités* (rectangles) et de positionner temporellement les échanges (visuel crée avec *Objecteering UML Modeler*) :

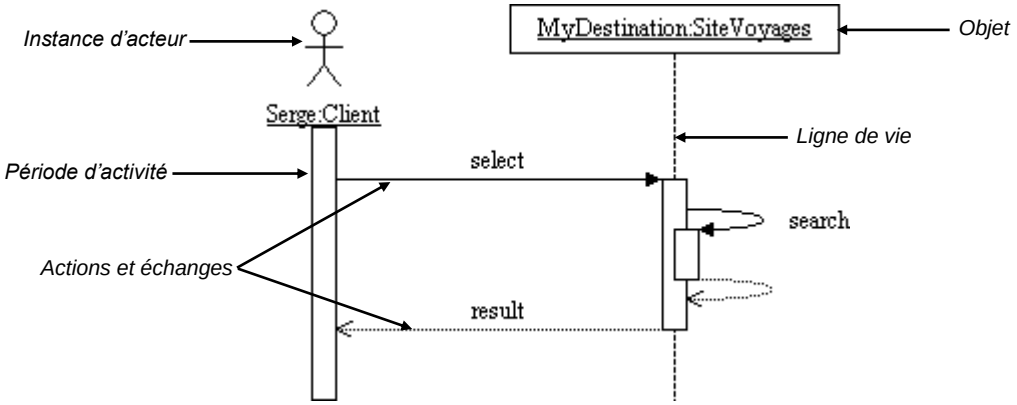


Figure n°20 : ***Allure globale d'un diagramme de séquence***

5.1.2 Actions et messages

Les objets et acteurs mis en œuvre dans la description d'un scénario agissent et communiquent par le biais d'actions et de messages disponibles dans le cadre de la construction d'un diagramme de séquence.

5.1.2.1 Appel de méthode (Call / Return)

Le principe de l'échange *Call/Return* est de permettre à un objet d'invoquer et de déclencher (faire appel) une opération d'un autre objet. L'objet émetteur du message doit donc être en droit d'accéder aux méthodes de l'objet récepteur (association entre les classes) et l'objet récepteur doit de son côté être porteur de la méthode appelée. Le message porte le nom de la méthode appelée (qui est une aptitude de l'objet cible).

Il est donc indispensable d'assurer la cohérence entre le diagramme de classes et un diagramme de séquence : les méthodes nécessaires à l'écriture d'un diagramme de séquence pertinent doivent apparaître dans les classes concernées. Cette notion de cohérence entre les différents diagrammes permet de garantir l'homogénéité du modèle et d'autoriser des retours constructifs sur les diagrammes déjà réalisés (il est très courant de revenir, par

exemple, sur un diagramme de classes lors de la réalisation d'un diagramme de séquence parce qu'on se rend compte qu'une méthode manque dans une classe telle qu'elle a initialement été conçue).

Si l'on considère un scénario simple mettant en jeu un individu utilisant une télécommande élémentaire et une télévision, on peut obtenir le diagramme de séquence suivant (en lien avec un diagramme de classes) correspondant à la mise en marche de la télévision depuis la télécommande, à quelques manipulations (changement de chaîne, diminution du son) et à sa mise en veille (visuel crée avec *Objecteering UML Modeler*) :

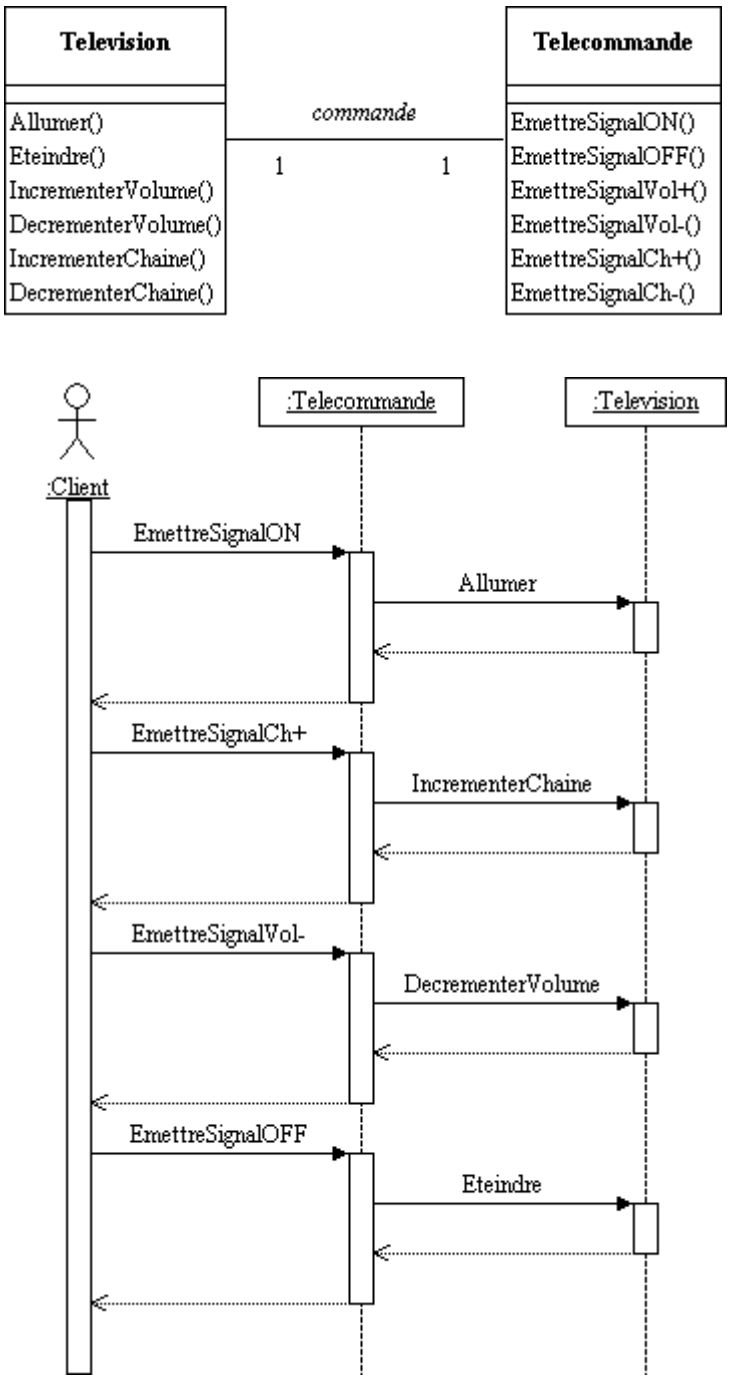


Figure n°21 : ***L'appel de méthode dans un diagramme de séquence***

Ce diagramme permet de voir que les opération de la télécommande (les méthodes *EmettreSignal*) sont appelées par l'acteur (ce qui est bien le cas en pratique puisque c'est le fait que l'utilisateur appuie sur une touche qui amène la télécommande à déclencher une de ses opérations) et que les méthodes de la télévision sont appelées par la télécommande (le changement de canal ou la gestion du volume sont des fonctions techniques réellement internes à la télévision qui ne s'exécutent que sur ordre de la télécommande).

5.1.2.2 Création et Destruction d'Objets (Create / Destroy)

Le principe de ce type de message *Create / Destroy* dans un diagramme de séquence est de permettre à un objet d'en créer ou d'en détruire un autre au cours de l'exécution d'un scénario. La restriction à ce type d'action concerne les droits entre les classes dont sont issus les objets (droits qui doivent être en accord avec les actions de création et de destruction).

En reprenant l'exemple de la télévision, on peut introduire l'objet image et voir comment la télévision manipule cet objet (visuel crée avec *Objecteering UML Modeler*) :

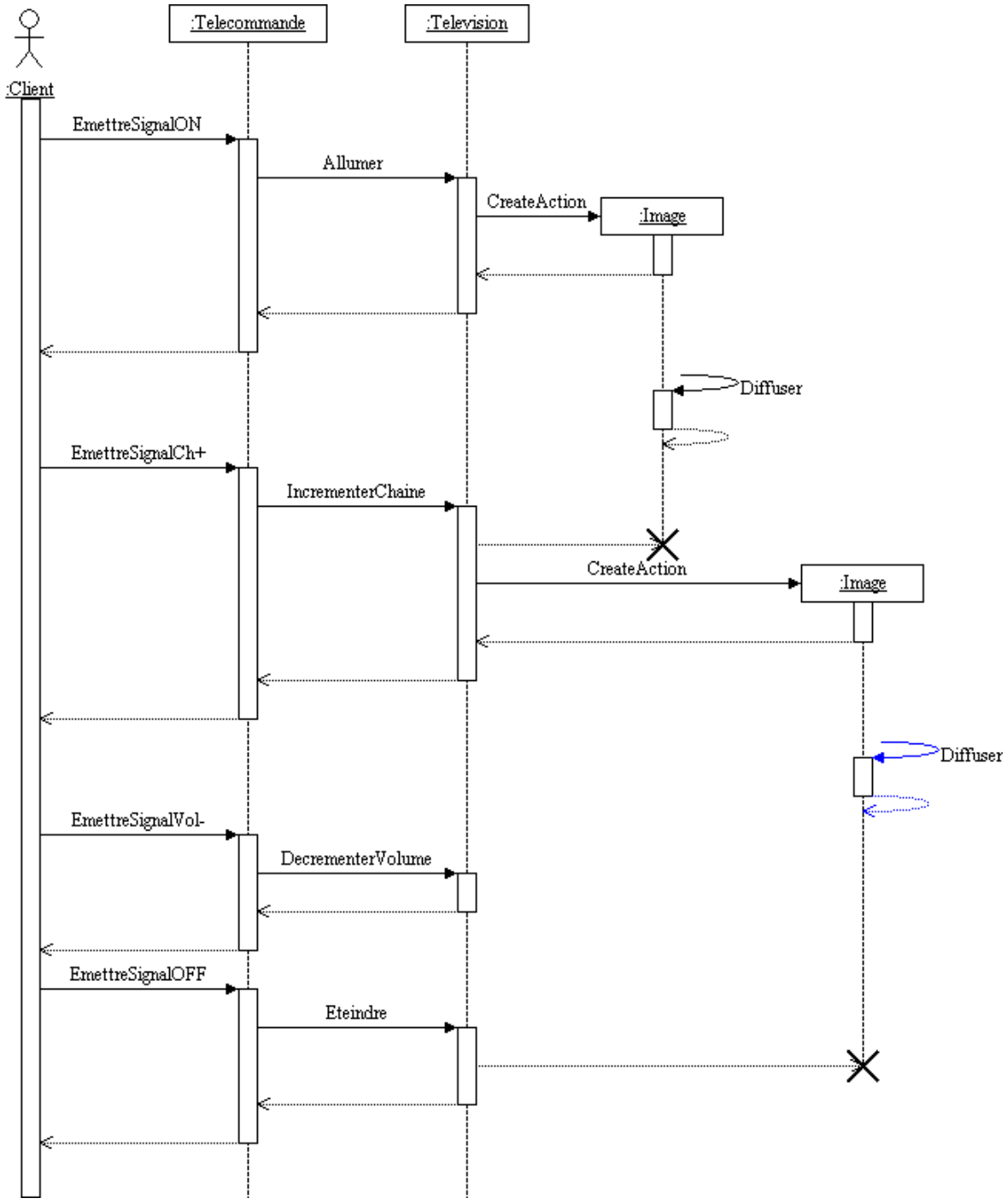


Figure n°22 : ***La création et la destruction d'objets dans un diagramme de séquence***

5.3 Diagramme de collaboration (Collaboration Diagram)

Ce diagramme dispose d'un statut un peu particulier au sein des diagrammes UML. En effet, le diagramme de collaboration reprend les principes du diagramme de séquence dans un cadre de présentation différent.

Les diagrammes de collaboration regroupent les éléments concernés par un même scénario et rappelle leurs interactions

Pour illustrer cette notion, on peut dire que si le diagramme de séquence permet la description d'un scénario dans un cadre chronologique, le diagramme de collaboration représente le groupe d'objet concerné par le déroulement de ce même scénario. Un diagramme de collaboration regroupe donc les objets et acteurs concernés par un même scénario et décrit leur cadre de coopération.

5.3.1 Structure d'un diagramme de collaboration

Un diagramme de collaboration décrit un groupe d'objets dans le cadre d'un comportement particulier du système. Ce type de diagramme présente donc un groupe d'objets concernés par un même scénario en précisant les échanges et les appels de méthodes effectués au cours du scénario. De manière similaire à ce qui concerne le diagramme de séquence, la construction d'un diagramme de collaboration nécessite une cohérence avec le diagramme de classes. Cette cohérence passe par les associations existant entre les classes dont sont issues les objets (un objet ne peut émettre un message ou appeler une méthode d'un autre objet que dans le cadre d'une relation entre les classes dont ils proviennent) mais également par les méthodes de ces classes (on ne peut évidemment appeler qu'une méthode existant chez un objet).

Le diagramme de collaboration décrivant l'utilisateur allumant et éteignant le téléviseur est le suivant (visuel crée avec *Objecteering UML Modeler*) :

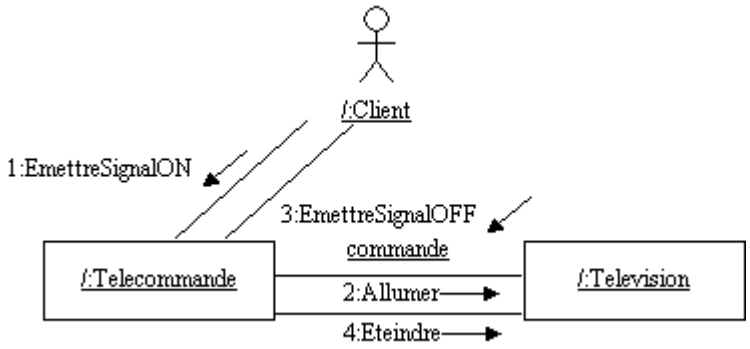


Figure n°23 : ***Le diagramme de collaboration « Allumer et éteindre la télévision »***

On peut constater que ce diagramme instancie les objets à partir des classes et décrit au moyen d'une numérotation l'ensemble des actions effectuées dans le cadre du scénario concerné (ces actions étant effectuées en accord avec l'instanciation des associations existant entre les classes, cf. « commande » qui permet l'action de la télécommande sur la télévision).

Ce diagramme correspond à une vision duale du diagramme de séquence en intégrant les notions héritées de la structure logique présentée dans le diagramme de classes.

5.3.2 Notion de collaboration

En UML, la notion de collaboration est associée à la *réalisation* d'un élément dynamique. Une collaboration possède une facette structurelle et statique et une facette comportementale et dynamique. Cette notion désigne un ensemble d'éléments du système qui interagissent et travaillent ensemble pour assurer un comportement du système résultant de la coopération de ces éléments.

Il est important de noter que la notion de coopération est fortement associée au concept d'exploitation des composants du système. Ainsi, parmi un ensemble de composants d'un modèle, une première sous-partie peut-être concernée par une première collaboration alors qu'une autre sous-partie (présentant certains recouvrements

avec la première) peut quant à elle être concernée par une autre collaboration. C'est ce concept d'exploitation coopératif qui justifie de la présence des collaborations. La figure suivante illustre conceptuellement l'intérêt de l'existence de cette notion de collaboration à partir d'un diagramme de classes :

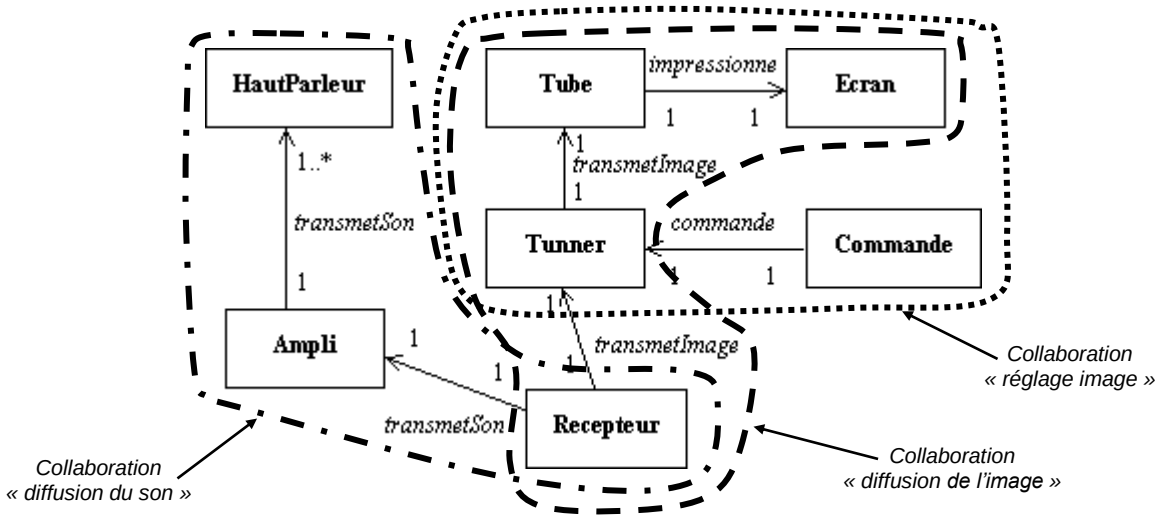


Figure n°24 : ***Exemple de collaborations dans un système***

Les collaborations, en tant que lien entre les plans structurel et comportemental, se trouvent associées à une opération ou à un cas d'utilisation. En pratique, on peut souhaiter attacher une collaboration à toutes sortes d'éléments de modélisation : une classe, un package… Une collaboration se représente conceptuellement sous la forme suivante :

NomDeLaCollaboration

Figure n°25 : ***Représentation d'une collaboration***

Le concept de collaboration doit être expliciter car il ne correspond pas simplement à un sous-ensemble de composants mais bien à la mise en action de ce groupe de composants dans le cadre d'une tâche particulière (scénario, cas d'utilisation, opération et autre phase dynamique). Une collaboration s'explicite donc en UML à l'aide d'un diagramme de séquence ou d'un diagramme de collaboration (le nom apparaît alors ambigu car il ne s'agit pas de la seule façon de décrire une collaboration).

5.3.3 Comparaison entre diagramme de séquence et diagramme de collaboration

Il semble évident, d'un point de vue conceptuel, de constater que ces deux types de diagrammes UML (collaboration et séquence) présentent une proximité d'expressivité et de sémantique qui leur confère un statut de *cousins* voire de *frères* (ils sont les outils de description d'une collaboration). Ces deux types de diagrammes sont d'ailleurs souvent désignés communément sous la dénomination « diagrammes d'interaction ».

Concernant l'apparente redondance entre ces deux diagrammes, une hypothèse quelque peu « cynique » pourrait amener à penser que puisque UML résulte de l'unification de différents langages de modélisation des années 80 et 90, il est tout à fait possible que ces deux diagrammes proviennent originellement de deux formalismes différents et que malgré leur indéniable proximité ils aient tous deux été conservés au sein du langage terminal pour des raisons qui peuvent être variées (nuances entre les deux qui s'avèrent constructives, attachement des différents auteurs des langages à ces diagrammes, négociations au cours de l'unification…).

Néanmoins, s'il ne faut pas occulter ce genre de possibilité de « pieds de nez historiques », on peut constater que les diagrammes de séquences et les diagrammes de collaboration présentent des nuances tout à fait structurantes par rapport aux objectifs du formalisme UML et qui peuvent en elles-mêmes justifier leur présence conjointe.

Afin de faciliter la sélection de l'un ou de l'autre en fonction des paramètres associés au modèle (objectif, positionnement dans le cycle de conception, domaine technologique concerné…), il est possible de proposer un tableau synthétique récapitulatif des caractéristiques respectives et relatives de ces deux types de diagrammes :

	Diagramme de séquence	**Diagramme de collaboration**
Thème	*Orientation vers le séquencement*	*Orientation vers le regroupement*
Objet	*Adaptation à la modélisation temps-réel*	*Adaptation aux contraintes de proximité*
Rôle des échanges	*Les échanges y sont présentés pour leur ordonnancement*	*Les échanges y sont présentés pour les couples émetteur/récepteur qu'ils induisent*
Positionnement	*Pertinence par rapport à l'analyse et à la conception*	*Pertinence par rapport à la conception et au développement*

Figure n°26 : ***Comparaison des diagrammes d'interaction***

Si ces deux types de diagrammes présentent certaines caractéristiques propres qui permettent de nuancer leur recouvrement et de souligner leurs intérêts respectifs, il n'en demeure pas moins qu'ils sont sémantiquement équivalents et ne peuvent être utilisés que de manière dissociée.

5.4 Diagramme d'activité (Activity Diagram)

Un diagramme d'activité est un décrit une séquence d'actions relatives à une tâche (activité) particulière.

Les diagrammes d'activités décrivent une phase active du sujet au moyen d'organigrammes regroupant une succession d'étapes organisées séquentiellement

Cette description met en jeu les objets, leurs méthodes et les messages qu'ils échangent. On peut illustrer la différence qui existe entre diagrammes de séquence et diagrammes d'activité en la comparant à celle qui existe entre programmation procédurale et programmation orientée objet.

5.4.1 Composants d'un diagramme d'activité

Un diagramme d'activité se compose d'états d'activité ou d'états d'action et de transitions. Un état d'activité est un état décomposable de durée non-nulle comportant des actions d'entrées et/ou de sortie. Un état d'action est un état atomique (ou élémentaire) d'une durée négligeable.

D'un point de vue plus formel, les éléments de vocabulaire dont le modeleur dispose pour réaliser un diagramme d'activité sont les suivants :

- Etat initial et état final,
- Etat d'action / état d'activité
- Branchement conditionnel,
- Synchronisation (fourches et jonctions),
- Travées (découpes structurantes du diagramme),
- Objets (créés ou modifiés par un état).

5.4.2 Structure d'un diagramme d'activité

On retrouve les éléments précédents dans le diagramme d'activité sommaire suivant qui décrit la procédure d'identification d'un utilisateur (visuel crée avec *Objecteering UML Modeler*) :

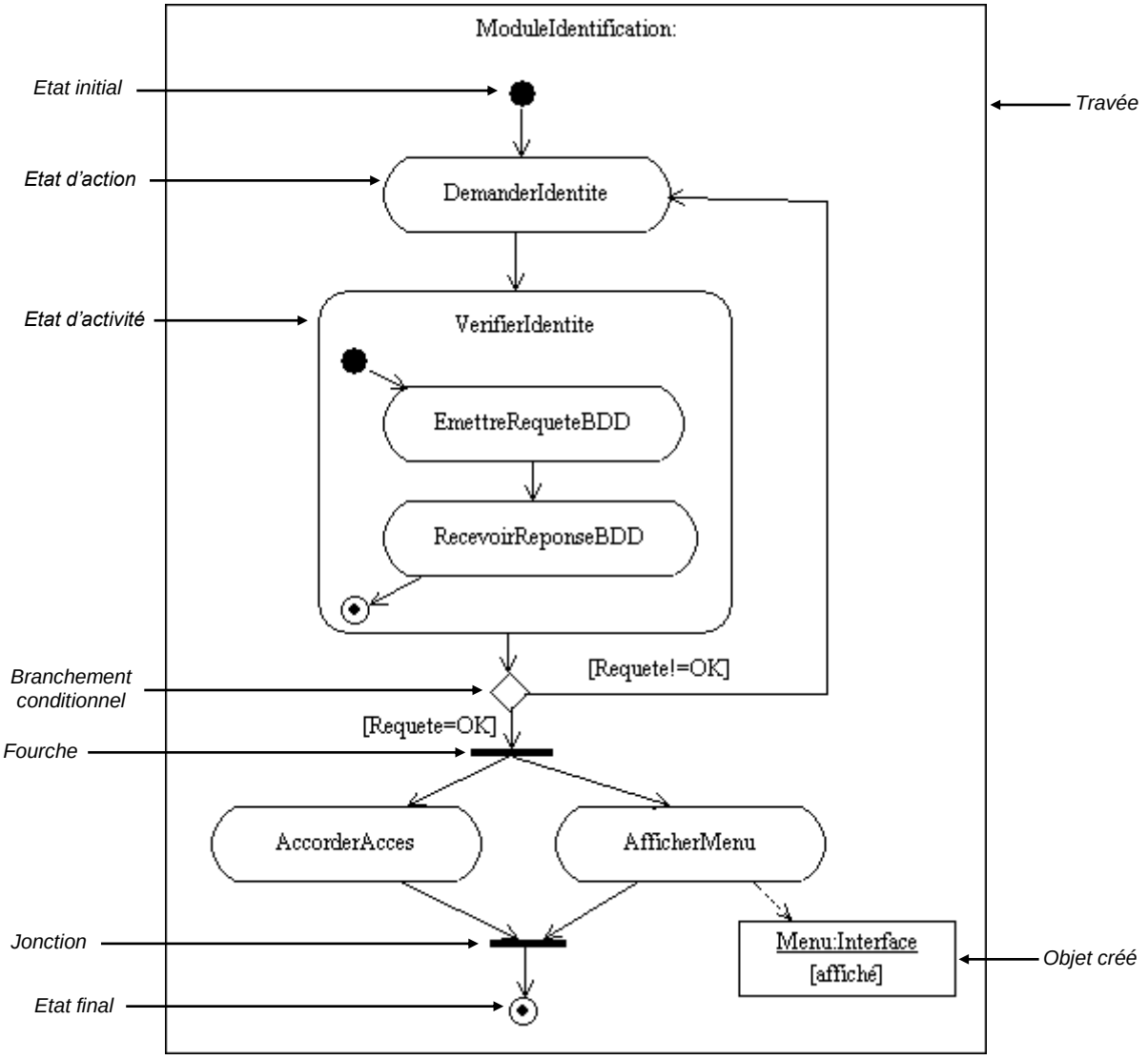


Figure n°27 : ***Composition d'un diagramme d'activité***

Le diagramme précédent permet de constater que les fourches et jonctions jouent effectivement un rôle de synchronisation (les états d'actions *AccorderAcces* et *AfficherMenu* sont initiés simultanément de même que l'état final n'est atteint que lorsque ses deux actions sont terminées).

L'objet Menu est créé par l'action *AfficherMenu* et sont état est précisé à ce stade du diagramme. Un objet modifié par un état d'action ou un état d'activité sera systématiquement associé à celui-ci et son nouvel état (conséquence de l'intervention de l'action ou de l'activité sur cet objet) sera précisé entre crochets.

Le formalisme relatif aux diagrammes d'activité est très similaire à celui parfois employé pour décrire des algorithme en langage procédural. Ce diagramme peut être utilisé comme outil de description de processus.

5.5 Diagramme d'états-transitions (State-Transition Diagram)

Un diagramme d'états-transitions est un outil de description du fonctionnement de chaque composant du sujet. Tout objet issu d'une même classe répondra au comportement décrit à l'aide du diagramme d'états-transitions.

Les diagrammes d'états-transitions décrivent le comportement d'une classe sous la forme d'un automate à états finis

L'enjeu de ce type de diagramme est de bien choisir les classes dont on va décrire le comportement à l'aide de ce type de diagramme. Le formalisme des diagrammes d'états-transitions reprend celui des *Statecharts*.

5.5.1 L'Etat

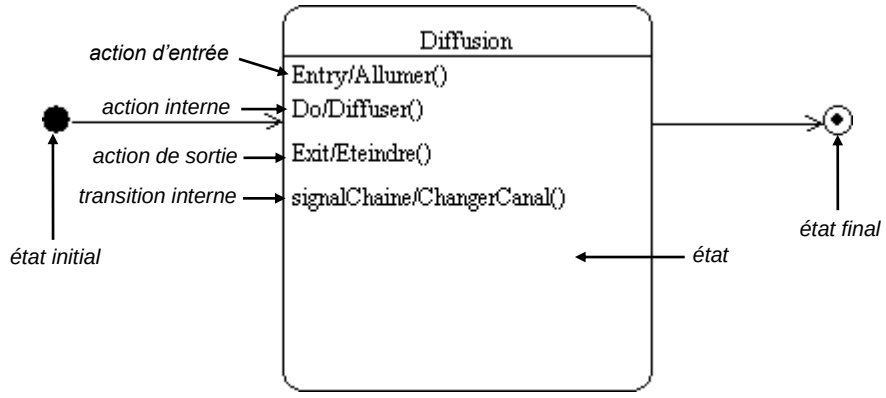
Un état constitue une situation ou une condition (au sens de condition humaine) accessible pour les objets relatifs à la classe décrite. Lorsqu'un objet est dans un état donné, il peut exécuter une activité (éventuellement composée d'actions) ou attendre un événement.

Un état est une <u>situation</u> qui se produit dans la vie d'un objet et au cours de laquelle l'objet satisfait une <u>condition</u>, réalise une <u>activité</u> ou attend un <u>évènement</u>

Un état est caractérisé par un nom et comporte :

- des actions d'entrées : elles sont optionnelles et sont exécutées à chaque entrée dans l'état (quel que soit le chemin par lequel on a accédé à l'état),
- une activité interne : il s'agit d'action(s) interne(s) qui compose(nt) le comportement interne de l'état,
- des actions de sortie : elles sont optionnelles et sont exécutées à chaque sortie de l'état (quel que soit le chemin de sortie de l'état,
- de transition(s) interne(s) : appelées aussi auto-transitions, constituées du couple *événement/action*, elles interrompent l'activité interne en cas d'apparition de *l'événement* et d'exécutent *l'action* associée.
- d'événements différés : il s'agit d'une liste d'événements suivie de /*defer* qui, s'ils se produisent, seront ignorés mais conservés en mémoire afin d'être déclarés dans le prochain état (à condition qu'ils ne soient pas non plus déclarés comme événement différés dans celui-ci).

Il existe deux états particuliers, l'état initial et l'état final. Ces deux états constituent des pseudo-états car ils ne contiennent pas de comportement et marquent simplement le début et la fin du comportement de la classe. La représentation graphique d'un état est la suivante (visuel crée avec *Objecteering UML Modeler*) :



<u>Figure n°28</u> : ***l'état dans le diagramme d'états-transitions***

<u>Remarque</u> : *Objecteering UML Modeler* ne permet pas (dans la version utilisée pour rédiger ce document) la représentation des événements différés.

5.5.2 La Transition

Les transitions permettent le passage d'un état à un autre (d'un état source à un état cible).

Une transition est une <u>relation entre deux états</u>, contraintes par certaines <u>conditions</u> et accompagnée de certaines <u>actions</u>

La transition se compose de plusieurs éléments (tous optionnels) :

- un événement déclencheur : occurrence d'un stimulus qui va déclencher la transition. Ce stimulus peut être de type varié : signal, appel, événement temporel, événement de modification,
- une condition de garde : expression booléenne placée entre crochets qui sera évaluée ponctuellement à l'apparition de l'événement déclencheur et qui conditionnera la validation de l'événement. Si pour un événement donné la condition de garde n'est pas vérifiée, l'événement sera perdu,

- une action : cette action, précédé d'un symbole « / », sera effectuée au passage de la transition.

La représentation graphique d'une transition est la suivante (visuel crée avec *Objecteering UML Modeler*) :

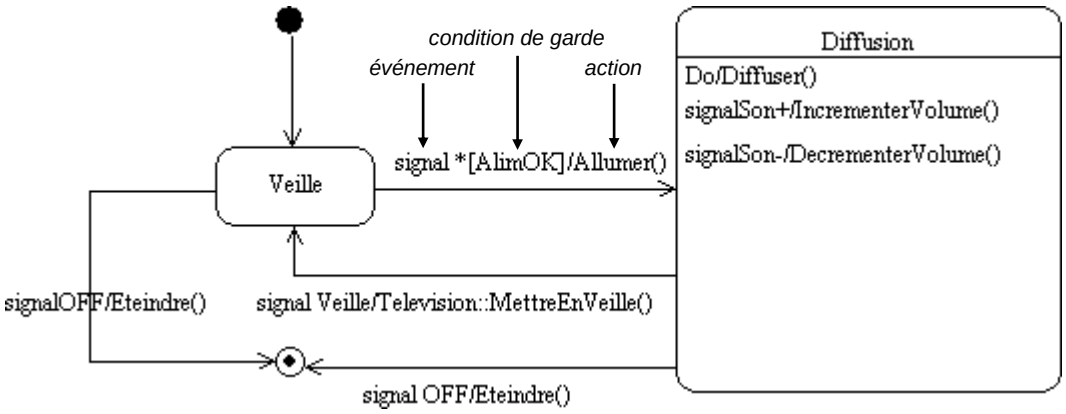


Figure n°29 : ***l'état dans le diagramme d'états-transitions***

Dans cet exemple, l'événement *signal** (c'est à dire un signal quelconque de la télécommande) conditionné par la condition de garde *AlimOK* (c'est à dire que la télévision soit encore sous tension) permet le franchissement de la transition vers l'état diffusion avec l'exécution de l'action *Allumer( )*.

Il est important de noter que la seule différence entre une action sur une transition et une action d'entrée est que la première ne sera effectuée que si la transition est franchie alors que la seconde sera exécutée à l'issue de tout franchissement d'une transition amenant dans l'état.

Ainsi, dans l'exemple de la figure n°22, l'action *Allumer( )* pourrait être placée, de manière rigoureusement équivalente, en action d'entrée de l'état *Diffusion* dans la mesure où il n'y a qu'un chemin d'accès à cet état (par la transition *Veille → Diffusion*).

5.5.3 L'Evénement

Un événement peut être interne ou externe (signal d'un composant, commande d'un utilisateur, fait du à l'environnement…) et intervient dans le déroulement de la vie du système.

Un événement relève de l'occurrence localisée (dans le temps et dans l'espace) d'un stimulus

Les évènements se classent parmi les quatre types suivants :

- signal : il s'agit d'un objet envoyé et reçu par des objets du système. C'est donc un objet ordinaire du modèle faisant partie d'une structure hiérarchisée de signaux. Il sera associé à une opération de l'objet émetteur par le biais d'une association de dépendance stéréotypée « *Send* » et les classes sensibles à ce signal comporteront un compartiment « abonnement » supplémentaire dans leur structure contenant les signaux auxquels la classe est abonnée (représentation non-incluse dans *Objecteering UML Modeler*),
- appel : il s'agit du déclenchement d'une opération,
- événement temporel : mesure d'une durée à l'aide de l'instruction *After* (*After 1s*…),
- événement de modification : surveillance d'un instant particulier à l'aide de l'instruction *When* portant généralement sur des attributs de la classe (*When heure=12h15, When altitude=5000*…).

Les signaux constituent une catégorie d'événements importante dans la mesure où ces derniers s'apparentent à des objets au sein du modèle. La hiérarchie des signaux peut alors être décrite à l'aide des concepts relatifs aux classes et aux objets et s'intégrer dans les diagrammes de classes.

L'exemple suivant décrit l'arrangement des signaux relatifs au système télécommande/télévision (visuel crée avec *Objecteering UML Modeler*) :

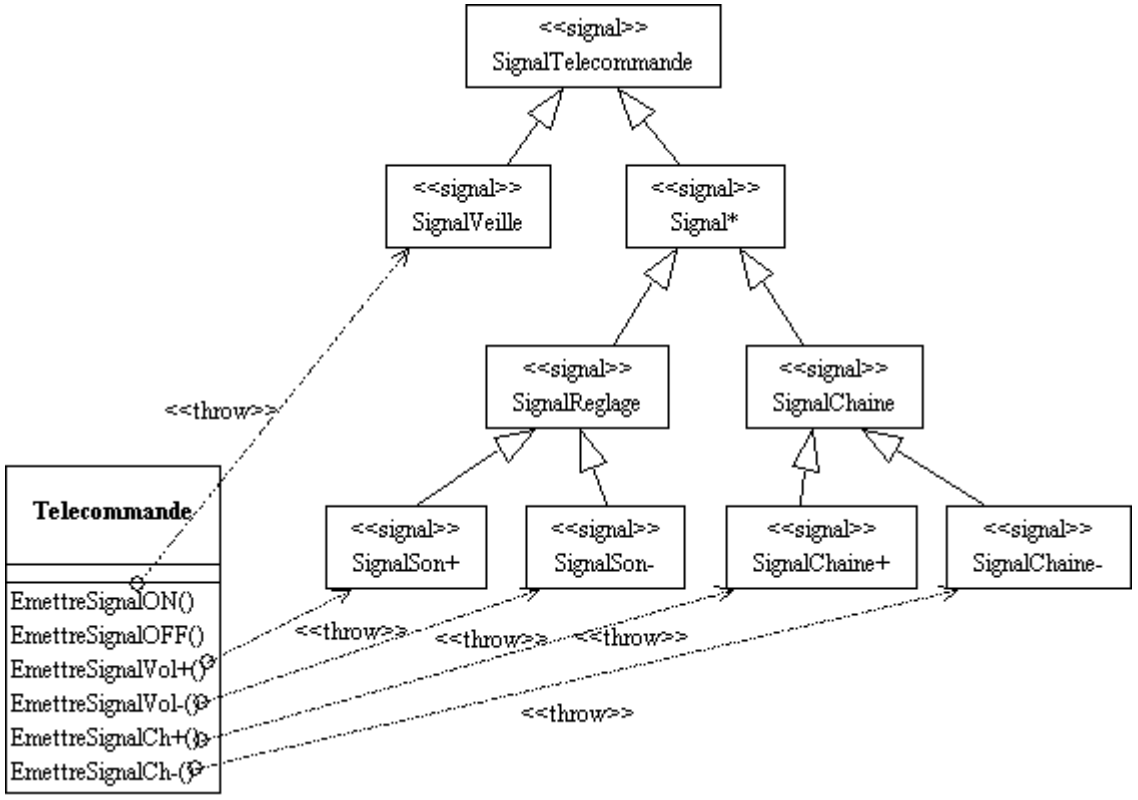


Figure n°30 : ***les signaux (hiérarchie et lien avec les classes)***

5.5.4 Principe de description des diagrammes d'états-transitions

5.5.4.1 Décomposition

Il peut être intéressant de regrouper un comportement élémentaire à l'intérieur d'un état plus général. On peut alors décrire ce comportement élémentaire à l'intérieur de l'état général désigné sous la forme de sous-états. Un sous-état est un état emboîté dans la description de l'activité d'un super-état.

La figure suivante présente l'intégration de sous-états (*Veille* et *Diffusion*) dans un super-état (*Sous-tension*) illustrant le principe de décomposition (visuel crée avec *Objecteering UML Modeler*) :

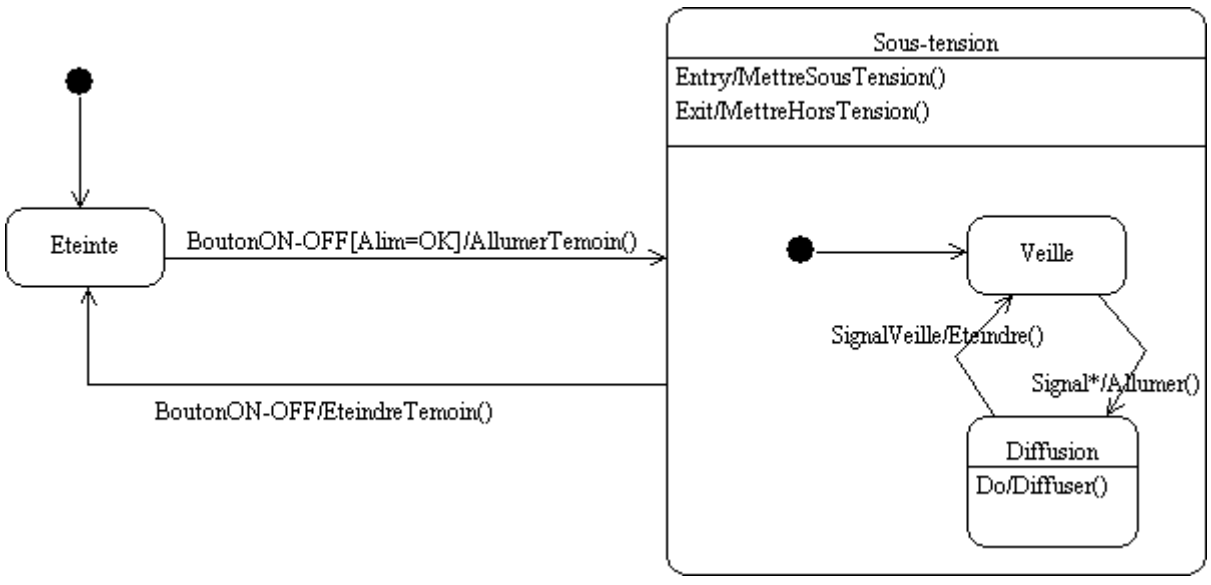


Figure n°31 : ***Décomposition d'états en sous-états***

5.5.4.2 Historique

L'évolution dans un diagramme d'états-transitions peut être interrompue et il peut alors s'avérer intéressant de pouvoir revenir dans un sous-état dans lequel on se trouvait avant l'interruption. Le symbole H (pour *Historique*) permet de représenter cette mémorisation.

Chaque entrée dans un super-état passe par l'historique et active le dernier état par lequel on est passé dans le super-état. Dans le cas du premier passage ou d'un passage indéterministe, c'est l'état par défaut (celui vers lequel pointe l'historique) qui est activé.

La figure suivante présente l'intégration de l'historique dans un super-état (*Diffusion*) illustrant le principe de « traçabilité » (visuel crée avec *Objecteering UML Modeler*) :

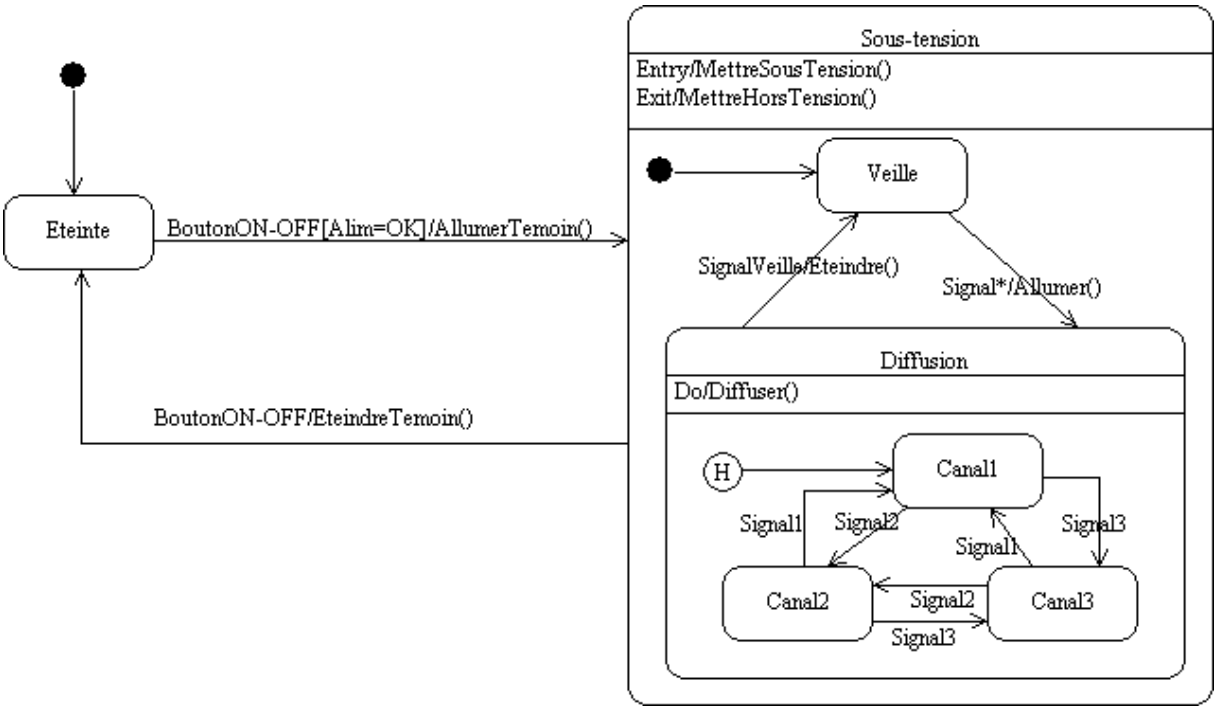


Figure n°32 : ***Décomposition d'états en sous-états***

Dans ce diagramme, on traduit le fait que lorsque la télévision est en veille, lorsqu'on sera amené à la rallumer, elle diffusera automatiquement celle des trois chaînes qui était activée lors de la dernière mise en veille.

5.5.4.3 Parallélisme

Dans un diagramme d'états-transitions on peut être amené à décrire deux sous-diagrammes qui évoluent en parallèle dans un super-état. Ces sous-états permettent de préciser plusieurs automates à états-finis qui fonctionnent de manière synchronisée : les états finaux doivent tous être activés pour que le super-état ait terminé son activité.

La figure suivante présente l'intégration du parallélisme pour la classe *CombinéTélévisionMagnétoscope* dans un super-état (*Sous-Tension*) illustrant le principe selon lequel l'appareil peut à la fois jouer son rôle de téléviseur classique et garantir en parallèle la programmation d'un enregistrement.

On ne peut alors mettre l'appareil hors-tension tant que si l'enregistrement est terminé (présence d'un état final), par contre, la diffusion n'entraîne pas de contrainte sur cette mise hors tension (pas de présence d'état final). Ce principe de parallélisme se retrouve sur la figure n°26 (visuel crée avec *Objecteering UML Modeler*) :

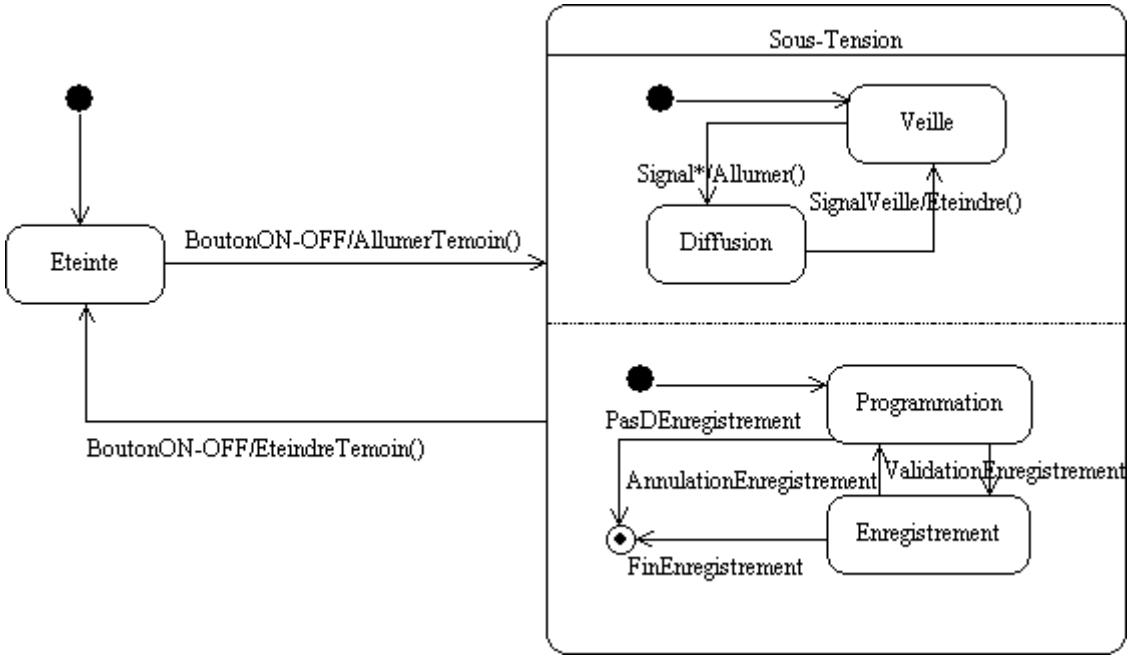


Figure n°33 : ***Parallélisme au sein d'un super-état***

Il est important d'identifier clairement qu'il s'agit de la description parallèle de deux comportements avant d'utiliser les concepts de parallélisme : on est en effet souvent tenté d'avoir recours à ce type de principe pour décrire la présence de plusieurs actions au sein d'un état (alors qu'il ne s'agit que d'une décomposition de l'activité interne de l'état).

6. Modèle Architectural

Les diagrammes de composants et de déploiement composent la vue architecturale du formalisme UML. Ces diagrammes permettent de décrire la dynamique interne du sujet. En pratique, on choisit généralement de représenter le comportement du système en utilisant un sous-ensemble de ces quatre diagrammes car s'ils décrivent des aspects complémentaires du fonctionnement, les recouvrements et redondances sont nombreux. Il s'agit donc de choisir le ou les diagrammes les plus pertinents vis à vis de l'objectif de modélisation.

6.1 Diagramme de composants (Components Diagram)

Le diagramme de composant entre directement dans la description physique (et statique) du sujet. Il s'attache à concrétiser les considérations logiques et abstraites vers le monde réel.

Les diagrammes de composants décrivent les éléments physiques concrétisant le système et leurs relations

C'est à partir de ce type de diagramme qu'on détermine les choix de réalisation (on va définir quels constituants techniques vont réaliser physiquement le système). Ces composants constituent la finalisation de certains éléments logiques (objets ou classes).

6.1.1 Le Composant

Un composant est une représentation d'une partie physique du système (éventuellement remplaçable) qui reprend et instancie un regroupement d'éléments logiques (il constitue l'emballage « tangible » des objets).

Un composant est un élément, généralement stéréotypé, qui participe à l'exécution réelle du système

Les composants peuvent être stéréotypés selon diverses natures :

- *executable* : composant pouvant être exécuté,
- *library* : composant implémentant une bibliothèque objet statique ou dynamique,
- *table* : composant représentant une table de base de données,
- *file*: composant assimilable à un document contenant un code source ou des données,
- *document* : composant implémentant un document général.

Chacun de ces stéréotypes offre une représentation graphique spécifique. Néanmoins, *Objecteering UML Modeler* ne propose pas ces symboles particuliers et se contente de certains stéréotypes. La figure suivante contient ainsi une représentation des symboles spécifiques et certaines équivalences obtenues avec *Objecteering UML Modeler* :

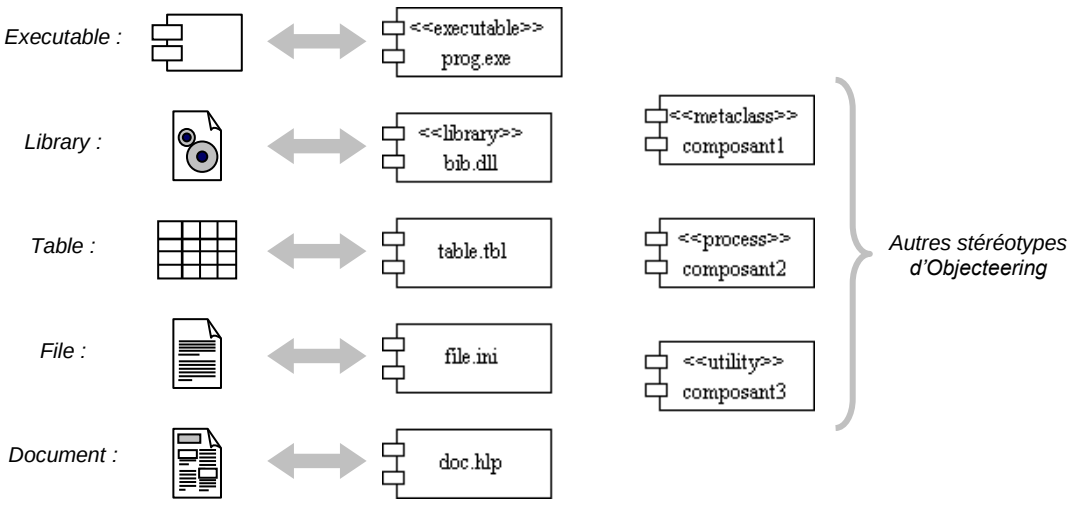


Figure n°34 : ***Les stéréotypes de composants (formalisme UML et outil Objecteering UML Modeler)***

Outre cette typologie des composants, on distingue un second mode de classification relatif au niveau d'intervention des composants. Cette approche permet de distinguer trois types de composants qui sont théoriquement à même de rassembler des composants issus des cinq types présentés précédemment (executable, library, table, file, document) :

- *composants de déploiement* : les composants nécessaires et suffisants pour former un système exécutable. Ces composants regroupent globalement les exécutables (*executable*), les bibliothèques dynamiques (*library*),
- *composants produits du déploiement* : ces composants résultent du processus de développement. Ils se composent des fichiers code source et des fichiers de données (*file*),
- *composants d'exécution* : ces composants résultent quant à eux d'une exécution du système. Ce sont les produits du fonctionnement du sujet.

6.1.2 Structuration du diagramme de composants

Un premier moyen d'organisation des composants relève de la possibilité de les regrouper dans des *paquetages*. De plus, les composants peuvent être connectés entre eux par le biais des relations « classiques » (*dépendance*, *généralisation*, *association*, *agrégation*) ou reliés aux classes, en particulier celles qu'ils « réalisent » physiquement par le biais de relations dites de *réalisation*.

En pratique, les relations qui sont les plus usitées lors de la réalisation d'un diagramme de composants sont celles de *dépendance* et en moindre proportion, celles de *réalisation* (lorsqu'on veut lier le diagramme de composants à des classes). Cette relation de dépendance relève en général plus d'une convention d'écriture qui signifie que le composant « origine de la dépendance » importe les interfaces du composant « destination de la dépendance ».

Un exemple de diagramme de composants représentant une application de gestion de commandes peut être représenté à l'aide de la figure suivante (visuel crée avec *Objecteering UML Modeler*) :

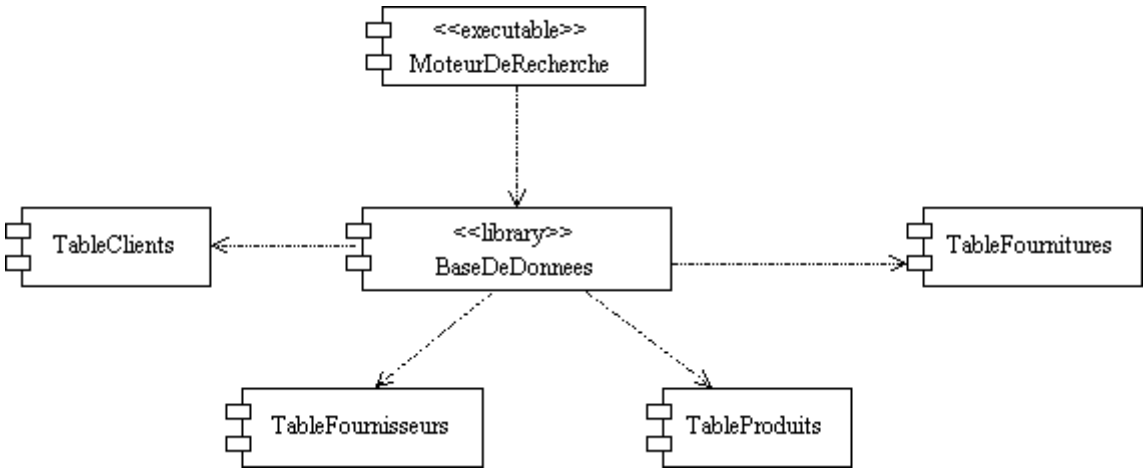


Figure n°35 : ***Un exemple de diagramme de composants***

Une notion importante relative au diagramme de composants est qu'il permet de représenter quels éléments logiciels ou matériels vont implémenter la vue logique du système mais qu'ils ne permettent de visualiser l'architecture physique (en terme de machines et de choix technologiques) qui va être réalisée.

6.2 Diagramme de déploiement (Deployment Diagram)

Le diagramme de déploiement entre dans la vue physique (et statique) du sujet. Il vise à représenter les choix technologiques et matériels finaux sur lesquels seront positionnés les composants.

Les diagrammes de déploiement représentent la configuration des nœuds d'exécution et des composants qui résident sur ces nœuds

C'est la topologie du matériel qui est abordé avec ce type de diagramme. Leur intervention est donc extrêmement tardive et vise essentiellement la phase de développement (même si on peut également placer cette étape en amont afin de conduire un travail de dimensionnement).

Si Le formalisme UML n'est pas restreint globalement dans son utilisation au seul domaine du génie logiciel, les diagrammes de déploiement sont pour leur part exclusivement dédiés à la représentation physique de systèmes informatisés.

6.2.1 Le Nœud

Les nœuds sont les éléments de base des diagrammes de déploiement. Il s'agit d'éléments physiques qui existent au moment de l'exécution et représente une ressource (mémoire, capacité de calcul ou de traitement). Les nœuds représentent généralement les processeurs ou les périphériques.

Si les composants représentent le regroupement et l'implémentation physiques de constituants logiques, les nœuds décrivent le matériel sur lequel ces composants sont déployés et exécutés.

La désignation d'un nœud se fait à l'aide d'un *nom* issu du vocabulaire de l'implémentation. Un nœud se représente comme indiqué sur la figure suivante (visuel crée avec *Objecteering UML Modeler*) :

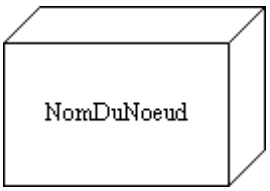


Figure n°36 : ***Représentation d'un nœud***

On peut noter que la représentation d'un nœud peut s'accompagner de compléments similaires à ceux qu'on fournirait pour décrire les caractéristiques d'une classe (attributs et méthodes). On peut, par exemple, préciser qu'un *Processeur* possède les attributs *vitesseProcesseur* et *mémoire* ainsi que les opérations *allumer( ), éteindre( ),* et *suspendre( )*. Cependant, cette fonctionnalité n'est pas prévue par la version d'*Objecteering UML Modeler* utilisée pour la réalisation de ce document.

6.2.2 Structuration du diagramme de déploiement

Les nœuds peuvent être organisés à l'aide de *paquetages* (tout comme les classes et les composants). Ils peuvent également être reliés par les relations de *généralisation*, *association*, *agrégation* et *dépendance*. On peut en particulier expliciter la relation entre un nœud et le composant qu'il déploie en positionnant une relation de dépendance du nœud vers le composant.

Cependant, en pratique, la relation la plus utilisée est l'*association* entre nœuds. Cette relation représente alors une connexion physique entre les composants (connexion ethernet, ligne série, bus partagé, liaison satellite, etc.). Un exemple de diagramme de déploiement est donné par la figure suivante (visuel créé avec *Objecteering UML Modeler*) :

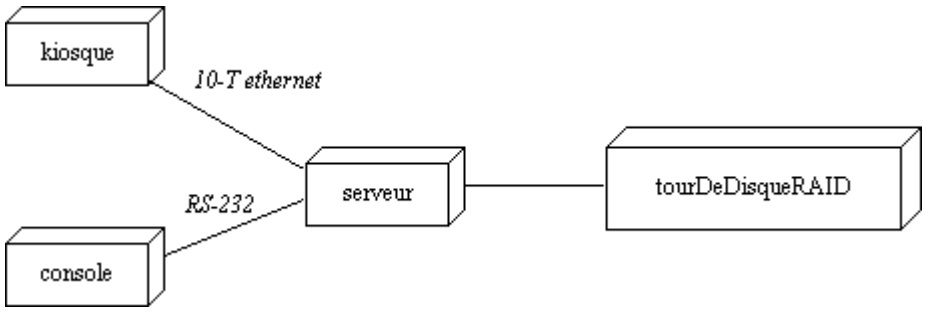


Figure n°37 : ***Un exemple de diagramme de déploiement***

7. Utilisation d'UML

« *UML n'est pas une méthode, c'est un langage de modélisation* », nous l'avons répété tout au long de ce document ; et la présentation d'UML qui a été faite s'est essentiellement attachée à en donner une vision permettant d'en appréhender concrètement les concepts.

L'objectif de ce document est donc de permettre au lecteur d'acquérir une connaissance de l'ensemble du formalisme UML suffisante pour le mettre en position d'organiser lui-même son utilisation de ce langage de modélisation en fonction de ses objectifs de modélisation.

Néanmoins, certains principes de modélisation ou certaines approches méthodologiques existent (de manière indépendante et généralement postérieure à la création du langage UML) et il est évidemment intéressant de les évoquer à l'issue de cette présentation d'UML.

7.1 Principes de modélisation

Comme nous l'avons déjà évoqué, la modélisation est une activité ancestrale indissociable du fonctionnement mental de l'être humain. Cet acquis historique nous permet de rappeler quatre grands principes de modélisation dictés par l'expérience (cf. « *Le Guide de l'utilisateur UML* » – *G. Booch, J. Rumbaugh, I. Jacobson*). Ces principes, globalement indépendants du formalisme de modélisation adopté, s'adaptent tout à fait à la modélisation orientée objet avec UML.

7.1.1 Premier principe de modélisation

Le choix des modèles à créer a une forte influence sur la manière d'aborder un problème et sur la nature de sa solution

Les modèles doivent être choisis avec soin car, s'ils sont pertinents, ils feront ressortir les problèmes de développement inhérents au sujet et apporteront par là-même un éclaircissement et de précieuses indications quant à la façon de les résoudre.

7.1.2 Deuxième principe de modélisation

Tous les modèles peuvent avoir différents niveaux de précision

Les différents utilisateurs d'un modèle (client, analyste, développeur…) de même que les différents stades du cycle de vie exigent de pouvoir visualiser le système avec une précision variable.

7.1.3 Troisième principe de modélisation

Les meilleurs modèles ne perdent pas le sens de la réalité

Une lacune courante des techniques d'analyse structurée réside dans le décalage fondamental entre le modèle analytique et le modèle conceptuel d'un système (i.e. les modèles relatifs aux phases d'analyse et de conception). Il est en effet indispensable de connaître les conventions de simplification et l'écart à la réalité qui accompagnent la construction du modèle de conception.

7.1.4 Quatrième principe de modélisation

Parce qu'aucun modèle n'est suffisant à lui seul, il est préférable de décomposer Un système important en un ensemble de sous-modèles presque indépendants

La cohérence de l'ensemble des différents points de vue d'un modèle garantit l'efficacité de la modélisation. La quasi-indépendance des différents modèles (couvrant les différents angles d'appréhension du système) permet leur développement séparé tout en assurant leur homogénéité et leur cohérence globale.

7.2 Rational Unified Process (RUP)

Un processus est un ensemble d'étapes plus ou moins ordonnées, destinées à atteindre un objectif. Concernant les objectifs d'un processus d'ingénierie logicielle, il s'agit de livrer un produit logiciel qui réponde aux besoins exprimés.

UML est largement indépendant des processus de conception, ce formalisme peut ainsi être utilisé avec de nombreux processus d'ingénierie logicielle. Le RUP (Rational Unified Process) est l'une de ces approches de cycle de vie particulièrement adaptée au langage UML.

7.2.1 Caractéristiques du processus RUP

Le RUP est un processus itératif particulièrement adapté au traitement des projets actuels. Leur complexité et leur degré de sophistication se prêtent particulièrement à une approche de ce type offrant la flexibilité suffisante pour réagir à l'introduction de nouvelles exigences ou de changement d'objectifs commerciaux. Ce processus est ainsi qualifié de configurable.

Les activités du RUP sont orientées vers la création et la maintenance de modèles plutôt que de documents papier. Ce processus est centré sur l'*architecture* : il s'agit de concentrer l'initiation du cycle sur la détermination de lignes de base de l'architecture logicielle. Ce plan architectural est une base solide facilitant le développement en parallèle, qui réduit le travail de révision et augmente la réutilisabilité et la maintenabilité du système.

Ce processus est piloté par les cas d'utilisation et met fortement l'accent sur la construction de systèmes basés sur une compréhension approfondie de la façon dont le système sera utilisé. C'est donc une conception guidée par le besoin et les scénarios.

7.2.2 La structure itérative du RUP

Le RUP s'appuie sur quatre concepts distincts qui sont les *cycles d'évolution*, les *phases*, les *itérations* et les *activités*. Il est important de cerner de quelle manière ces éléments du RUP sont agencés.

Le cycle de développement d'un système repose sur un premier *cycle d'évolution* appelé *cycle de développement initial*. Ce premier *cycle d'évolution* conduira à la première version du système et pourra être suivi d'autres *cycles d'évolution* (= *cycles de développement des versions ultérieures*).

Chacun de ces *cycles d'évolution* se décompose en quatre *phases*. Une *phase* est l'intervalle temporel séparant deux jalons importants du processus :

1. *Création* (ou *Inception*) : définition du cadre du projet, étude d'opportunité,
2. *Elaboration* : établissement du plan du projet et d'une architecture solide,
3. *Construction* : développement du système en version béta,
4. *Transition* : livraison du système aux utilisateurs finaux.

Chacune de ces *phases* se décompose en *itérations* (en général entre 1 et 3) qui correspondent à la conduite d'un ou plusieurs cycle(s) d'*activités*. En fonction de la *phase* dans laquelle on se trouve, l'*itération* (et donc le *cycle d'activités*) sera plus ou moins consacrée à certaines des *activités* du *cycle d'activités*. Les *activités* (appelée aussi *workflow* selon les descriptions du RUP) présentes dans chacun des *cycle d'activités* ainsi que les **attendus** (en particulier en terme de délivrables UML) de chacune d'elles sont les suivantes :

1. Gestion de projet : planification, allocation (tâches et ressources), étude (faisabilité et risques)
   → calendrier du projet, diagramme de GANTT (par le chef de projet)
2. Modélisation métier : modélisation (structure et fonctionnement)
   → cas d'utilisation de l'organisation (par le concepteur d'organisation)
3. Analyse des besoins : exigences (fonctionnelles et non-fonctionnelles)
   → cas d'utilisation du système, descriptif de l'interface (par l'analyste)
4. Analyse et conception : description de la solution théorique répondant aux besoins
   → diagrammes de classes, de collaboration, d'états-transitions, de composants (par l'architecte et/ou le concepteur)
5. Implémentation : transcription en programme (utilisation de composant existants)
   → code (par les développeurs)

6. Test : déroulement du jeu de test, limites
   → diagnostics (par le testeur)
7. Déploiement : distribution du logiciel dans son environnement opérationnel
   → diagramme de déploiement
8. Maintenance/Evolution : gestion des évolutions (pendant l'avancement du projet)
   → plan de modification
9. Environnement : support du développement (sélection des outils, administration système…)
   → documentation

Ces *activités* s'agencent, dans le *cycle d'activités* relatif à chacune des *itérations* selon le principe suivant :

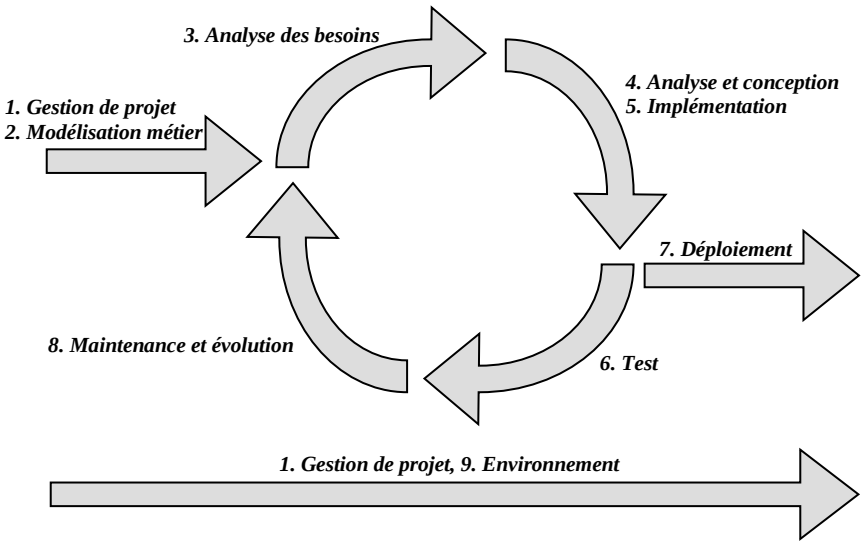


Figure n°38 : ***Un cycle d'évolution d'une itération***

A chaque itération, les résultats (planification, modèles, prototypes, jeux de tests) de l'itération précédente sont réutilisés, évalués, corrigés, améliorés. La conséquence naturelle à ce phénomène est que chacune des activités est plus ou moins présente dans un cycle d'activité selon la phase dans laquelle on se trouve. La figure suivante illustre ce principe d'implication quantitative des activités (ce qui permet donc de déduire la nature des cycles d'activités) dans les phases :

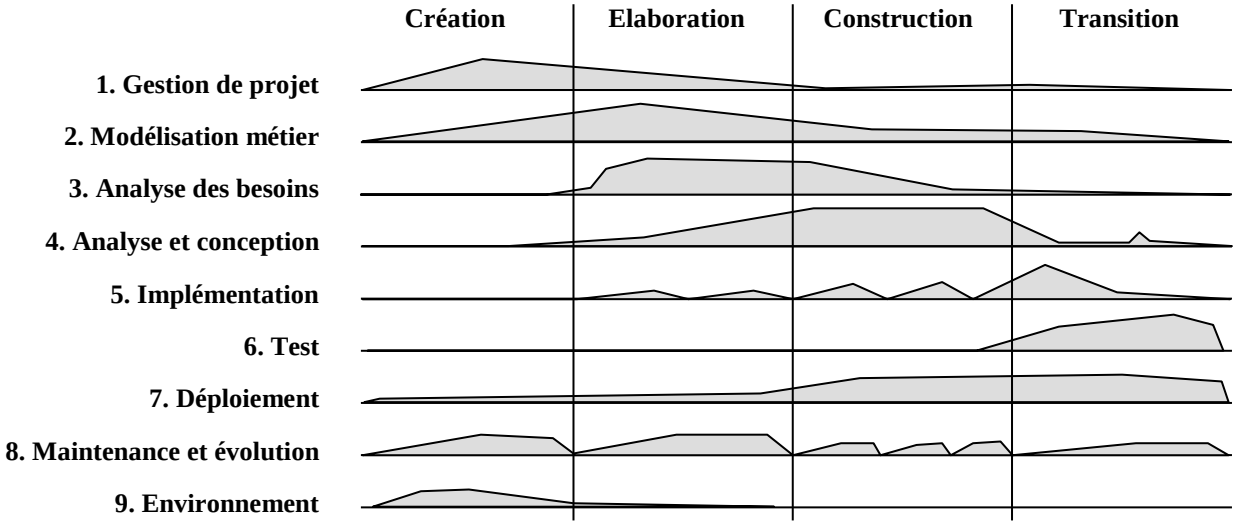


Figure n°39 : ***Implication des activités en fonction des phases***

Le *RUP* est donc un *processus de conception* basé sur un *cycle de développement* (résultant éventuellement de *cycles d'évolution*) composé de quatre *phases* regroupant une ou plusieurs itérations du *cycle d'activité* constitué de *workflows de processus* (*activités*).

8. Bibliographie

Les ouvrages et documents suivants ont permis d'alimenter ce cours :

- ***Le guide de l'utilisateur UML*** *Grady Booch, James Rumbaugh, Ivar Jacobson*

  Ouvrage : EYROLLES – ISBN 2-212-09103-6

- ***UML en action*** *Pascal Roques, Franck Vallée*

  Ouvrage : EYROLLES – ISBN 2-212-11213-0

- ***UML 2 en action*** *Pascal Roques, Franck Vallée*

  Ouvrage : EYROLLES – ISBN 2-212-11462-1

- ***UML pour l'analyse d'un système d'information*** *Chantal Morley, Jean Hugues, Bernard Leblanc*

  Ouvrage : DUNOD – ISBN 2-100-04826-0

- ***UML et RUP : un survol*** *Thérèse Libourel, Marianne Huchard*

  Cours DEA Informatique LIRMM (Université de Montpellier II)

- ***Unified Modeling Language*** *J.-P. Bourey*

  Cours Ecole Centrale Lille

- ***UML 2 par la pratique*** *Pascal Roques*

  Ouvrage : EYROLLES – ISBN 2-212-11480-X

- ***UML 2.0 – Guide de référence*** *Grady Booch, James Rumbaugh, Ivar Jacobson*

  Ouvrage : EYROLLES – ISBN 2-7440-1820-1