1. Introduction

1.1 Définition du GL

Le Génie Logiciel (*Software Engineering*) est un domaine des sciences de l'ingénieur dont la finalité est la conception, la fabrication et la maintenance de systèmes logiciels complexes, sûrs et de qualité. *C'est un ensemble de méthodes, techniques et outils pour la production et la maintenance de composants logiciels de qualité.*

Contrairement au développement artisanal (production individuelle d'un système simple), dans le cas du GL, il s'agit d'une production collective d'un système complexe concrétisée par un ensemble de documents de conception, de programmes et de jeux de tests avec souvent de multiples versions (*multi-person construction of multi-version software*).

1.2 Problématique

Compte tenu des évolutions des techniques de programmation, du matériel et des besoins, plusieurs problèmes ont confronté le développement de logiciels dont les plus importants sont :

- Taille et complexité des systèmes logiciels de plus en plus accrues :
  - Besoins et fonctionnalités qui augmentent et évoluent sans arrêt
  - Evolution du matériel
  - Diversification des architectures
- Délais de réalisation courts
- Développement collectif (différents intervenants au développement avec des compétences multiples).

Les difficultés liées à la nature du logiciel

- un logiciel ne s'use pas, sa fiabilité ne dépend que de sa conception
- mais, pour rester utilisé, un logiciel doit évoluer
- pas de direction clairement exprimée
- changements fréquents
- contradictions des besoins, etc.

Les difficultés liées aux personnes

- ne savent pas toujours ce qu'elles veulent, ou ne savent pas bien l'exprimer
- communication difficile entre personnes de métiers différents

Les difficultés technologiques

- courte durée de vie du matériel
- beaucoup de méthodes et de langages
- évolution des outils de développement, etc.

1.3 Objectifs du GL – La règle du CQFD

Le GL se préoccupe des procédés de fabrication de logiciels de façon à s'assurer que les quatre critères suivants soient satisfaits :

- ☞ Le système développé doit assurer les **fonctionnalités** attendues ;
- ☞ Les **coûts** du développement doivent rester dans les limites prévues au départ ;
- ☞ Les **délais** doivent rester dans les limites prévues au départ ;
- ☞ La **qualité** du logiciel correspond au contrat de service initial. C'est une notion multiforme qui recouvre :
  - *Fiabilité* : capacité d'un logiciel à assurer de manière continue le service attendu,
  - *Correction (validité)* : aptitude d'un logiciel à réaliser exactement les tâches telles qu'elles ont été définies par sa spécification,
  - *Robustesse* : aptitude d'un logiciel à fonctionner même dans des conditions anormales,
  - *Extensibilité* : facilité d'adaptation d'un logiciel aux changements de spécification,
  - *Réutilisabilité* : aptitude d'un logiciel à être réutilisé en tout ou partie,
  - *Compatibilité* : aptitude des logiciels à être combinés les uns aux autres,
  - *Efficacité* : capacité d'un logiciel à bien utiliser le minimum des ressources matérielles (mémoire, puissance de l'U.C., etc.),
  - *Portabilité* : facilité à être porté sur différents environnements matériels et/ou logiciels,
  - *Traçabilité* : capacité à identifier et/ou suivre un élément du cahier des charges lié à un composant logiciel,
  - *Vérifiabilité* : aptitude d'un logiciel à être testé (optimisation de la préparation et de la vérification des jeux d'essai)
  - *Intégrité* : aptitude d'un logiciel à protéger ses composants contre des accès ou des modifications non autorisés,
  - *Autres qualités* : facilité d'utilisation, réparabilité, etc.

Ces qualités sont parfois *contradictoires* (chic et pas cher!). Il faut les *pondérer* selon le type du logiciel (critique/grand public, systèmes sur mesure/produits logiciels de grande diffusion, etc.).

1.4 Objectifs de la première partie du cours

Couvrir le domaine de la production de logiciels

- mettre en évidence les besoins
- aspects organisationnels
  - cycles de vie
  - démarches
- aspects techniques
  - méthodes
  - spécification
  - *design patterns*
  - qualité, test.

2. Cycles de vie de logiciels

2.1 Introduction

Le cycle de vie du logiciel (ou processus de développement) est un ensemble cohérent d'activités pour spécifier, concevoir, implémenter et tester des systèmes logiciels. Il y a alors différents livrables (modèles d'analyse, codes sources, manuels d'utilisation, etc.) associés chaque activité.

2.1.1 Activités

a. - *Etude de faisabilité* : déterminer si le développement proposé est réalisable.

   - *Analyse du marché* : déterminer s'il y a un marché potentiel pour ce produit.

b. - *Expression des besoins* : déterminer quelles sont les fonctionnalités que le logiciel doit offrir.

   - *Recueil des besoins* : obtenir les besoins à partir des utilisateurs.

   - *Analyse du domaine* : déterminer quelles sont les tâches et les structures qui sont communes à ce problème.

c. - *Planification du projet* : déterminer comment développer le logiciel.

   - *Analyse des coûts* : déterminer les estimations du coût.

   - *Assurance qualité* : déterminer les activités qui permettront d'assurer la qualité du produit pour garantir la satisfaction du client (selon les objectifs contractuels).

   - *Structure work-breakdown* : déterminer les sous-tâches nécessaires pour développer le produit.

d. - *Conception* : déterminer comment le logiciel doit fournir la fonctionnalité.

   - *Conception architecturale* : concevoir la structure du système.

   - *Conception d'interfaces* : spécifier les interfaces entres les différentes parties du système.

   - *Conception détaillée* : concevoir les algorithmes pour les parties individuelles.

e. - *Implémentation* : construire le logiciel.

f. - *Test* : exécuter le logiciel avec des données pour permettre de s'assurer que le logiciel opère correctement.

   - *Test unitaire* : tester par le développeur original.

   - *Test d'intégration* : tester durant l'intégration du logiciel.

   - *Test du système* : tester le logiciel dans un environnement.

   - *Test d'acceptation* : tester pour satisfaire le client.

g. - *Livraison* : fournir au client une solution logicielle efficace.

   - *Installation* : mettre le logiciel disponible dans le site opérationnel du client.

   - *Formation* : former les utilisateurs à utiliser le logiciel et répondre à leurs questions.

h. - *Maintenance* : mettre-à-jour et améliorer le logiciel pour garantir une utilisation efficace continue.

2.1.2 Documents

Les résultats des différentes activités sont représentés par plusieurs types de documents, dont les plus importants sont :

- *Spécification des besoins logiciels* : décrit ce que doit faire le logiciel.

- *Modèle d'objet* : montre les principaux objets / classes.

- *scénarios de cas d'utilisation* : montrent les séquences des comportements possibles du point de vue utilisateur.

- *Plan du projet* : décrit l'ordre des tâches et estime les besoins en matière *temps* et *efforts*.

- *Plan de test du logiciel* : décrit comment le logiciel serait testé pour garantir un comportement correct.

- *Conception du logiciel* : décrit la structure du logiciel

- *Plan assurance qualité du logiciel* : décrit les activités à effectuer pour assurer la qualité.

- *Manuel d'utilisation* : décrit comment utiliser le logiciel final.

- *Code source* : le code du produit final.

- *Rapport du test* : décrit quels sont les tests effectués et quel était le comportement du système.

2.2 Modèles de cycles de vie

Un modèle d'un processus de développement est une représentation abstraite d'un processus. Il présente la description du processus d'une perspective particulière. Nous avons quatre principaux modèles de cycles de vie de logiciel :

2.2.1 Modèle séquentiel linéaire

Appelé aussi le modèle en cascade (*Waterfall model*) du fait que le diagramme ressemble à des séries de cascades. C'est un modèle adapté seulement quand tous les besoins sont bien déterminés à priori.

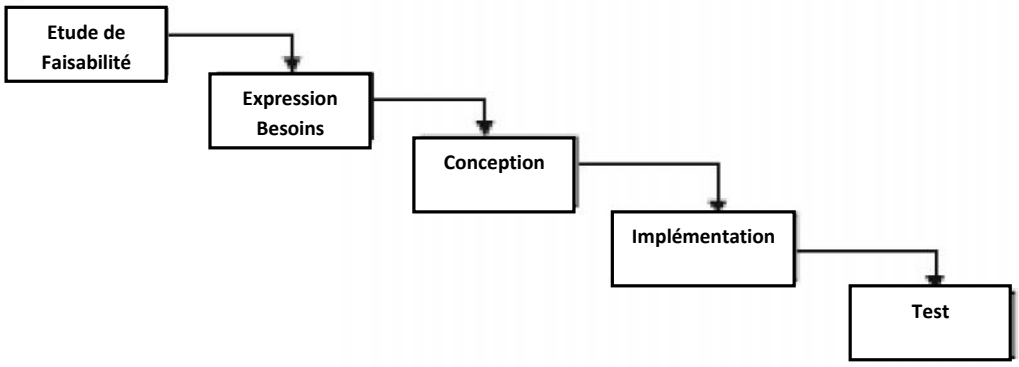


Figure 1. Modèle en cascade.

Le modèle d'origine a été décrit initialement par Royce (1970), mais actuellement il y a plusieurs versions qui peuvent mettre l'accent sur certaines activités et négliger d'autres selon le besoins. Dans la version présentée par la figure 1, les activités de *planification du projet* sont inclues dans la phase expression des besoins. D'autre part, les phases *livraison* et *maintenance* ont été négligées dans cette version du modèle.

L'une des variantes les plus importantes de ce modèle et qui a été même considérée comme étant un modèle à part entière est le modèle en V (figure 2).

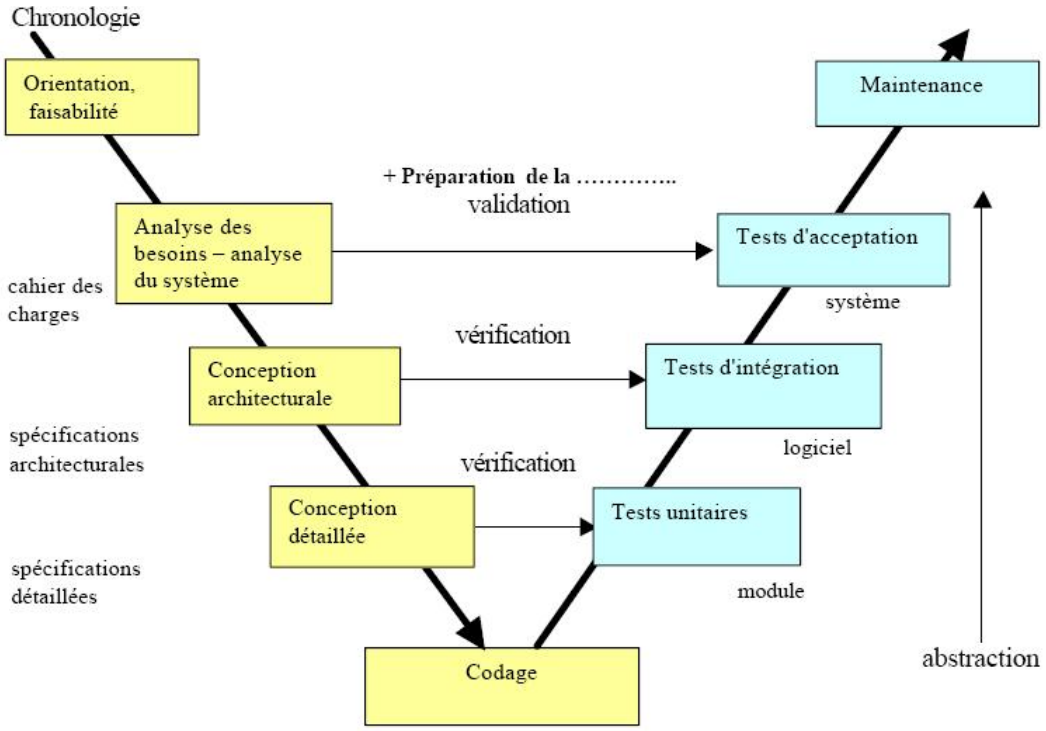


Figure 2. Modèle en V.

2.2.2 Modèle de prototypage

Ce modèle de cycles de vie construit un prototype pour tester les concepts et les besoins. Après accord du client, le développement du logiciel se poursuit en passant toujours par les mêmes phases du modèle précédent.

2.2.3 Modèle incrémental

Proposé par D. L. Parnas (1979) pour concevoir et livrer au client un sous-ensemble opérationnel du système global. Le processus continue d'itérer, comme le montre la figure 3, à travers le cycle de vie global avec des incréments additionnels.

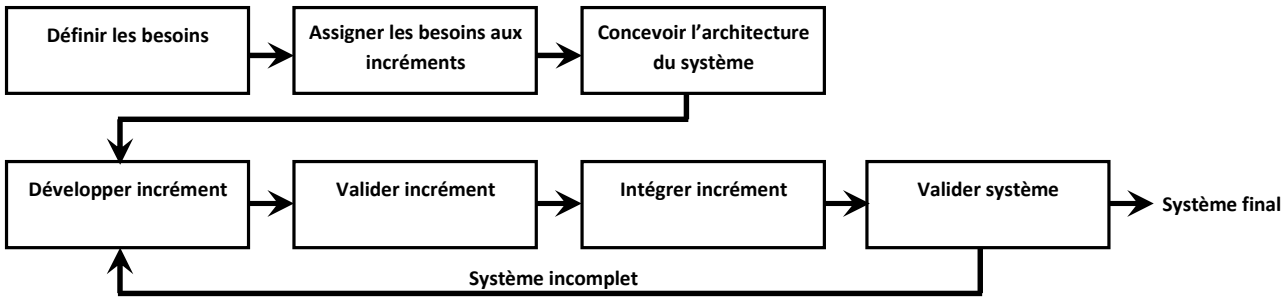


Figure 3. Modèle incrémental.

2.2.4 Modèle en spirale

C'est un modèle qui a été introduit par B. Boehm (1988). L'image du modèle est une spirale qui commence au milieu et qui réitère continuellement les tâches de base (figure 4).

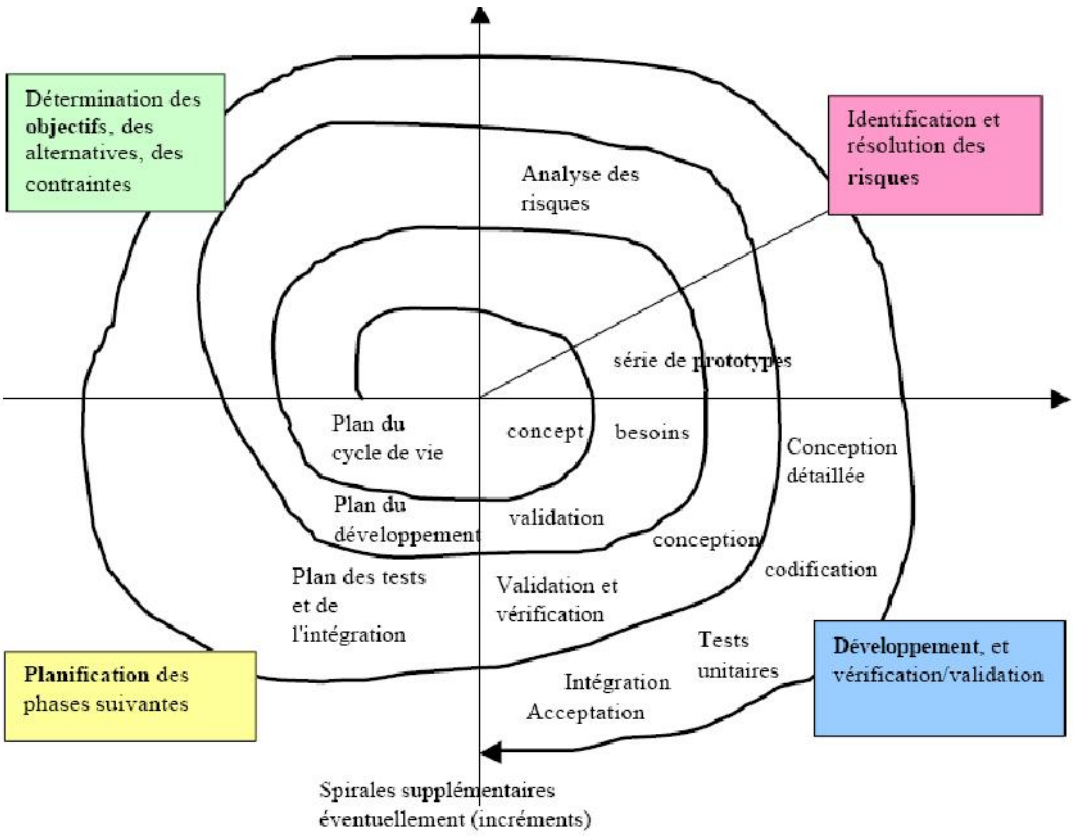


Figure 4. Modèle en spirale.

Les principaux risques et leurs remèdes, tels que définis par Boehm, sont les suivants :

- Défaillance de personnel : embauches de haut niveau, formation mutuelle, leaders, adéquation profil/fonction, ...
- Calendrier et budgets irréalistes : estimation détaillée, développement incrémental, réutilisation, élagage des besoins, ...
- Développement de fonctions inappropriées : revues d'utilisateurs, manuel d'utilisation précoce, ...
- Développement d'interfaces utilisateurs inappropriées : maquettage, analyse des tâches, ...
- Produit "*plaqué or*" : analyse des coûts/bénéfices, conception tenant compte des coûts, ...
- Volatilité des besoins : développement incrémental de la partie la plus stable d'abord, masquage d'information, ...
- Problèmes de performances : simulations, modélisations, essais et mesures, maquettage,
- Exigences démesurées par rapport à la technologie : analyses techniques de faisabilité, maquettage, ...
- Tâches ou composants externes défaillants : audit des sous-traitants, contrats, revues, analyse de compatibilité, essais et mesures, ...

2.2.5 Autres modèles

Il y a plusieurs autres modèles ou variantes de modèles déjà existants tels que le modèle de développement des systèmes formels, développement basé réutilisation, programmation extrême, etc.

3 Méthodes d'analyse et de conception

3.1 Introduction

Les méthodes d'analyse et de conception fournissent des *notations* standards et des *démarches* pratiques pour aboutir à des conceptions appropriées.

3.2 Classification des méthodes

Il existe différentes manières pour classer ces méthodes, dont :

- la distinction *composition / décomposition* : les méthodes peuvent être *ascendantes* et consistent à construire un logiciel par composition de modules existants ou, *descendantes* et décomposent récursivement un système jusqu'à arriver à des modules simples ;
- la distinction *fonctionnel / objet*. Dans un raisonnement fonctionnel, le système est considéré comme un ensemble d'unités fonctionnelles en interaction. Les fonctions disposent d'un état local, mais le système a un état centralisé partageable par l'ensemble des unités fonctionnelles. Par contre, un raisonnement objet considère qu'un système est un ensemble d'objets en interaction. Chaque objet dispose d'un ensemble d'attributs décrivant son état et l'état du système est décrit (de façon décentralisé) par l'état de l'ensemble des objets constituant ce système.

3.2.1 Méthodes fonctionnelles (années 60)

Basées sur une décomposition fonctionnelle du système (inspirée de l'architecture des ordinateurs), les méthodes fonctionnelles 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 dont le plus haut niveau représente l'ensemble du problème. Chaque niveau est ensuite décomposé en respectant les entrées/sorties du niveau supérieur. La décomposition se poursuit jusqu'à arriver à des composants simples et maîtrisables.

En plus des problèmes rencontrés avec les langages procéduraux, ce style de raisonnement dissimule également une sérieuse limite touchant directement la stabilité de l'architecture du système. En fait, cette dernière est basée principalement sur des fonctionnalités qui peuvent être sujet de modifications ou d'évolutions.

Parmi ces méthodes, nous pouvons citer SA (*Structured Analysis*), SART (*Structured Analysis for Real Time systems*), SADT (*Structured Analysis and Design technique*), etc.

3.2.2 Méthodes systémiques (années 70)

Une variante des méthodes fonctionnelles qui sépare complètement les données des traitements. Une étape de validation du système de données par celui des traitements est alors nécessaire. Ce type de méthodes est très bien adapté pour les systèmes d'information d'entreprise.

Exemple : MERISE (*Méthode d'Etude et de Réalisation Informatique des Systèmes d'Entreprises*), etc.

3.2.3 Méthodes Objet (années 80)

L'approche objet propose une méthode de décomposition (*décomposer pour réunir*) basée sur l'intégration de ce que le système est (*structure*) et fait (*fonction*). Le système est constitué d'un ensemble d'objets en interaction (par échange de messages) pour réaliser les fonctionnalités attendues. L'architecture du système est alors basée sur la partie statique qui est plus stable.

Le principe de l'orientation objet étant basé sur l'identification et l'organisation des concepts du *domaine d'application*, plutôt que leur représentation terminale dans un langage de programmation qu'il soit orienté objet ou non. Ce processus est un style de raisonnement et non pas une technique de programmation, entre autres, il est indépendant des langages de programmation jusqu'aux derniers stades. Il se concentre sur la *modélisation* et non pas sur l'implantation des concepts, ce qui permet de bien comprendre et organiser les concepts inhérents à l'application avant de chercher une implantation efficace des structures de données et des algorithmes. Aussi, en plus de préparer la programmation, la modélisation peut servir de support pour la documentation et l'interface avec le client.

Les principaux avantages des ces méthodes peuvent être :

- ✓ stabilité de la modélisation par rapport aux entités du monde réel,
- ✓ construction itérative facilitée par le couplage faible entre composants,
- ✓ possibilités de réutiliser des éléments d'un développement à un autre,
- ✓ …

Cependant, les méthodes objet restent encore récentes et trouvent toujours des difficultés dans le développement de systèmes critiques, temps réel, ou encore embarqués qui nécessitent des méthodes rigoureuses permettant d'accomplir une vérification formelle.

Exemple : OMT (*Object Modeling Technique*), Booch'93 (le nom de son auteur), OOD (*Object Oriented Design*), OOSE (*Object Oriented Software Engineering*), etc.

3.3 Spécification

Tout produit complexe à construire doit d'abord être spécifié ; par exemple *un pont de 30 mètres de long, supportant au moins 1000 tonnes, construit en béton, etc.* ces spécifications peuvent être considérées comme un contrat entre le client (la collectivité qui veut réaliser le pont) et le producteur (l'entreprise de génie civil).

En informatique, le client et le producteur peuvent être différents selon les phases du cycle de vie du logiciel :

- Spécification des besoins ou spécification des exigences (*requirements specification*) : c'est un contrat entre les futurs utilisateurs et les concepteurs. Elle concerne les caractéristiques attendues (exigences fonctionnelles et non fonctionnelles : efficacité, sûreté, portabilité, taille, etc.). Elle intervient en *phase d'analyse des besoins* et se rédige en langue naturelle.
- Spécification du système : c'est un contrat entre les futurs utilisateurs et les concepteurs et concerne la nature des fonctions offertes, les comportements souhaités, les données nécessaires, etc. Elle intervient pendant *la phase d'analyse du système*.
- Spécification de l'architecture du système : c'est un contrat entre les concepteurs et les réalisateurs et définit l'architecture en modules de l'application à réaliser. Elle intervient pendant *la phase de conception générale*.
- Spécification technique (d'un module, d'un programme, d'une structure de données, etc.) : c'est un contrat entre le programmeur qui l'implémente et les programmeurs qui l'utilisent. Elle intervient pendant *la phase de conception détaillée*.

De manière générale, une spécification décrit les caractéristiques attendues (le quoi) d'une implémentation (le comment).

- ☞ Il est souhaitable qu'une spécification soit claire, non ambiguë et compréhensible,
- ☞ Les spécifications du langage naturel manquent souvent de précision,
- ☞ Les spécifications doivent aussi être cohérentes et complètes. Ici, la complétude peut prendre deux formes :

  - *Interne* : tous les concepts utilisés sont complètement spécifiés,
  - *Externes* : complétude de la spécification par rapport à la réalité décrite. C'est une forme quelque peu illusoire en pratique ; on ne peut pas en général spécifier tous les détails qui entourent le système.

3.3.1 Classification des styles de spécification

Il y deux critères de classification orthogonaux :

- *Formalité* : on distingue les spécifications informelles (langage naturel), semi-formelles (sémantique plus ou moins précise – souvent graphique), et formelles (syntaxe et sémantique définies formellement par des outils mathématiques).
- *Caractère opérationnel ou déclaratif* : les spécifications opérationnelles décrivent *le comportement désiré* par un modèle ce qui permet d'une certaine manière de le simuler; par opposition, les spécifications déclaratives décrivent seulement *les propriétés désirées*.

3.3.2 Techniques de spécification

Les techniques de spécification utilisées dans les méthodes d'analyse et de conception peuvent être :

- *Les spécifications en langage naturel,*
- *Les spécifications techniques avec des langages spécialisés*

- *Les machines d'états finis*
- *Les réseaux de Petri*
- *Les schémas entité-association*
- *Les spécifications formelles et logiques temporelles*
- *…*

Souvent, les techniques de spécification se complètent pour décrire différentes vues d'un même système. Les méthodes tentent de proposer des assemblages efficaces de telles techniques avec des guides pour les construire et les valider. Le langage de modélisation UML constitue une bonne combinaison des différentes techniques de spécifications objets couvrant la totalité des aspects fonctionnel, statique, dynamique et architectural d'une application logicielle.

3.4 Langage de modélisation unifié (UML)

UML est un langage de modélisation objet qui permet de spécifier, construire, visualiser et décrire les artéfacts d'un système logiciel. Le langage représente l'état de l'art des langages de modélisations objets et possède une notation graphique riche et expressive.

UML permet la modélisation de la structure et du comportement du système indépendamment de toute méthode ou de tout langage de programmation.

- *Spécifier & Documenter* : modélisation précise, non ambiguë et complète
    - Syntaxe et sémantique bien définies des éléments de modélisation UML.
    - Support complet pour les étapes *analyse, architecture/conception, implémentation, et test*.
- *Construire* : mapping UML- OOPL (langage de programmation orientés objet).
- *Visualiser* : notation graphique riche et expressive.

Le langage de modélisation unifié ou UML est une *notation* graphique standard semi formelle qui regroupe les meilleures pratiques de l'objet et est adopté par l'OMG (*Object Management Group*).

3.4.1 Historique

Au début des années 90, il a été constaté que les méthodes objet (environ une cinquantaine) étaient liées uniquement par un accord sur les concepts de base de l'objet (attribut, objet, classe, héritage, …). Cependant, chacune des ces méthodes possédait sa propre notation et aucune méthode ne pouvait prétendre couvrir tous les besoins ni modéliser correctement les différentes vues de l'application.
En 1995, des efforts d'unification des méthodes objet, pratiques industrielles et notations ont conduit à la proposition de la méthode unifiée (*Unified Method*). Cette tentative a échoué pour les deux principales raisons :

- Dissemblance des styles de conception des développeurs,
- Diversité des classes de systèmes à développer (ordinaire/critique, …).

En fait, les méthodes objet se partagent les concepts objets et non pas les démarches. Les efforts ont été orientés depuis vers l'unification des notations manipulées par les méthodes ; UML a ainsi vu le jour.

L'historique en quelques points.

- Apparition des langages OO (mi 60 et fin 80)
- Entre 89 et 94, le nombre de méthodes OO a augmenté de 10 à 50.
- En 1995 tentative d'unification des méthodes objet, pratiques industrielles et notations :
  - *Unified Method* v0.8 draft 95,
  - Efforts réorientés vers l'unification des notations.
- UML (*Unified Modeling Language*) proposé en 1996,
- Normalisation OMG.
  - UML v0.9 en Juin 96,
  - UML 1.1 adopté par l'OMG en novembre 97,
  - UML 1.3 en 99,
  - UML 1.4 (UML 1.4.2 standard ISO/IEC 19501),
  - UML 2.0 2003,
  - UML 2.1.2 novembre 2007,
  - Version actuelle : UML 2.4.1 ; les travaux d'amélioration continuent toujours http://www.uml.org/
- Principaux auteurs :
  - Ivar Jacobson (OOSE),
  - Grady Booch (BOOCH'93),
  - James Rumbaugh (OMT), et
  - Autres partenaires…
- Rôle de l'OMG (*Object Management Group)* http://www.omg.org/
  - Standardisation des technologies Objet (UML, CORBA, etc.),
  - Révisions basées sur les contributions de la communauté des concepteurs UML.

3.4.2 Organisation du langage.

UML n'impose pas un processus de développement particulier, mais il est préférable (selon ses auteurs) de prévoir un processus de développement centré sur l'architecture, guidé pas les cas d'utilisation, itératif et incrémental (principales caractéristiques du Processus Unifié). La figure 3.1 présente l'organisation en vues des neuf principaux diagrammes UML (UML 1.4).

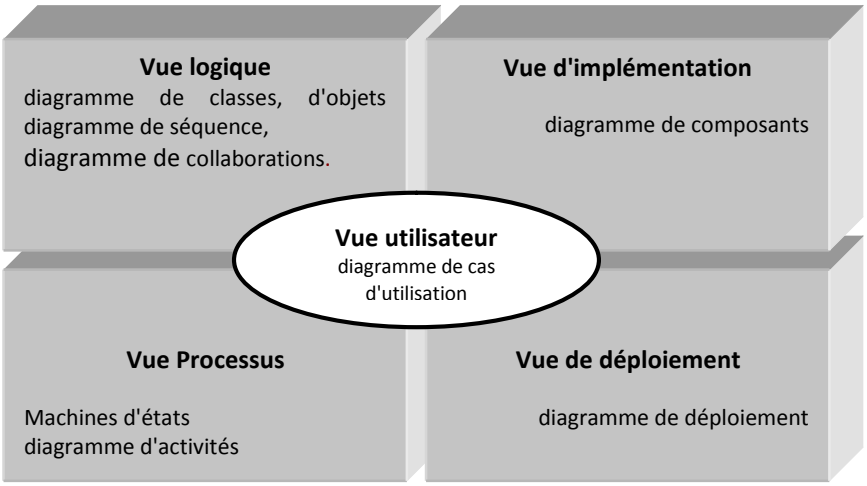


Fig. 3.1 organisation d'UML.

3.5 Diagrammes UML

A partir de la version 2.0, UML a défini quatre nouveaux diagrammes et une nouvelle structuration de sa collection de treize diagrammes : structure, comportement et interaction.

Structural Diagrams	Behavioral Diagrams	Interaction Diagrams
Class	Use case	Sequence
Object	State Machine	Communication
Package	Activity	Interaction Overview
Composite Structure		Timing
Component		
Deployment		

3.5.1 Diagrammes de Structure

Cette partie définit les constructions statiques et structurelles (classes, composants, nœuds, etc.) et leurs utilisations dans les différents diagrammes structurels, tels que le diagramme de classes, le diagramme de composants, et le diagramme de déploiement.

3.5.1.1 Diagramme de Classes

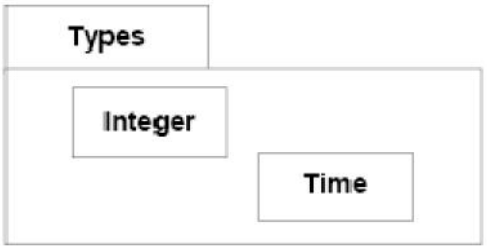
Un diagramme de classe exprime de manière générale la structure statique du système en termes de classes et de relations entre classes. Un diagramme de classes regroupe généralement les éléments de modélisation suivants : *Association, Aggregation, Class, Composition, Dependency, Generalization, Interface, InterfaceRealization, Realization*.

3.5.1.2 Diagramme d'Objets

C'est une instance d'un diagramme de classe (comprenant des objets et des valeurs réelles) qui donne une image instantanée de l'état détaillé du système. Un diagramme d'objets contient principalement des objets (instances de classes) et des liens (instances d'associations).

3.5.1.3 Diagrammes de Packages

Un package est utilisé pour grouper des éléments (et même d'autres packages) et fournir ainsi un *namespace* pour ce groupe d'éléments.



Un package peut importer soit une partie ou la totalité des membres des autres packages. En plus, une relation *'merge'* peut être également définie entre packages.

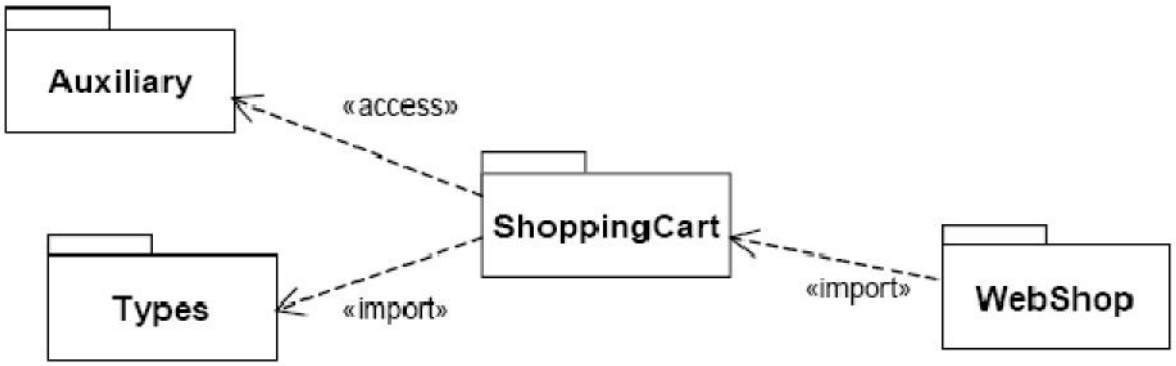


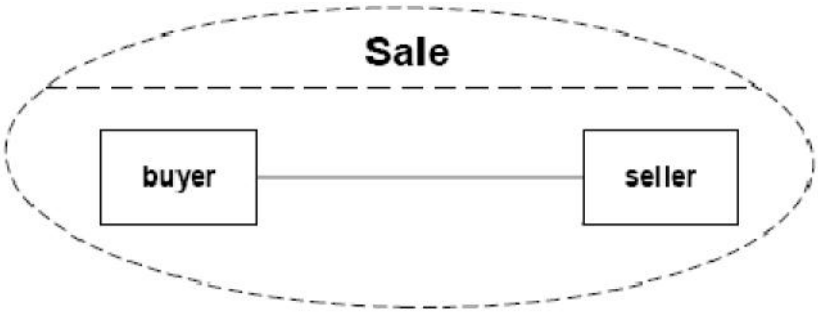
Fig. 3.2 diagramme de packages.

Les éléments de modélisation typiquement utilisés dans un diagramme de package sont : *Dependency, Package, PackageExtension*, et *PackageImport*.

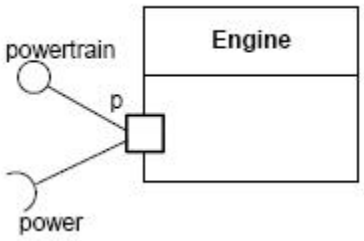
3.5.1.4 Diagramme de Structure Composite

Un diagramme de structure composite montre la structure interne d'un classificateur, ainsi que l'utilisation d'une collaboration dans un *'collaboration use'*. Le terme *"structure"* désigne ici une composition d'éléments interconnectés, représentant des instances *run-time* en collaboration à travers des liens de communications pour accomplir certains objectifs communs.

Structures internes. Des structures d'éléments interconnectés qui sont créés dans une instance du classificateur container. Une structure de ce type représente une décomposition de ce classificateur.



Ports. Dont l'objectif est d'isoler un classificateur de son environnement en offrant un point permettant d'accomplir les interactions entre ses éléments internes du classificateur et son environnement. Ce découplage entre les éléments internes du classificateur et son environnement permet une définition indépendante et même réutilisable du classificateur.



Collaborations. Les *Collaborations* permettent une description exclusive des aspects pertinents d'une coopération d'un ensemble d'instances par l'identification des rôles spécifiques joués par chacune de ces instances.

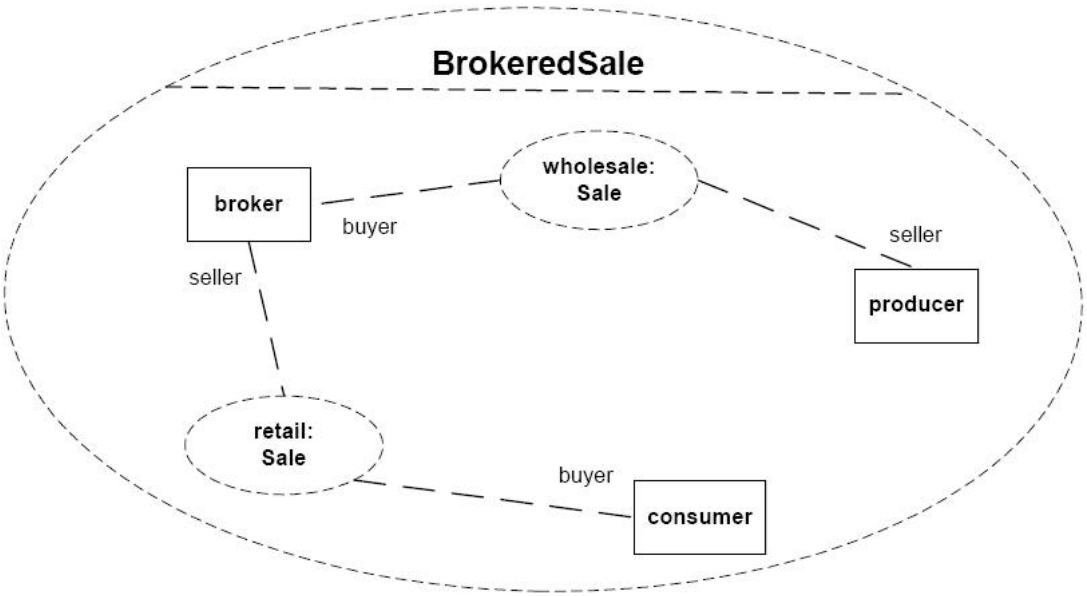


Fig. 3.3 collaboration.

Classes structurées. C'est un moyen permettant la représentation des classes qui peuvent avoir des ports en plus d'une structure interne.

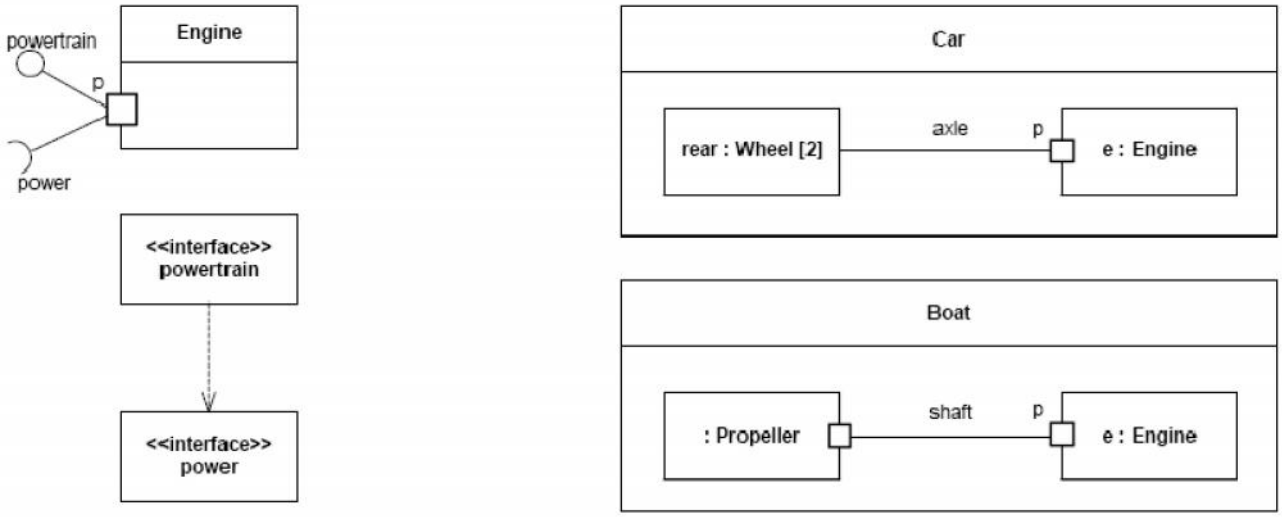


Fig. 3.4 classes structurées.

3.5.1.5 Diagramme de Composants

Les diagrammes de composants montrent la structure des composants, les classificateurs qui les spécifient (classes d'implémentation par exemple) et les artéfacts qui les implémentent (code source, binaire, exécutables, etc.). Ces diagrammes contiennent généralement des composants (descriptions d'implémentation) qui peuvent être composites, et des dépendances (utilisations de services des composants).

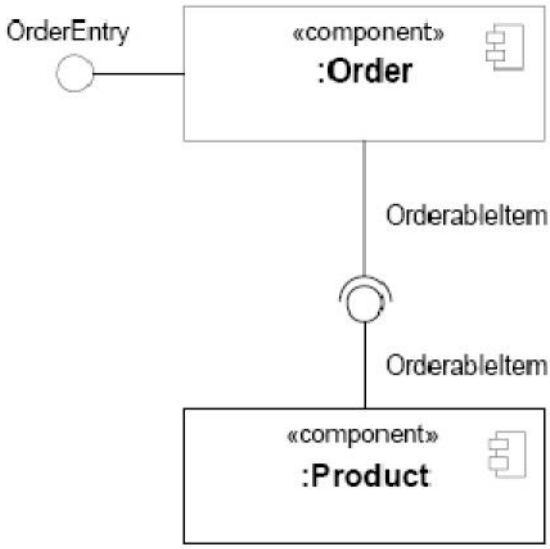


Fig. 3.5 diagramme de composants.

Les composants représentent en général des implémentations de classes, implantent des services et/ou requièrent des services offerts par d'autres composants. Le type du composant peut être spécifié par un stéréotype (document, exécutable, table, etc.).

3.5.1.6 Diagrammes de Déploiement

Les diagrammes de déploiement reflètent la structure des nœuds sur lesquels les composants sont déployés et les moyens de communication entre ces nœuds. Ils existent sous deux formes : spécification et instance. En général, un nœud représente une ressource matérielle qui possède ses propres attributs (capacité mémoire, vitesse d'horloge, …). La nature de ces ressources peut être précisée par un stéréotype (environnement d'exécution, dispositif, etc.).

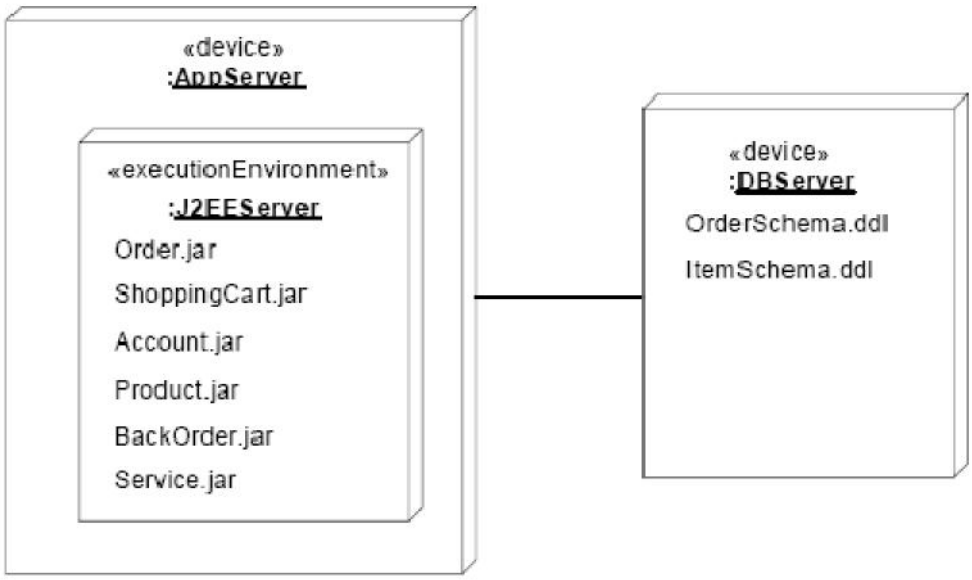


Fig. 3.6 diagramme de déploiement.

Dans l'exemple que montre la figure 2.8, les composants déployés sont exprimés textuellement (*Order.jar*, *OrderSchema.dll*, …).

3.5.2 Diagrammes du Comportement

3.5.2.1 Diagramme de Cas d'utilisation

Les cas d'utilisation décrivent les fonctionnalités d'un système ou d'un classificateur (sous-système ou classe) du point de vue de l'utilisateur (ou des éléments externes) en interaction avec le système ou le classificateur.

Un *cas d'utilisation* décrit le comportement du système du point de vue d'un utilisateur en précisant les limites du système et ses relations avec l'environnement.

Un *acteur* représente un *rôle* joué par une personne ou un système externe qui interagit avec le système étudié ; la même personne physique (ou système externe) peut jouer différents rôles (donc différents acteurs).

Un diagramme de cas d'utilisation traduit la relation entre les cas d'utilisation dans le système et leurs acteurs. Des descriptions détaillées des différents scénarios possibles du même cas peuvent être réalisées à l'aide des diagrammes d'interaction.

Relations entre cas d'utilisation :

Généralisation. Le cas d'utilisation fils est une spécialisation du cas père.

Inclusion. Une instance du cas source comprend le comportement décrit par le cas cible.

Extension. Le cas source ajoute son comportement au cas destination; l'extension est soumise à la vérification d'une condition (point d'extension).

Dans l'exemple :

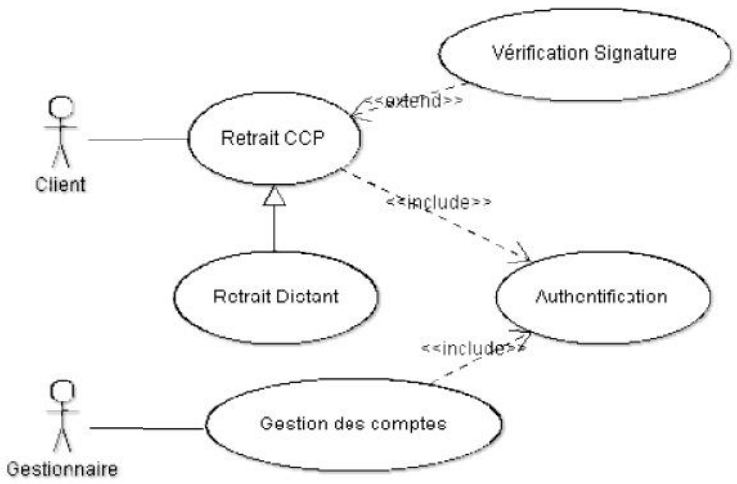


Fig 3.7 un diagramme de cas d'utilisation.

- Le comportement "Retrait Distant" est une spécialisation du comportement "RetraitCCP".
- Le comportement "RetraitCCP" inclut toujours celui de "Authentification".
- Le comportement "Vérification Signature" étend le comportement "RetraitCCP" ; le point d'extension étant "MontantRetrait ≥ 20.000DA".

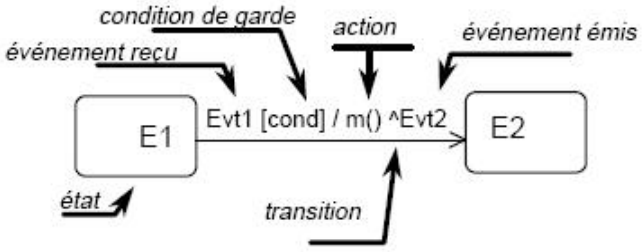
3.5.2.2 Machines d'états (*Statecharts*)

Les machines d'états (dérivées des statecharts 1987) décrivent le comportement des instances d'un élément modèle (objet ou interaction par exemple). Elles représentent les séquences possibles des états et des actions par lesquels les instances de l'élément peuvent passer en réaction aux événements reçus (signal, invocations d'opération, etc.). Une machine d'état est attachée d'habitude à chaque classe active dans le diagramme de classes. Le comportement global du système est défini par l'ensemble des machines d'états des objets actifs constituants ce système.

Evénement. Un stimulus pouvant générer des réactions dans le système. Il peut avoir différentes formes : *appel, signal, changement* ou *temporel*.

Etat. C'est une étape dans l'évolution du système pendant laquelle, il exécute une action ou attend un événement. Le système doit également satisfaire une condition appelée invariant tant qu'il se trouve dans cet état. Les états peuvent être *simples, composites séquentiels* ou *composites concurrents*.

Transition. Passage éventuel d'un état à un autre (qui peut être le même). Les transitions sont instantanées et le temps ne peut s'écouler que si le système est dans un état propre. Chaque transition est caractérisée par un état source, un état destination, un événement déclencheur (trigger), une garde (condition) et une liste d'actions à exécuter lors du franchissement de la transition :



Sémantique opérationnelle. Après occurrence de l'événement déclencheur, si la garde de la transition est évaluée à *vrai*, la transition est dite alors *franchissable*. Elle est exécutée (tirée ou traversée) s'il existe une configuration atteignable (sans violation d'invariant) et si la transition ne présente aucun conflit avec le reste des transitions franchissables. Deux transitions exécutables présentent un conflit si elles appartiennent à une même hiérarchie d'états. UML donne priorité à la transition du plus bas niveau et les autres transitions conflictuelles seront alors abandonnées.

Le principe de base d'exécution d'une machine d'états est qu'elle traite un seul événement à la fois et finit de traiter toutes ses conséquences avant d'en traiter un autre (*run-to-completion*). Les

actions sont instantanées et les événements ne sont jamais simultanés. Si un nouvel événement (*deferrable*) est reconnu pendant l'exécution d'une étape *run-to-completion*, l'occurrence de l'événement est placée dans un pool d'événement (aucune structure de données n'est imposée). Après achèvement d'une étape RTC, le système traite tous les événements dans le pool un par un (aucune politique de sélection n'est imposée) de la même manière et exécute ses transitions et passe à une autre configuration stable.

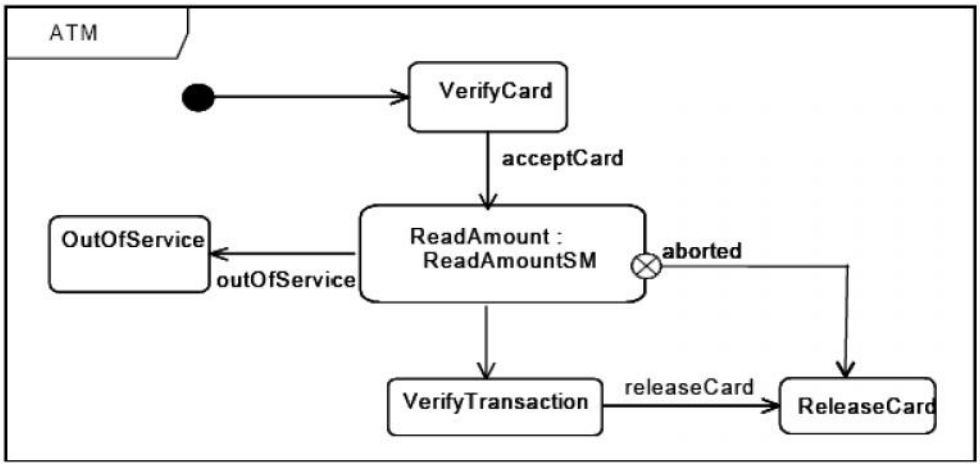


Fig. 3.8 un exemple d'une machine d'états.

3.5.2.3 Diagrammes d'activité

C'est une variante des machines d'états dans laquelle les états représentent l'exécution des actions (ou sous-activités) et les transitions sont déclenchées par accomplissement des actions (ou sous-activités). Ils sont proposés pour la représentation du comportement des opérations d'une classe ou la formalisation d'un processus d'une organisation.

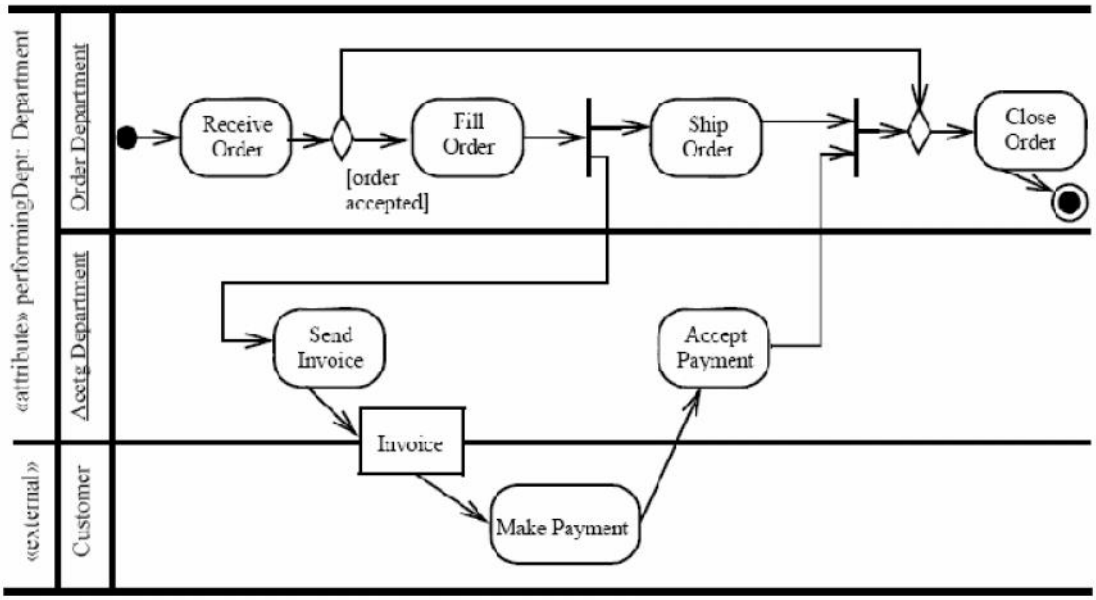


Fig. 3.9 diagramme d'activités.

3.5.3 Diagrammes d'interaction

Les diagrammes d'interaction possèdent différentes variantes :

- diagramme de Séquence qui focalise sur l'échange de messages entre un nombre de lignes de vie (*lifelines*),
- diagramme de Communication montrant les interactions d'un point de vue architectural,
- diagramme *'Interaction Overview'* qui est une variante du diagramme d'activités définissant les interactions de façon à encourager la représentation des flots de contrôle,
- diagramme *'Timing'* est utilisé pour la représentation des interactions lorsque l'objectif principal est de raisonner sur le temps.

3.5.3.1 Diagramme de séquence

Les diagrammes de séquence montrent des interactions entre objets selon un point de vue temporel. Dans ce contexte, il est utile de noter que UML propose un large éventail de mécanismes de communication inter objets (appels d'opérations, signaux, invitations, exceptions, envois synchrone et asynchrone, etc.).

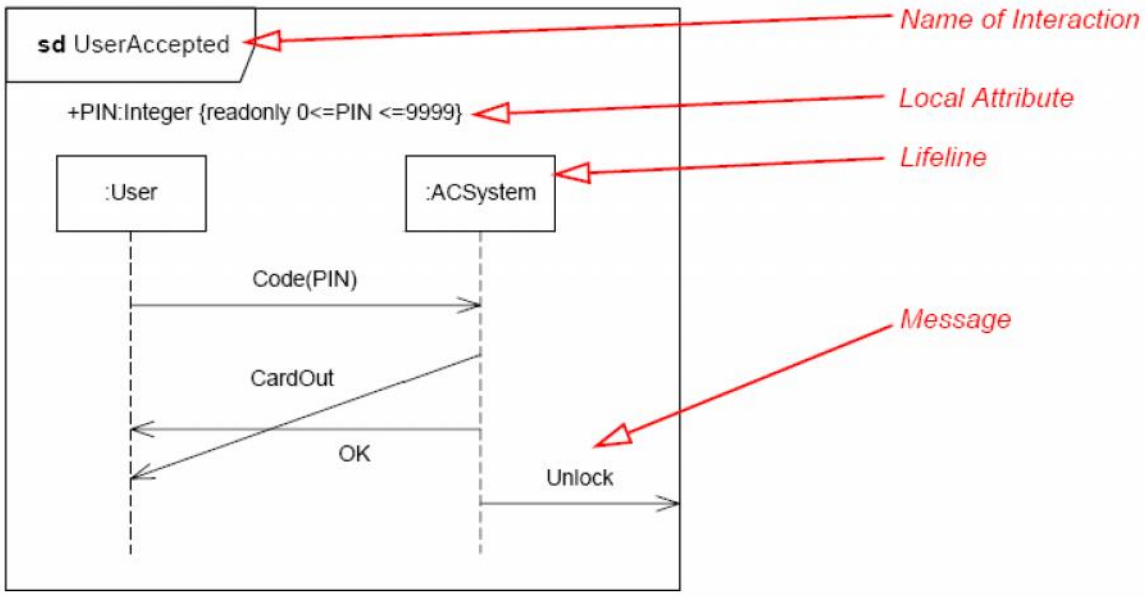


Fig. 3.10 notation du diagramme de séquence.

3.5.3.2 Diagrammes de communication (collaboration)

Ces diagrammes insistent sur les interactions entre lignes de vie du point de vue architecture de la structure interne les échanges de messages correspondants. L'ordonnancement des messages est réalisé à l'aide d'un schéma de numérotation de séquence.

Un diagramme de collaboration, montre l'interaction organisée autour des rôles et leurs relations. Le temps n'est pas montré comme dimension séparée et les séquences de communication et threads concurrentes doivent être déterminées par l'utilisation de nombres de séquence. Une collaboration est utilisée pour décrire la réalisation d'une opération ou d'un classificateur ce qui

simplifie l'identification des design patterns présents (un pattern est une collaboration paramétrique dont chaque utilisation, les classificateurs réels vont remplacer les paramètres de définition du pattern).

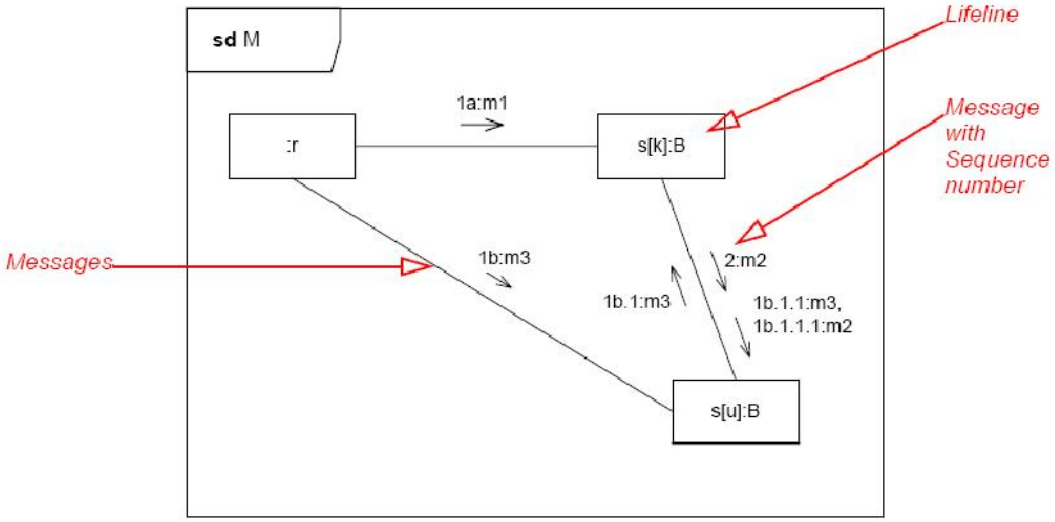


Fig. 3.11 notation du diagramme de communication.

3.5.3.3 *'Interaction Overview* Diagrams*'*

Ces diagrammes définissent les interactions par le biais d'une variante des diagrammes d'activités de manière à décrire le flot de contrôle. Les lignes de vie et les messages n'apparaissent pas au niveau *overview*.

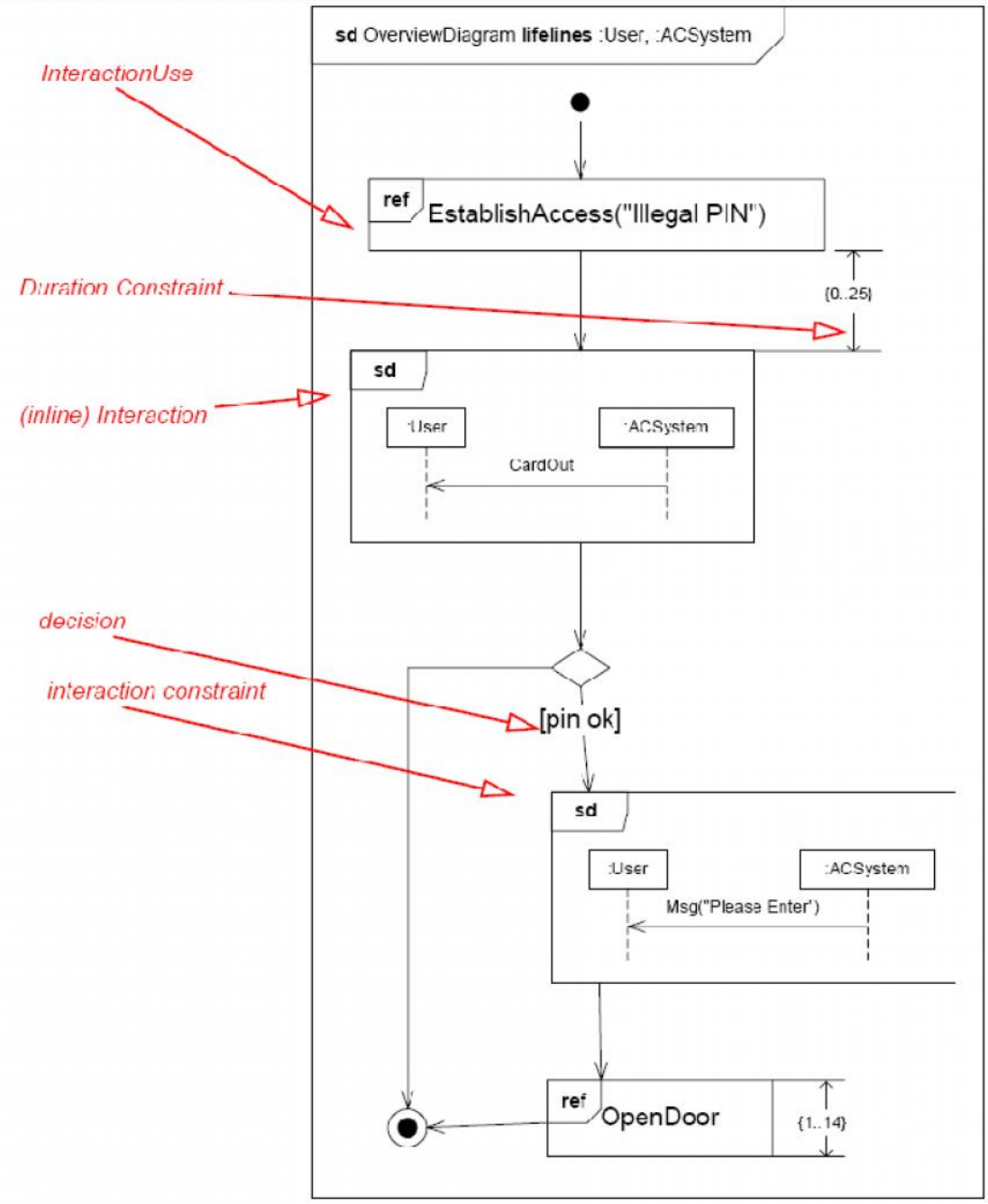


Fig. 3.12 notation *'Interaction Overview'*

3.5.3.4 *'Timing Diagrams'*

L'objectif principal de ces diagrammes est la représentation des contraintes de temps sur les interactions. Ils décrivent le comportement avec les temps d'occurrence des événements qui provoquent des changements dans les conditions ou états modélisés des lignes de vie.

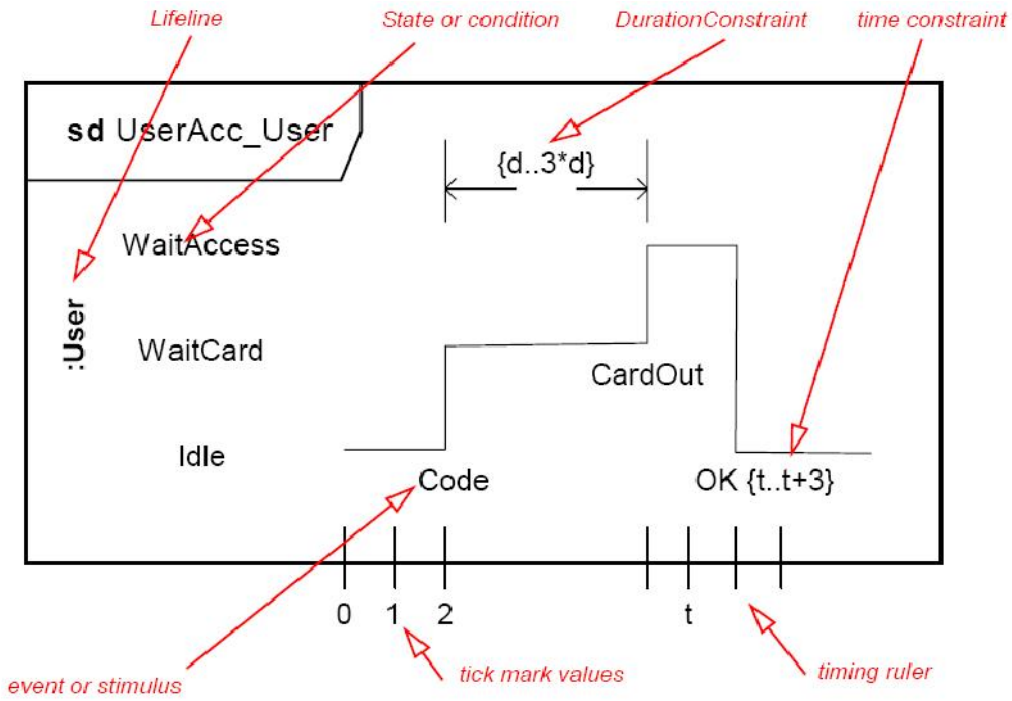


Fig. 3.13 notation 1 *'Timing Diagram'*

Ces diagrammes peuvent être également utilisés pour montrer les changements d'états d'un objet en réponse à des événements ou stimuli d'une perspective temporelle.

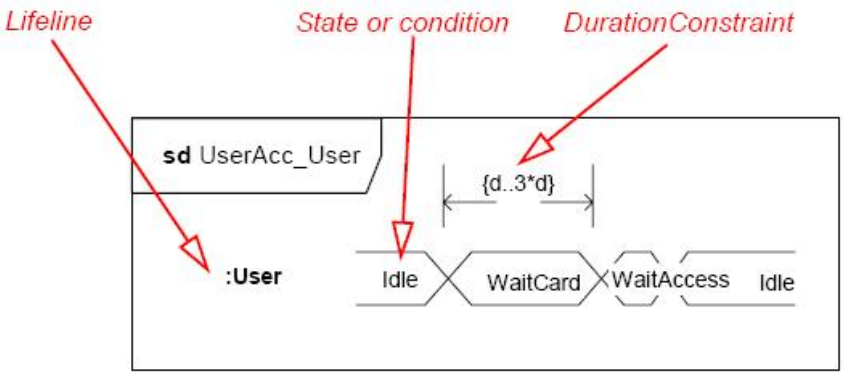


Fig. 3.14 Notation 2 *'Timing Diagram'*

Finalement, il est possible d'avoir une forme plus élaborée de ces diagrammes dans laquelle plusieurs lignes de vie et messages seront représentés.

3.5.4 Extensibilité UML & notion de Profils

UML fournit un ensemble riche et soigneusement choisi de notations et concepts de modélisation. Cependant, certains projets peuvent parfois exiger des mécanismes de représentation additionnels autres que ceux prédéfinis. Pour répondre à ce besoin, UML permet par l'intermédiaire de ses mécanismes d'extension l'ajout de nouveaux éléments de modélisation ou de formes libres d'information. C'est un moyen pour raffiner la sémantique standard d'UML et permettre ainsi l'ajout de nouveaux éléments de modélisation qui seront utilisés dans la création de modèles UML spécifiques. Une restriction fondamentale sur toutes les extensions définies est qu'elles doivent être strictement additives à la sémantique standard.

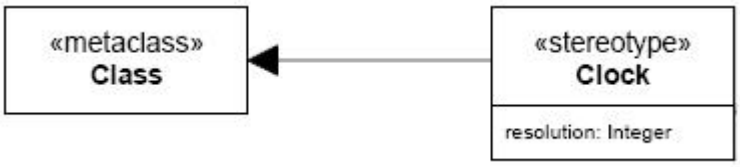
Les éléments d'un modèle UML sont personnalisés et étendus avec de nouvelles sémantiques en utilisant *des stéréotypes*, *des contraintes* et *des valeurs marquées*. Un ensemble cohérent de telles extensions définies pour un objectif spécifique constitue un *profil UML*.

Contraintes. Des expressions écrites dans un langage de définition de contraintes donné pour permettre la spécification linguistique de nouvelles sémantiques d'un élément modèle (des restrictions sémantiques que l'élément doit obéir). Le langage peut être spécifique (*OCL* par exemple), un langage de programmation, des notations mathématiques ou le langage naturel.

Stéréotypes. Fournissent un moyen de classification d'éléments de modélisation (classes, associations, etc.) pour qu'ils se comportent sous certaines considérations comme étant des instances de nouvelles constructions virtuelles du méta modèle. C'est un élément modèle qui définit des valeurs (basées sur valeurs marquées) et contraintes additionnelles et facultativement une nouvelle représentation graphique. Tout élément marqué par un stéréotype particulier reçoit alors ces valeurs et contraintes en plus des attributs, associations et super classes que l'élément possède.

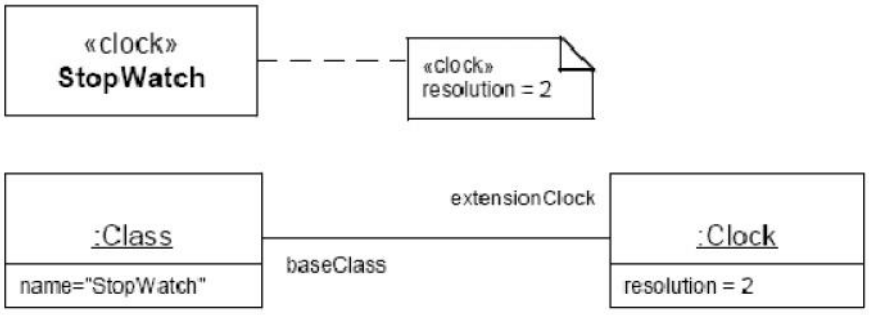
Valeurs marquées (*Tagged value*). Permettent d'attacher l'information à n'importe quel élément modèle en conformité avec la définition de marque. Cette dernière (*Tag definition*) spécifie les valeurs marquées qui peuvent être attachées à un genre d'éléments modèles.

Définition de stéréotype :

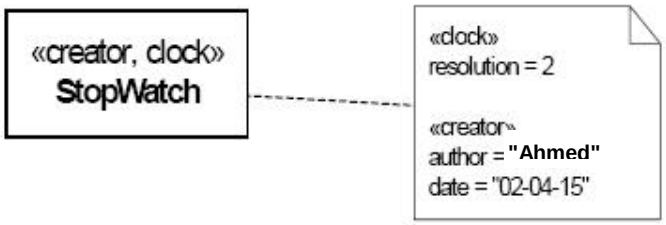


Utilisation du stéréotype :

Utilisation du stéréotype avec *présentation de valeurs* :



Application de *plusieurs stéréotypes* sur le même élément de modélisation :



Profils. Les profils UML présentent un mécanisme permettant de spécialiser le langage pour un contexte particulier (analyse, conception technique, codage, etc.) par l'introduction de notions plus adaptées au contexte de travail actuel, de règles de modélisation spécifiques, et de modes de présentation des modèles adaptés.

Un profil est un package stéréotypé qui contient des éléments de modélisation personnalisés pour un objectif ou un domaine spécifique en étendant le méta modèle par des stéréotypes, des définitions de marques, et des contraintes.

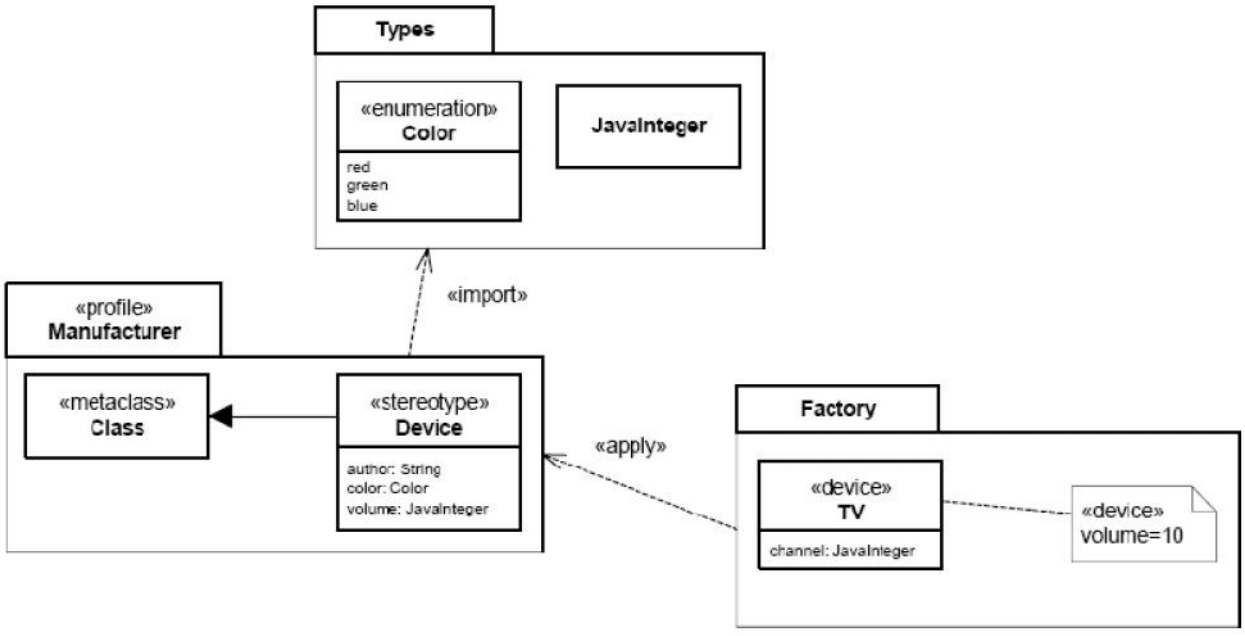


Fig. 3.15 définition et utilisation de profil.

Pour traiter la complexité de certains domaines particuliers, La politique de l'*OMG* repose sur le lancement de plusieurs *RFPs* (*Requests For Proposal*) spécifiques et d'évaluer par la suite les propositions. De cette manière, divers profils ont été définis ou améliorés par l'OMG tels que le *SPT et MARTE (*profils du temps réel*)*, *EDOC*, *CORBA*, *EJB*, etc.

3.5.5 Langage de définition de contraintes (OCL)

Les diagrammes UML ne sont pas typiquement assez raffinés pour fournir tous les aspects pertinents d'une spécification. Il y a également le besoin de décrire des contraintes additionnelles sur les objets du modèle. D'un autre coté, les contraintes exprimées en langage naturel conduisent souvent à des ambiguïtés.

OCL (*Object Constraint Language*) est un langage formel simple développé pour répondre à ce besoin. C'est un pur langage de spécification ; il ne peut pas donc changer quoi que ce soit dans le modèle ou son exécution. La version actuelle (janvier 2009) adoptée par l'OMG est OCL 2.0.

OCL peut être utilisé dans différentes situations :

- spécification d'invariants sur les classes et les types dans un modèle de classes,
- spécification des invariants de type pour les stéréotypes,
- description des pré et post-conditions sur les opérations et les méthodes,
- description des gardes,
- spécification des destinations des messages et actions,
- spécification de contraintes sur les opérations,
- spécification des règles de dérivation d'attributs pour toute expression sur un modèle UML.

Un exemple de contrainte OCL attachée à l'attribut *numberOfEmployees* d'une classe *Company* :

```
context Company inv:
self.numberOfEmployees > 50
```

Des contraintes de pré et post-condition d'une opération peuvent être exprimées de la manière suivante :

```
context Typename::operationName(param1 : Type1, ... ): ReturnType
pre parameterOk: param1 > ...
post resultOk : result = ...
```

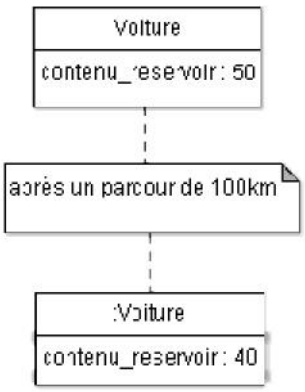
4 Concepts objet

4.1 L'objet

Un objet définit une représentation d'une entité atomique réelle ou virtuelle, dans le but de le piloter ou de le simuler. Il encapsule une partie des connaissances du monde dans lequel il évolue. Un objet associe données et traitements en ne laissant visible que l'interface de l'objet (les traitements que l'on peut faire dessus).

Objet = Identité + Etat + Comportement

- *L'identité* : permet de distinguer l'objet de manière non ambiguë indépendamment de son Etat (non explicitée);
- *L'état* : défini par les valeurs instantanées de tous les attributs d'un objet. Il évolue au cours du temps;



- *Le comportement* : regroupe toutes les compétences (services) d'un objet et décrit les actions et les réactions de cet objet. Chaque comportement élémentaire d'un objet est appelé *opération* et est déclenché suite à une stimulation externe (message envoyé par un autre objet).

Les objets d'un système informatique travaillent en collaboration pour réaliser les fonctions de l'application. Le comportement global de l'application repose sur la communication entre les objets. Cette communication est réalisée par envoi et réception de messages.

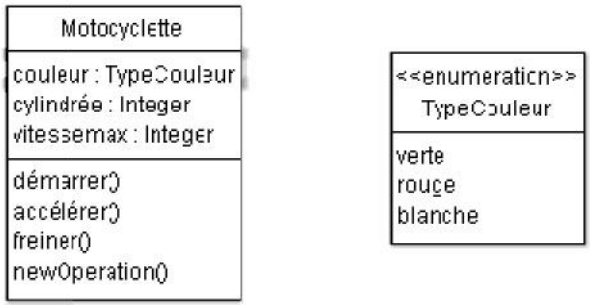
4.2 Démarche d'abstraction

L'abstraction consiste à concentrer la réflexion sur un élément d'une représentation en négligeant tous les autres.

La démarche d'abstraction procède de l'identification des caractéristiques communes à un ensemble d'éléments, puis de la description condensée de ces caractéristiques dans ce qui est appelé *classe*. La démarche se définit par rapport à un point de vue (critères pertinents dans le domaine considéré).

4.3 Classe

Une classe décrit le domaine de définition d'un ensemble d'objets; les objets d'une classe sont construits par instanciation.



4.4 Attributs et Opérations

- les attributs correspondent aux propriétés de la classe. En phase d'analyse, il est recommandé de ne pas confondre entre objet et attribut :

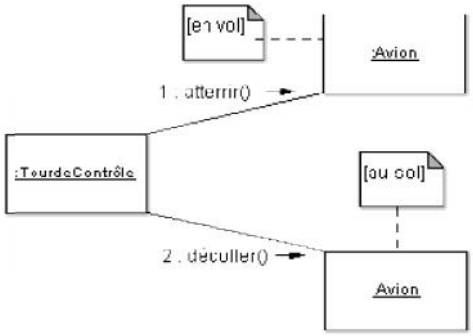
"Si l'on ne peut demander à un élément que sa valeur, il s'agit d'un simple attribut; si plusieurs questions s'y appliquent, il s'agit plutôt d'un objet qui possède lui même plusieurs attributs ainsi que des liens avec d'autres objets"

- les opérations décrivent la spécification du comportement des objets de la classe (une méthode est une implémentation d'une opération ou service). Elles sont identifiées après étude des différents scénarios décrivant les fonctionnalités du système. Cinq types d'opérations sont distingués :

- Constructeurs,
- Destructeurs,
- Sélecteur (opération de consultation qui renvoie l'état de l'objet),
- Modificateurs, et
- Itérateurs (qui visitent l'état d'un objet ou une structure de donnée qui contient plusieurs objets.

4.5 Communication entre objets

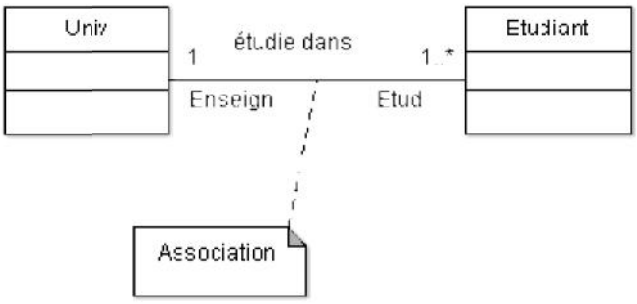
La communication entre objets est réalisée par envoi et réception de messages. Un message regroupe les flots de contrôle et les flots de données et peut prendre différentes formes : appel de méthode, événement discret, interruption, etc.



4.5 Relations entre classes

Les relations entre classes peuvent prendre différentes formes :

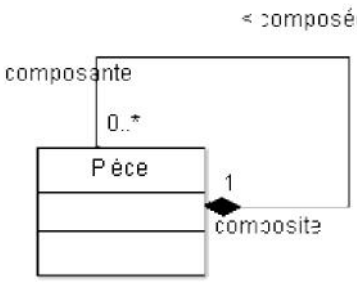
1. Association : abstraction des liens qui existent entre objets (connexion sémantique bidirectionnelle).



Décoration : étudie dans
Rôles : Etud, Enseign
Multiplicités : 1, 0..1, M..N, * ou 0..*, 1..*

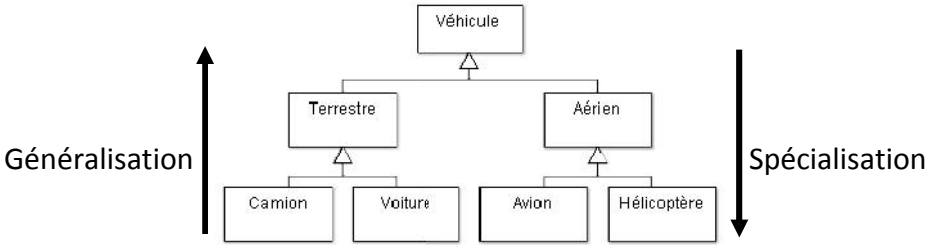
2. Agrégation : permet d'exprimer des relations de type maître / esclave (ensemble-élément, tout-partie, composé-composant, etc.).

3. Composition : forme d'agrégation avec couplage plus important. Les composants ne sont pas partageables et la destruction de l'agrégat entraîne la destruction des composants agrégés.



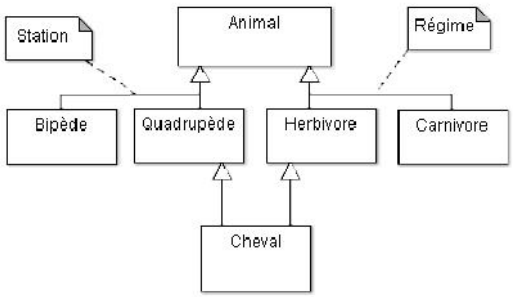
4. Hiérarchie de classes : c'est une classification des objets au sein d'une arborescence de classes permettant ainsi de gérer la complexité par réutilisation des caractéristiques héritées. C'est une relation non réflexive et non symétrique.

L'héritage peut prendre l'une des deux formes (selon le sens de la lecture ou le besoin d'analyse) ; généralisation ou spécialisation.



- La généralisation est employée une fois que les éléments du domaine ont été identifiés. Elle consiste alors à factoriser les informations communes entre classes.
- La spécialisation permet de capturer les particularités (d'un ensemble d'objets) non couvertes par les classes déjà identifiées. Les nouvelles caractéristiques sont représentées par une nouvelle classe, sous classe d'une des classes déjà existantes.
- La définition des sous classes doit répondre à un critère de classification pertinent et non pas sur des valeurs particulières des attributs d'une même classe.
- La généralisation introduit un couplage statique très fort et non mutable. Donc, elle n'est pas adaptée pour représenter les métamorphoses.

5. Héritage multiple :

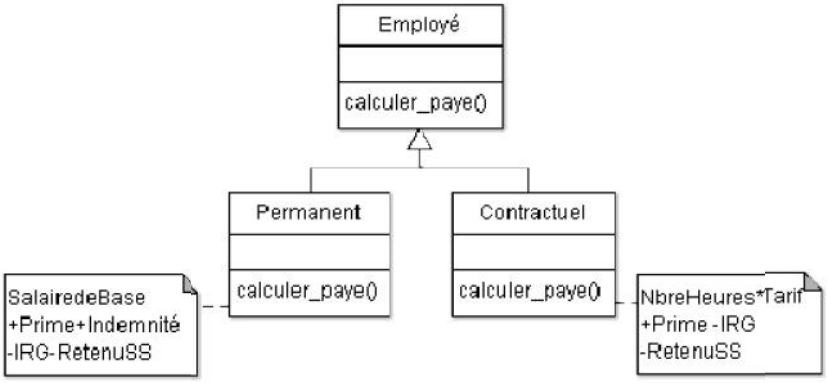


4.6 Délégation

La délégation est basée sur l'agrégation et consiste à une propagation manuelle des propriétés. Une représentation équivalente de l'héritage multiple de la figure précédente peut être réalisée par délégation.

4.7 Polymorphisme

Le polymorphisme implique qu'un nom d'objet peut désigner des instances de classes différentes issues d'une même arborescence; le polymorphisme d'opérations offre la possibilité de déclencher des opérations différentes en réponse à un même message (désignation donnée au niveau de la super classe).



5.1 Introduction

Malgré le développement des nouvelles technologies de l'information, des défaillances et des erreurs sérieuses découlent toujours des différentes phases du développement de logiciels. Ces erreurs sont dues principalement aux :

- Les personnes qui rédigent les cahiers des charges sont très rarement les personnes qui développent les systèmes logiciels,
- Le cahier des charges est très souvent écrit en langage naturel; source d'ambiguïté, d'imprécision ou d'incohérence,
- La complexité des applications que les machines actuelles permettent de réaliser.

Donc, il est crucial de permettre la *détection* (vérification) et la *correction* de ces erreurs (maintenance curative) bien avant la livraison ou l'exploitation du produit logiciel. Le rôle de la vérification se limite alors à la détection des erreurs susceptibles d'être présentes dans le système. Elle ne propose, par contre, aucune solution vis-à-vis aux problèmes rencontrés.

La vérification vise principalement à s'assurer que le comportement du produit logiciel (spécification ou exécutable) satisfait toutes les propriétés exigées dans le système.

Différentes techniques ont été proposées afin d'accomplir cette tâche, dont les plus importantes sont : le test, la démonstration et le model-checking.

5.2 Test

Le test est le processus manuel ou automatique qui vise à vérifier qu'un système respecte les propriétés exigées par sa spécification ou à détecter des différences entres les résultats attendus et ceux générés par le système.

Types de tests. Les tests peuvent être statiques ou dynamiques. Les méthodes de *test statique* consistent en l'analyse textuelle du code du logiciel (revue de code ou recherche d'anomalies) afin de détecter des erreurs sans avoir à exécuter le programme. Les techniques de spécification utilisées dans ce type de test peuvent varier des graphes de contrôle, les diagrammes de flots de données, les systèmes de transitions, etc.

Pour les techniques de *test dynamique*, l'exécution du programme est nécessaire pour déceler les différentes fautes. La démarche repose sur la sélection et l'exécution d'un jeu de tests et la comparaison des résultats *obtenus* avec ceux *prévus* ce qui permet de décider le succès ou l'échec du test.

5.2.1 Construction des jeux de tests

Trois différentes approches sont distinguées pour : structurelle, fonctionnelle et aléatoire.

- *Approche structurelle* (boite blanche), consiste à une sélection de jeu de tests sur la base de la structure du code source de l'implémentation. Cette sélection est souvent fondée sur

la notion de critère de couverture qui définit la façon dont les constructions de base du code sont testées.

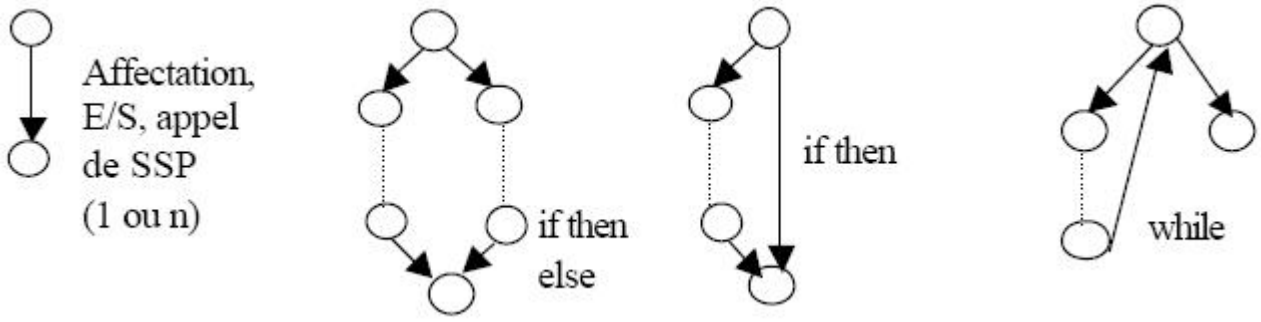
- *Approche fonctionnelle* (boite noire), repose sur la sélection du jeu de test à partir de la spécification des fonctionnalités du logiciel. La sélection dans ce cas dépend du degré de formalité de la spécification.
- *Approche aléatoire*, caractérisée par une sélection au hasard des jeux de test sur le domaine des entrées du programme. Le domaine de définition des entrées du programme est déterminé à l'aide des *interfaces* de la spécification ou du programme. Cette méthode ne garantit pas une bonne couverture de l'ensemble des entrées du programme. En particulier, elle peut ne pas prendre en compte certains cas limites ou exceptionnels. Cette méthode a donc une *efficacité très variable*.

5.2.1.1 Approche structurelle (boite blanche)

Ce test consiste à analyser la structure interne du programme en déterminant les chemins minimaux. Afin d'assurer que:

- Toutes les conditions d'arrêt de boucle ont été vérifiées.
- Toutes les branches d'une instruction conditionnelle ont été testées.
- Les structures de données internes ont été testées (pour assurer la validité).
- …

Structures de la représentation de la boîte blanche. La structure de contrôle se présente sous la forme d'un graphe de flots. On représente les instructions comme suit :



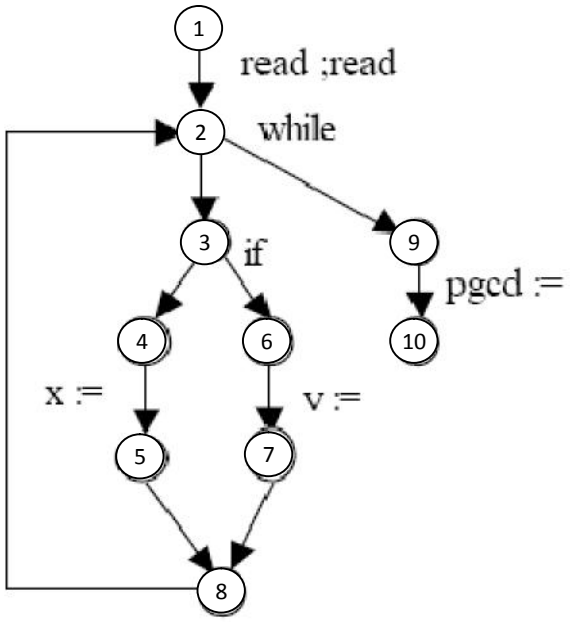
Remarque : If A & B & C ⇔ If A then if B then if C then…

Mesure de complexité Cyclomatique. Cette mesure donne le nombre de chemins minimaux. Elle est donnée par la formule suivante qui correspond au nombre de régions du graphe de flot :

Nbre Arcs – Nbre Nœuds + 2 = *Nombre de cycles*

Exemple : Supposons le programme représenté par l'organigramme suivant:

```
begin
  read(x) ; read(y) ;
  while (not (x = y) ) loop
    if x > y then
      x := x - y ;
    else
      y := y - x ;
    end if ;
  end loop ;
  pgcd := x ;
end ;
```



Donc le nombre de cycles est : Nbre Arcs - Nbre Nœuds + 2 = 11 - 10 + 2 = **3**

Pour vérifier, on regarde les chemins minimaux (un test par chemin pour tester toutes les possibilités du programme). Dans l'exemple, les trois chemins sont :

1. **1.2.9.10**
2. **1.2.3.4.5.8.2.9.10**
3. **1.2.3.6.7.8.2.9.10**

5.2.1.2 Approche fonctionnelle (boite noire)

On considère seulement la spécification de ce que doit faire le programme, sans tenir compte de sa structure interne. On peut donc vérifier chaque fonctionnalité décrite dans la spécification (On s'appuie principalement sur les données et les résultats). Le danger est l'explosion combinatoire qu'entraîne un grand nombre d'entrées du programme. Par contre, on peut écrire ces tests très tôt, dès qu'on connaît la spécification.

Principe :

1. On considère le programme dans son aspect fonctionnel et non plus structurel.
2. On partitionne le domaine (DE) en classes.
3. On génère des cas de test aux limites des classes.

Exemple : Soit P un programme. Supposons que les données de P soient des nombres de cinq chiffres. Alors les classes de nombre à cinq chiffres s'obtiennent de la manière suivante:

1. $x < 10\ 000$
2. $10000 \leq x \leq 99\ 999$
3. $x \geq 100\ 000$

Les cas de test aux limites de ces classes sont donc 00 000 et 09 999 pour la première classe, 10 000 et 99 999 pour la deuxième classe et 100 000 pour la troisième.

5.2.2 Types de tests

a) Les test unitaires de programmes ou de modules Dans ce qui précède nous avons fait l'hypothèse du test d'un programme isolé. Le test d'un module ressemble au test d'un programme isolé si ce n'est que le module ne fonctionne pas seul mais utilise d'autres modules et est appelé par d'autres modules. Pour tester un module, il faut simuler le comportement des modules appelés (relation 'utilise') et simuler les appels du module.

b) Les tests d'intégration Après avoir testé unitairement les modules il faut tester leur intégration progressive jusqu'à obtenir le système complet.

Test alpha : l'application est mise dans des conditions réelles d'utilisation, au sein de l'équipe de développement (simulation de l'utilisateur final).

c) Les test de réception Test, généralement effectué par l'acquéreur dans ses locaux après installation d'un système, avec la participation du fournisseur, pour vérifier que les dispositions contractuelles sont bien respectées.

Test bêta : distribution du produit sur un groupe de clients avant la version définitive.

d) Les tests de non régression A la suite de la modification d'un logiciel (ou d'un de ses constituants), un test de non régression a pour but de montrer que les autres parties du logiciel n'ont pas été affectées par cette modification.

5.2.3 Test de spécification.

Dans le test fonctionnel, l'extraction des séquences de test à partir des spécifications permet en particulier de découvrir des lacunes et des ambiguïtés de spécifications d'une manière indépendante de n'importe quelle exécution particulière de ces spécifications. Le but immédiat inclut un appui pour l'automatisation des transformations des spécifications fonctionnelles en une suite de tests.

Le test de spécification apporte l'information utilisée comme entrée de test et les résultats prévus à partir de la spécification du système sous test. Cette spécification représente une description précise du comportement du système, mais qui n'intègre pas les détailles d'implémentation. Les spécifications de type machines d'états sont très utilisées pour la représentation de la dynamique du système en termes d'états et de transitions.

Les spécifications formelles sont très adaptées pour ce type de test et peuvent servir dans une procédure de test totalement mécanisée (utilisation de model-checkers par exemple). Mais ces spécifications sont très complexes et nécessitent une expertise spécialisée souvent rare. Les spécifications semi formelles sont beaucoup plus simples à utiliser mais nécessitent une prise en charge particulière afin de bien préciser leur sémantique. La technique de spécification semi formelle la plus utilisée est certainement le langage de modélisation unifié UML. Les diagrammes état-transition d'UML (appelés également machines d'états ou statecharts) représentent un outil éprouvé de modélisation du comportement basé sur les machines d'états finis. Donc, leur utilisation semble inéluctable dans le contexte de génération de test à partir d'une spécification UML. Cependant, leur utilisation nécessite d'abord une étape de formalisation permettant de préciser leur sémantique.

5.3 Vérification formelle

5.3.1 Démonstration ou preuve de théorèmes

La démonstration permet à partir d'un système et d'une propriété (les deux spécifiés formellement par le même langage), de prouver que le système vérifie ou non la propriété. Ceci est réalisé en utilisant des *règles de déduction*, comme on pourrait le faire pour démontrer un théorème mathématique. Un outil permettant de faire une telle preuve pour chaque système et chaque propriété est bien entendu impossible à implémenter. Cependant, des *assistants à la preuve* (comme PVS par exemple) permettent d'accomplir certaines classes de preuve tout en maintenant l'intervention de l'opérateur humain. L'assistant de preuve fournit alors un certain nombre de lemmes intermédiaires qui permettront de prouver le théorème et donc d'affirmer que le modèle du système satisfait la propriété (exprimée par la formule du théorème).

Cette technique traite avec des systèmes complexes de taille considérable (espace d'états infinis). Elle couvre également un très grand nombre de propriétés. En grande partie, parce qu'elle fait constamment appel à l'opérateur humain pour palier les lacunes de ses stratégies de preuves automatiques. Cependant, la technique présente deux principaux inconvénients majeurs qui possèdent la même cause : l'indécidabilité. Le premier est l'absence de garantie sur le résultat ; le théorème que l'on vent démontrer peut s'avérer indécidable. Le deuxième problème est que l'assistant de preuve nécessitera toujours l'intervention humaine.

5.3.2 Model-checking

La vérification de modèles peut prendre deux formes : partielle appelée simulation ou complète appelée Model-checking.

La simulation est une technique de vérification semi-formelle utilisée pour vérifier les systèmes matériels ou logiciels pour détecter les erreurs de comportement assez tôt pendant la phase de conception. Le problème posé par cette technique est qu'elle ne permet pas de couvrir toutes les possibilités d'exécution du système en cours de vérification. Cette déficience est traitée par les

techniques de vérification formelle de modèles appelées Model-checking ou exploration de l'espace d'états du système.

La model-checking est une procédure automatique permettant de vérifier qu'un modèle d'un système donné satisfait ou non une spécification particulière. Cette procédure ne se fait pas par *déductions* comme dans le cas des assistants de preuve, mais grâce à des algorithmes tirant profit des spécifications utilisées pour décrire le système et ses propriétés.

Le model-checking repose sur la modélisation formelle du système généralement par des machines d'états finis (automates, structures de kripke, etc.) et la spécification des propriétés à vérifier par des formules logiques (logique classique, logique temporelle, etc.). L'algorithme de vérification combine le modèle et la formule pour calculer l'ensemble des états *accessibles* qui vérifient cette formule ; La propriété est vérifiée s'il y a au moins *chemin* qui relie *l'état initial* à l'ensemble des états qui *vérifient* cette propriété.

Le model-checking est une procédure complètement automatisée et rapide, et peut être utilisé même pour vérifier des spécifications partielles. Il produit également des contre-exemples représentant généralement des erreurs subtiles de conception. Son inconvénient majeur est le problème d'explosion combinatoire causé par une complexité accrue des systèmes développés et confrontée à des limites sérieuses de ressources matérielles. Ce problème peut être partiellement surmonté par l'utilisation des techniques du model-checking symbolique (des représentations plus abstraites des éléments constituant le système à vérifier). L'implémentation du model-checking symbolique peut être réalisée par utilisation des BDD (*Binary Decision Diagram*) par exemple.

A l'heure actuelle, on utilise la preuve assistée conjointement avec le model-checking par le biais d'abstractions finies du système réalisées de façon automatique afin de réduire la complexité du système et de rester dans le cadre d'une théorie décidable. En pratique, ces techniques formelles, plutôt d'origines académiques commencent à ce développer dans les entreprises et des outils basés sur ces techniques ont déjà fait leurs preuves sur des exemples industriels concrets. Des exemples comme PVS, Coq, etc. (s de preuve), Spin, Java Pathfinder, Kronos, Uppaal, etc. (model-checkers).

Ces méthodes n'ont pas la prétention de pouvoir certifier le comportement exact de n'importe quel système. En fait, elles s'appliquent à des modèles et non pas à des systèmes réel. Des modèles qui ne permettent pas toujours de représenter tout ce qui peut se passer dans la réalité ; les systèmes réel sont influencés par un environnement non contrôlé, alors que les modèles de ces systèmes ne peuvent refléter qu'une partie minime du comportement de cet environnement. Le test classique vient toujours en complément de ce genre de méthodes : il permet de mettre en condition réelles les systèmes, ce que ne permettent pas les model-checkers et pas toujours les assistants de preuve.

5.4 Techniques de spécification formelle

Principalement, deux classes de techniques de spécification sont considérées : techniques formelles et semi formelles (les spécifications informelles sont omises volontairement). Les techniques semi formelles offrent des spécifications conviviales concédant des représentations graphiques simples des systèmes à développer. Cependant, l'utilisation de certains concepts informels rend la spécification obtenue imprécise ce qui peut tromper les procédures de vérification.

Les techniques formelles se basent, par contre, sur des fondements mathématiques et disposent d'une axiomatique permettant une spécification précise et donc des opérations de vérification patentes. Plusieurs techniques de spécification ont été suggérées dans la littérature ; certaines se basent sur la théorie des ensembles (Z, VDM, ...), d'autres sur les logiques temporelles (LTL, CTL, TCTL …), etc. Plusieurs entre elles ont prouvé une efficacité dans le milieu industriel.

A.1 Introduction

Rappel des objectifs des processus.

Un processus définit QUI fait QUOI, QUAND et COMMENT pour atteindre un certain objectif. Les principaux objectifs peuvent être :

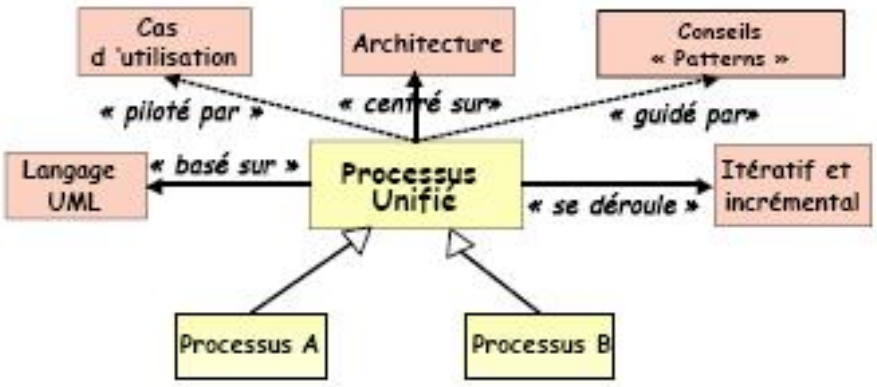
- Construction des modèles d'un ou plusieurs systèmes,
- Organisation et gestion de la totalité du cycle de vie du projet,
- Gestion des risques,
- construction répétitive de produits logiciels de qualité constante.

A.2 Principes du processus unifié

Comme l'expérience l'a démontré avec l'histoire d'unification des méthodes objet, il est clair qu'il n'y a pas un seul processus de développement standard et utilisable sur toutes les classes de systèmes et par tous les développeurs de systèmes logiciels. Cependant, des caractéristiques communes et essentielles peuvent être mises en avant.

Un processus d'ingénierie logicielle doit être :

- Piloté par les cas d'utilisation,
- Centré sur l'architecture,
- Itératif et incrémental,
- Guidé par les Design Patterns.



Ces caractéristiques sont génériques ; le processus unifié ne peut être utilisé directement et nécessite une spécialisation qui tient compte des facteurs organisationnels et techniques du domaine.

1. *Piloté par les cas d'utilisation*

A partir des cas d'utilisation, il est possible de créer toute une série de modèles UML :

- Les modèles d'analyse *spécifient* les cas d'utilisation,
- Les modèles de conception *réalisent* les cas d'utilisation,

- Les modèles de déploiement *distribuent* les cas d'utilisation,
- Les modèles d'implémentation *implantent* les cas d'utilisation,
- Les modèles de tests *vérifient* les cas d'utilisation.

2. *Centré sur l'architecture*

L'architecture regroupe les différentes vues du système qui doit être construit. Elle doit prévoir la réalisation de tous les cas d'utilisation. L'architecture et les cas d'utilisation évoluent de façon concomitante.

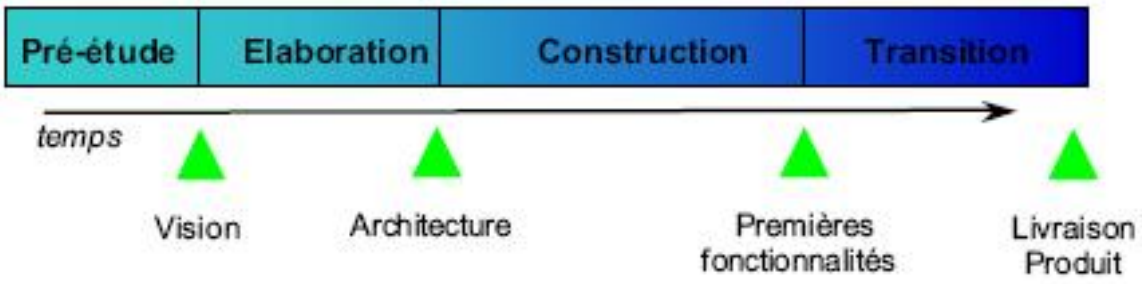
- Recherche de la *forme générale* du système dès le début,
- Approche systématique pour trouver une *bonne architecture* qui soit :
  - support des cas d'utilisation,
  - adaptable aux changements,
  - pour et avec la réutilisation,
  - compréhensible intuitivement.

3. *Itératif et incrémental*

Découpage du projet en *'mini-projets'* : des ITERATIONS qui donnent lieu à un INCREMENT.

- Chaque itération traite un certain nombre de cas d'utilisation en donnant priorité aux risques majeurs.
- Une itération reprend les livrables dans l'état où les a laissé l'itération précédente et les enrichit progressivement (incrémental).
- Les itérations sont regroupées dans une *phase*. Chaque phase est ponctuée par un repère qui marquera la décision que les objectifs (fixés préalablement) ont été remplis.
- Les premières itérations sont des prototypes qui définissent le noyau de l'architecture.

Les phases. Le processus unifié comme le montre la figure suivante propose quatre phases sur une échelle temporelle : Pré-étude (*Inception*), Elaboration, Construction et Transition.



a. Pré-étude :

- Délimiter la portée du système,
- Définir les frontières et identifier les interfaces,
- Développer les cas d'utilisation,
- Décrire et esquisser l'architecture candidate,

- Identifier les risques les plus sérieux,
- Démontrer que le système proposé est en mesure de résoudre les problèmes ou de prendre en charge les objectifs fixés.

Le résultat ➔ **Vision :** Glossaire, Détermination des **parties prenantes** et des utilisateurs, Détermination de leurs **Besoins fonctionnels** et **non fonctionnels**, **Contraintes** de conception.

b. Elaboration :

- Spécifier des fondements de l'architecture et créer une architecture de référence,
- Identifier les risques qui peuvent bouleverser le plan, le coût et le calendrier,
- Définir les niveaux de qualité à atteindre,
- Formuler les cas d'utilisation pour couvrir environ 80% des besoins fonctionnels et de planifier la phase de construction,
- Planifier le projet, élaborer une offre abordant les questions de calendrier, de personnel et de budget.

Le résultat ➔ **Architecture :** Document d'architecture Logicielle, différentes vues selon la partie prenante, une architecture candidate, comportement et conception des composants du système.

c. Construction :

- Etendre l'identification, la description et la réalisation des cas d'utilisation,
- Finaliser l'analyse, la conception, l'implémentation et les tests,
- Préserver l'intégrité de l'architecture,
- Surveiller les risques critiques et significatifs identifiés dans les deux premières phases et réduire les risques.

Le résultat ➔ **Produit**. Premières fonctionnalités.

d. Transition :

- Préparer les *activités*,
- Recommandations au client sur la mise à jour de l'environnement logiciel,
- Elaborer les manuels et la documentation concernant la version du produit,
- Adaptation du logiciel,
- Correction des anomalies liées au béta test,
- Dernières corrections.

Le résultat ➔ **Livraison** du produit aux utilisateurs.

Les Activités.

1. Modélisation métier :

- Compréhension de la structure et la dynamique de l'organisation,
- Comprendre les problèmes posés dans le contexte de l'organisation,
- Conception d'un glossaire.

2. **Recueil et expression des besoins :**

- Auprès des clients et parties prenantes du projet,
- Ce que le système doit faire,
- Expression des besoins dans les cas d'utilisation,
- Spécifications des cas d'utilisation en scénarios,
- Spécifications fonctionnelles et non fonctionnelles,
- Planification et prévision de coût,
- Production de Maquettes de l'IHM :
    - La production de maquettes peut être réalisée avec n'importe quel outil graphique :
        - ce sont de simples dessins d'écrans et descriptions de contenu de fenêtres,
        - prototype d'interface généré par un outil,
        - Intéressant pour décrire les interfaces avec des acteurs non humains.
    - Pourquoi si tôt dans le processus ?
        - Aide à la description et validation des cas d'utilisation,
        - moyen de communication avec le client,
        - permet l'identification de classes,

3. **Analyse et conception :**

- Transformation des besoins utilisateurs en modèles UML,
- Modèle d'analyse et modèle de conception.

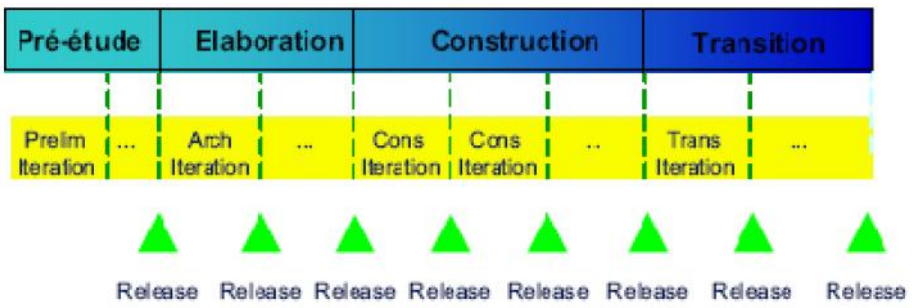
4. **Implémentation :**

- Développement itératif,
- Découpage en couches,
- Composants d'architecture,
- Retouche des modèles et des besoins,

5. **Test, Déploiement,** Configuration et gestion des changements, Gestion du projet et de l'environnement.

Les Itérations.

Une itération est une séquence d'activités selon un plan préétabli et des critères d'évaluation, résultant en un produit exécutable.



4. Guidé par les Design patterns

La notion de pattern désigne une solution générique d'un problème récurrent.

- Les bonnes pratiques et solutions.
- La plupart des patterns visent la réutilisabilité et l'extensibilité
- Un moyen de transférer des compétences.

Nom du Pattern
Problème **(problème récurrent de conception OO)**
Solution (exprimée en UML)
Bénéfices

Fig. A.1 structure template du pattern.

La figure suivante montre l'exemple du design pattern *'composite'* qui est un pattern de structure (chapitre suivant) permettant d'établir des structures arborescentes entre des objets et les traiter uniformément.

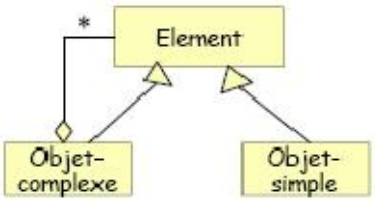
Nom du Pattern	*Composite Pattern*
Problème (problème récurrent de conception OO)	*Représenter des objets complexes*
Solution (exprimée en UML)	
Bénéfices	- Construction récursive de hiérarchies - Considérer de manière homogène les nœuds simples et complexes

Fig. A.2 exemple du pattern *composite*.

B.1 Définition des patterns

Le terme pattern a été initialement introduit dans le domaine d'architecture par C. Alexander qui a défini 253 patrons de conception architecturaux (64, 77, 79).

Un patron (selon C. Alexander) décrit à la fois un **problème** qui se produit très fréquemment dans votre environnement et l'architecture de la **solution** à ce problème de telle façon qu'on puisse **utiliser** cette solution plusieurs fois sans avoir à l'adapter de la même manière.

➔Décrire avec succès des **types de solutions** récurrentes à des **problèmes** communs dans des **types de situations**.

B.2 Classification des patterns

Les patterns utilisés dans l'ingénierie logicielle peuvent être identifiés dans différents domaines et à différents niveaux :

Patterns Architecturaux. Schémas d'organisation structurelle de logiciels (*pipes*, *filters*, …)

Design Patterns. Caractéristiques clés d'une structure de conception commune à plusieurs applications, de portée plus limitée que les patterns architecturaux. Selon leurs portées (classe ou objet), les design patterns peuvent être réutilisés par héritage (classe) ou par composition (objet).

Anti-patterns. Mauvaise solution ou comment sortir d'une mauvaise solution.

Coding patterns. Solution liée à un langage particulier.

Analysis Patterns. Schémas d'analyse (vérification) de conception.

Specification Patterns. Schémas de spécification de propriétés à vérifier par model-checking.

Patterns Organisationnel. Organisation de tout ce qui entoure le développement d'un logiciel (humains)…

B.3 Objectifs des design patterns

Les design patterns présentent de nombreux avantages, dont les plus importants sont :

- Documentation d'une expérience éprouvée de conception,
- Identification et spécification d'abstractions de haut niveau,
- Vocabulaire commun et aide à la compréhension de principes de conception,
- Moyen de documentation de logiciels,
- Aide à la construction de logiciels à caractéristiques précises, de logiciels complexes et hétérogènes,
- produit logiciel plus simple à modifier et moins fragiles.

B.4 Classification des design patterns

l'ouvrage des GoF (gang of four : ses quatre auteurs) Design Patterns: Elements of Reusable Object-Oriented Software**,** Addison-Wesley Publishing Company, 1994. A défini la première classification de design patterns. Initialement, 23 patterns ont été répartis dans trois catégories :

- *Creational patterns* : processus de création d'objet,
- *Structural patterns*, composition et structure statiques des classes et des objets,
- *Behavioral patterns*, interactions dynamiques entre classes e objets.

Creational Patterns
Abstract Factory
Builder
Factory Method
Prototype
Singleton

Structural Patterns
Adapter
Bridge
Composite
Decorator
Façade
Flyweight
Proxy

Behavioral Patterns
Chain of Responsibility
Command
Interpreter
Iterator
Mediator
Memento
Observer
State
Strategy
Template Method
Visitor

Plusieurs autres patterns ont été ensuite proposes par d'autres auteurs.

- L'ouvrage *'Data Access Patterns'* de *Clifton Nock* a introduit 4 *decoupling patterns*, 5 *resource patterns*, 5 *I/O patterns*, 7 *cache patterns*, et 4 *concurrency patterns*.
- Autres langages de patterns incluent *telecommunications patterns*, *pedagogical patterns*, *analysis patterns*, etc.

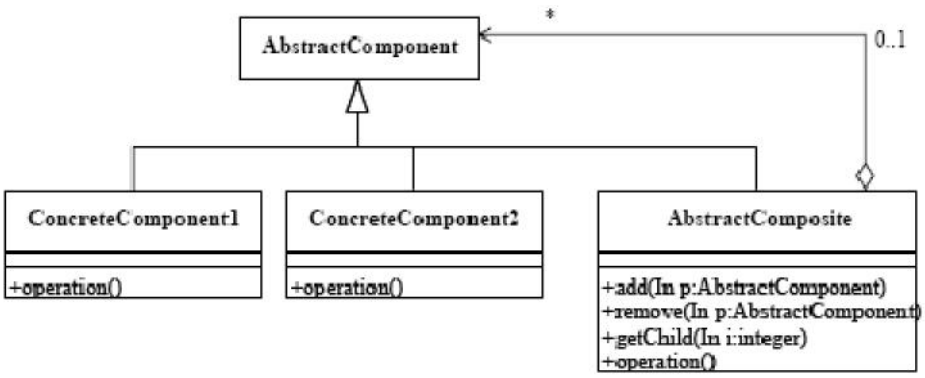
B.4.1 Design Patterns de création

1. Abstract Factory : interface pour la création de familles d'objets sans spécifier les classes concrètes.
2. Builder : séparation de la construction d'objets complexes de leur représentation afin qu'un même processus de construction puisse créer différentes représentations.
3. Factory Method : définition d'une interface pour la création d'objets associés dans une classe dérivée.
4. Prototype : spécification des types d'objet à créer en utilisant une instance prototype.
5. Singleton : comment assurer l'unicité de l'instance d'une classe.

```
// Only one object of this class can be created
class Singleton
{
    private static Singleton _instance = null;
    private Singleton() { fill in the blank }
    public static Singleton getInstance()
    {
        if ( _instance == null ) _instance = new Singleton();
        return _instance;
    }
    public void otherOperations() { blank; }
}
class Program
{
    public void aMethod()
    {
    X = Singleton.getInstance();
    }
}
```

B.4.2 Design Patterns de structure

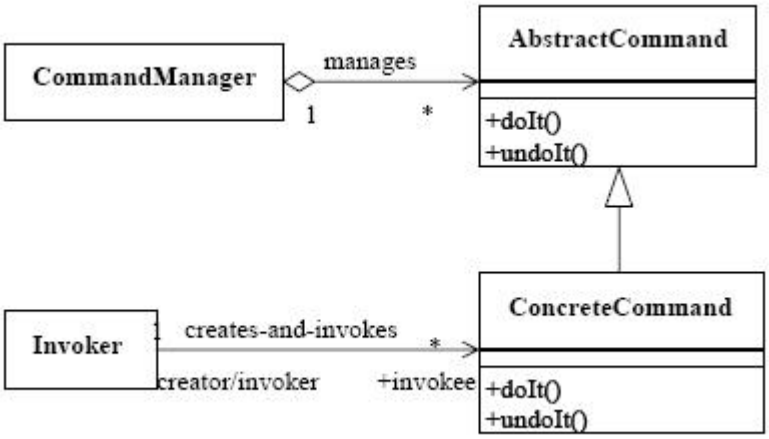
6. Adapter : traducteur adaptant l'interface d'une classe en une autre interface convenant aux attentes des classes clientes.
7. Bridge : découplage de l'abstraction de l'implémentation pour faire varier les deux indépendamment.
8. Composite : structure pour la construction d'agrégations récursives.



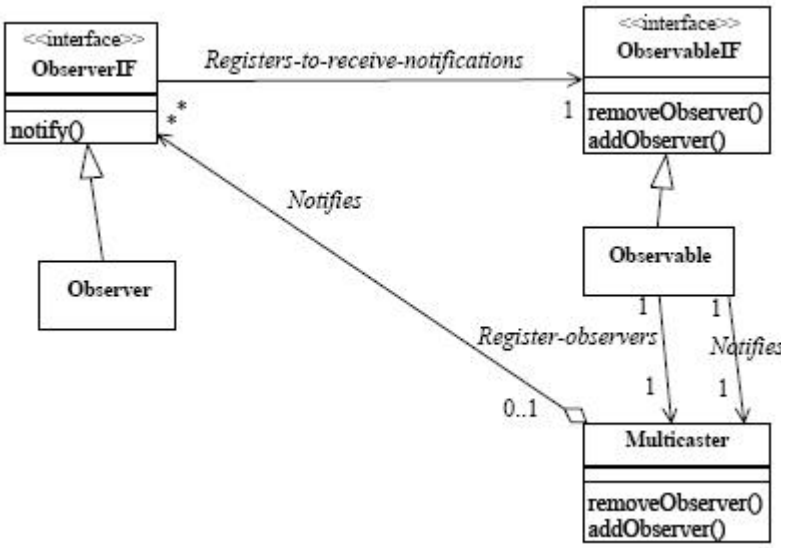
9. Decorator : extension d'un objet de manière transparente.
10. Façade : unification de plusieurs interfaces de sous-systèmes.
11. Flyweight : partage efficace de plusieurs objets.
12. Proxy : approximation d'un objet par un autre.

B.4.3 Design Patterns de comportement

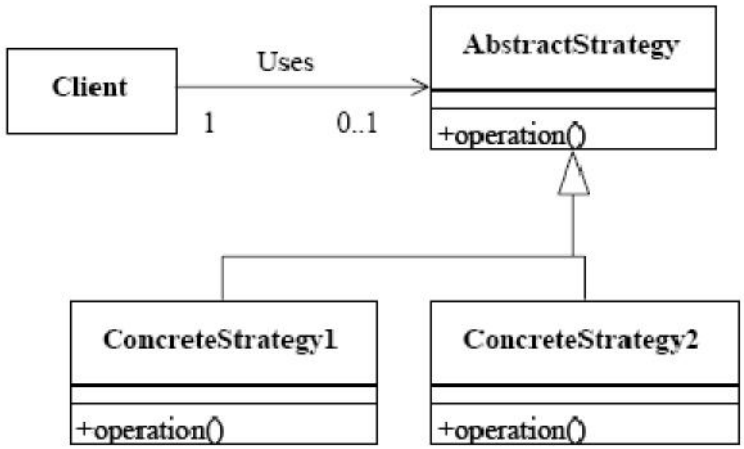
13. Chain of Responsibility : délégation des requêtes à des responsables de services.
14. Command : encapsulation de requêtes par des objets afin de permettre à un objet de traiter plusieurs types de requêtes.



15. Interpreter : étant donné un langage, représentation de la grammaire le définissant pour l'interpréter.
16. Iterator : parcours séquentiel de collections.
17. Mediator : coordination d'interactions entre des objets associés.
18. Memento : capture et restauration d'états d'objets.
19. Observer : mise à jour automatique des dépendants d'un objet.



20. State : permettre à un objet de modifier son comportement lorsque son état interne change.
21. Strategy : abstraction pour sélectionner un algorithme parmi plusieurs.



22. Template method : définition d'un squelette d'algorithme dont certaines étapes sont fournies par une classe dérivée.
23. Visitor : représentation d'opérations devant être appliquées à des éléments d'une structure hétérogène d'objets.

B.5 Style de description de Patterns

Différentes structures templates peuvent être utilisées pour la description des patterns. La structure que nous présentons semble satisfaisante :

Pattern name
Recurring problem Description du problème que traite le pattern
Solution Approche générale du pattern
UML model Spécification UML de la solution
Use Example(s) Exemples du pattern dans un langage de programmation