

UNIVERSITÉ DES SCIENCES ET DE LA TÉCHNOLOGIE HOUARI  
BOUMEDIENE

---

## GÉNIE LOGICIEL

---

Chargés de cours :

Section A : M. HAMMAL Youcef

Section B : Mme MAHDAOUI Latifa

Section C : Mme BOUSSAÏD Ilhem



Licence 3 académique

Année universitaire 2013/2014

---

---

# Table des matières

---

<b>1</b>	<b>Introduction au Génie Logiciel</b>	<b>1</b>
1.1	Introduction . . . . .	1
1.2	La crise du logiciel . . . . .	1
1.3	Le Génie Logiciel . . . . .	4
1.3.1	Qualité exigée d'un logiciel . . . . .	5
<b>2</b>	<b>Cycle de vie d'un logiciel</b>	<b>8</b>
2.1	Les étapes d'un cycle de vie de logiciels . . . . .	8
2.2	Les différents modèles de cycle de vie . . . . .	8
2.2.1	Modèle de la cascade . . . . .	8
2.2.2	Modèle en V . . . . .	11
2.2.3	Approches évolutives : Maquettage & Prototypage . . . . .	12
2.2.4	Modèle par incrément	13
2.2.5	Modèle en Spirale . . . . .	14
2.2.6	Modèles hybrides . . . . .	16
<b>3</b>	<b>UML</b>	<b>17</b>
3.1	Modélisation . . . . .	17
3.1.1	Qu'est ce qu'un modèle ? . . . . .	17
3.1.2	Rôles d'un modèle en Génie Logiciel . . . . .	18
3.1.3	Langages de modélisation . . . . .	18
3.1.4	Histoire des modélisations par objets . . . . .	19
3.2	UML . . . . .	19
3.2.1	Objectifs d'UML . . . . .	20
3.2.2	Les diagrammes d'UML . . . . .	20
<b>4</b>	<b>Principes généraux d'analyse et spécification des besoins</b>	<b>25</b>
4.1	Analyse & Définition des Besoins . . . . .	25
4.2	Cahier des Charges . . . . .	26
4.3	Structure d'un Cahier des Charges . . . . .	26
4.4	Modèle Conceptuel du Système . . . . .	27
4.5	Besoins fonctionnels . . . . .	29
4.5.1	Définition des besoins en langage naturel . . . . .	30
4.5.2	Définition des besoins en langage structuré . . . . .	30
4.5.3	Langages de spécification . . . . .	31
4.6	Besoins non fonctionnels . . . . .	31
4.7	Validation des Besoins . . . . .	32
4.8	Diagramme de cas d'utilisation . . . . .	33
4.8.1	Cas d'utilisation . . . . .	33

4.8.2	Acteur . . . . .	35
4.8.3	Relations entre cas d'utilisation et acteurs (association) . . . . .	36
4.8.4	Relations entre cas d'utilisation . . . . .	37
4.8.5	Recenser les cas d'utilisation . . . . .	40
4.8.6	Identifier les scénarios . . . . .	40
4.8.7	Spécification du cas d'utilisation ( <i>use case template</i> ) . . . . .	42
4.9	Diagramme d'activité . . . . .	42
4.9.1	Éléments d'un diagramme d'activité . . . . .	43

# INTRODUCTION AU GÉNIE LOGICIEL

---

## 1.1 Introduction

La structure des coûts des systèmes informatiques s'est profondément modifiée depuis les années 60. Dans un premier temps, les coûts matériels ont été prépondérants mais les progrès considérables de la technologie électronique (apparition des mini-ordinateurs puis des micro-ordinateurs) a entraîné des baisses de coûts du matériel dans des proportions jamais vues conduisant à une utilisation très accrue des ordinateurs dans des domaines très divers et de plus en plus complexes et ayant des missions de plus en plus critiques (télécommunications, systèmes temps réel,...).

Malheureusement, les coûts des logiciels n'ont pas diminué autant que ceux du matériel. Pour s'en convaincre, il suffit de constater qu'ils représentent actuellement environ 80% du coût total d'un système informatique. C'est dans ce contexte que l'un des buts du Génie Logiciel est de diminuer sensiblement les coûts de développement (et de maintenance) de ces gros systèmes qui sont immenses.

**Exemple 1.1.** *Les coûts de développement de logiciels aux USA pendant les années 77 dépassaient les 50 milliards de dollars. Aux années 88, ils étaient déjà de l'ordre de 100 milliards de dollars.*

## 1.2 La crise du logiciel

Cette fameuse crise est apparue avec l'avènement des ordinateurs de la 3<sup>ème</sup> génération (années 60) et s'est accentuée avec l'ère de la micro-informatique. En effet, leur coût et leur puissance ont permis la prolifération de l'informatique dans beaucoup de domaines avec la réalisation de grands systèmes logiciels. Les premières expériences de construction de ce type de systèmes ont montré que les méthodes de développement de l'époque n'étaient pas adéquates.

Donc, il n'était plus possible de se contenter d'adapter les techniques applicables à de petits systèmes car les problèmes rencontrés dans le développement des gros systèmes logiciels ne sont pas directement comparables à ceux relatifs aux petits. La complexité des petits programmes peut être appréhendée par une seule personne qui peut avoir à l'esprit tous les détails de la conception et de la réalisation. Quant aux spécifications, elles peuvent rester informelles

et les effets de toutes modifications sont en général évidents. En revanche, la complexité des grands systèmes est telle qu'il est impossible à une seule personne d'avoir présent à l'esprit tous les détails du projet et de les mettre à jour. En réalité, ce sont des équipes d'analystes, de concepteurs, de développeurs et de testeurs qui doivent collaborer pour parvenir à réaliser ce type de projet.

### Notations :

**HA** : Homme-Année (unité de coût de logiciel)

**HM** : Homme-Mois

**Kls** : un millier de lignes de code source (unité de mesure de la taille d'un logiciel)

**Ls** : une ligne de code source.

TABLEAU 1.1:

Nature du Logiciel	Coût en HA	Volume	Remarque
Compilateurs :			Délais :
Pascal	10 ha	20 à 30 kls	1 à 2 ans
COBOL, FORTRAN	80 à 100 ha	100 à 200 kls	2 à 3 ans
ADA	150 à 200 ha	>300 kls	> 3 ans
SGBD Relationnel (ORACLE, DB2, ...)	300 à 500 ha	300 à 600 kls	Délais : de 3 à 5 ans incluant une 1 <sup>ère</sup> version.
Grands systèmes temps réel :			Délais :
Navette Spatiale	>1000 ha	2200 kls	6 ans, écrit en HAL
SAFEGUARD <sup>1</sup>	5000 ha	2260 kls	7 ans, écrit en pl1
SABRE <sup>2</sup>	955 ha	960 kls	10 ans pour un MTTF <sup>3</sup> de 55 h.
Systèmes constructeurs : VMS, GCOS7	d'exploitation	2500 à 5000 ha	Durée de vie 15 à 20 ans dont au moins 5 ans de délai pour une 1 <sup>ère</sup> version.
Systèmes industriels : GPAO, MRP, ...	150 à 500 ha	500 à 1000 kls	Bases de données importantes.

**Remarque 1.1.** La productivité de développement peut être exprimée comme étant le rapport entre la taille du logiciel et son coût. Cet indicateur dénote la difficulté de fabrication du logiciel. (Unité de mesure = ls/hm)

Grosso modo, les projets souffraient de :

- Retards dans les livraisons (parfois des années).
- Surcoûts considérables (dépassant largement les budgets prévus).
- Produits qui ne répondent pas aux besoins des usagers (peu fiables, peu performants, difficiles à maintenir, etc.).

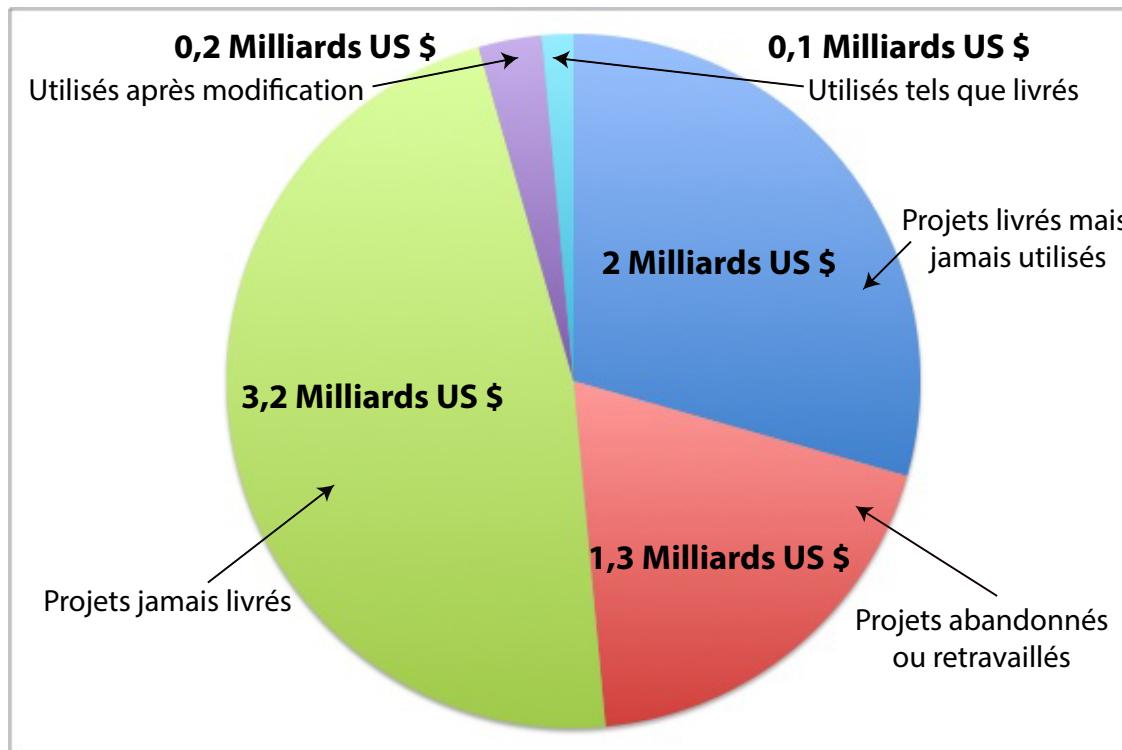


FIGURE 1.1: Répartition de 6,8 Milliards US \$ en projets logiciels commandités par le gouvernement US en 1979.

- Produits peu sûrs (comportant des erreurs dont le nombre et l'emplacement sont souvent inconnus et ayant des répercussions graves sur la mission du système).

Ces problèmes s'apparentent avec une croissance exponentielle de la demande de logiciels et avec la croissance dans la complexité des systèmes construits dans un contexte caractérisé par la diminution des prix du matériel et le gonflement incontrôlable des prix de logiciels.

**Remarque 1.2.** *De 1965 à 1995, en 30 ans, le volume de chaque logiciel a été multiplié par 100, alors que la productivité n'augmentait que d'un facteur de 3. Attention, il ne suffit pas d'augmenter l'investissement en hommes pour augmenter d'autant la productivité. Donc, il est évident que de nouvelles techniques formelles de spécification, de conception et de réalisation sont nécessaires ; Chaque étape du projet doit être correctement évaluée et documentée et une gestion soignée est essentielle.*

### Quelques Bogue :

- Echec de la mission de la sonde Mariner vers Vénus dans les années 71 : (perdue dans l'espace) passage à 5.000.000 Km de la planète au lieu de 5.000 Km prévus.  
**Cause :** Remplacement d'une virgule par un point.
- OS 360 : Système d'exploitation pour la série 360 des ordinateurs IBM. Livré en retard avec beaucoup de bogues.

- Perte de satellites dans les années 70.

**Cause** : frappe de  $+I$  à la place de  $+1$  dans une instruction d'itération du programme source (le langage Fortran admet sans le signaler des identificateurs non déclarés, leur valeur était alors aléatoire).

- Missile Ami : Navire de Guerre Britannique coulé par un Exocet français de la marine Argentine au cours de la guerre des Malouines. Le vaisseau anglais n'avait pas activé ses défenses.

**Cause** : l'Exocet n'est pas répertorié comme missile ennemi.

- Socrate : système de réservation de places de la SNCF. Ses plantages fréquents, sa mauvaise ergonomie, le manque de formation préalable du personnel, ont amené une défection importante et durable de la clientèle vers d'autres moyens de transport.

**Cause** : rachat par la SNCF d'un système de réservation de places d'une compagnie aérienne, sans réadaptation totale au cahier des charges du transport ferroviaire.

- Echec du 1<sup>er</sup> lancement d'Arian V : (explosion en vol)

Le système informatique conçu pour Ariane IV n'a pas été convenablement réadapté pour Ariane V (plan de vol n'a pas été changé). Coût du programme de développement d'Arian V : 38 Milliards de Francs.

- Perte de Mars Climate Orbiter : Le 23 septembre 1999, après 9 mois de voyage. Coût : 120 Milliards US\$

**Cause** : confusion entre pieds et mètres.

- Bogue 2000 : dysfonctionnement de plusieurs systèmes informatiques.

**Cause** : la donnée "année" était codée sur deux caractères, pour gagner un peu d'espace.

### 1.3 Le Génie Logiciel

Dans un souci de pallier les problèmes suscités et dans un souci d'**augmenter la productivité des équipes de développement et d'améliorer la qualité des produits**, des procédés de fabrication de ces différents logiciels ont été proposés sous le nom de Génie Logiciel afin de s'assurer que :

- ce qui est fabriqué par le maître d'œuvre répond aux besoins de celui qui les a formulés (le maître d'ouvrage ou le représentant du client final)
- Les coûts et les délais de réalisation restent dans les limites fixés au départ.
- Le contrat de service sera effectivement respecté (performance, sûreté de fonctionnement, sécurité, ...) lors de l'exploitation future du logiciel.
- Aussi, s'assurer de l'aptitude d'évolution lors de l'apparition de nouveaux besoins est une caractéristique importante ayant un effet direct sur la durée de vie du système d'où une

répercussion directe sur son coût d'amortissement.

**Remarque 1.3.** Ce terme est né en 1968 (7-11 octobre 1968) en pleine période de crise de logiciel sous le nom anglo-saxon : *software engineering* et sous le parrainage de l'OTAN.

Cette nouvelle discipline de l'ingénierie est donc concernée par le développement de logiciels de *qualité* et *coûts* raisonnables et dans les *délais* prévus. Pour ce faire, Elle s'occupe de l'étude de l'ensemble des méthodes et techniques qu'il faut réunir pour spécifier, construire, distribuer et maintenir les logiciels qui soient :

- **Sûrs** : réagissant de façon déterministe à toutes les sollicitations éventuellement entachées d'erreurs, qui correspondent à sa mission.
- **Conviviaux** : adaptés aux capacités réelles et non supposées des usagers (ergonomie).
- **Evolutifs** : s'adaptant aux nouveaux besoins dans des délais raisonnables.
- **Economiques** : réalisant l'optimum entre le service rendu et les coûts de développement et de la maintenance initialement annoncés.

### 1.3.1 Qualité exigée d'un logiciel

Si le génie logiciel est l'art de produire de *bons logiciels*, il est par conséquent nécessaire de fixer les critères de qualité d'un logiciel.

#### QU'EST CE QU'UN LOGICIEL ?

Par logiciel on n'entend pas seulement l'ensemble des programmes informatiques (du **code**) associés à une application ou à un produit, mais également un certain nombre de **documents** se rapportant à ces programmes et nécessaires à leur **installation, utilisation, développement et maintenance** : spécification, schémas conceptuels, jeux de tests, mode d'emploi, etc.

Pour les grands systèmes, l'effort nécessaire pour écrire cette documentation est souvent aussi grand que l'effort de développement des programmes eux-mêmes.

En plus des fonctionnalités requises, d'autres facteurs caractérisent la qualité d'un logiciel :

### **Fiabilité**

- Fonctionnement du logiciel conforme aux besoins des clients dans les conditions d'utilisation spécifiées par ces derniers.
- Absence des bugs.
- Intégrité (aptitude d'un logiciel à protéger son code et ses données contre des accès non autorisé).
- Précision.
- Tests réussis pour tous les cas.
- Uniformité de conduite.

**Efficacité** : Optimisation de la consommation des ressources, performances optimales :

- Economie de mémoire
- Rapidité d'exécution...

**Ergonomie** : Interface utilisateur s'adaptant aux capacités réelles des usagers

- Facilité d'utilisation. La facilité d'emploi est liée à la facilité d'apprentissage, d'utilisation, d'interprétation des erreurs et de rattrapage en cas d'erreur d'utilisation.
- Accessibilité.
- Documentation pour l'utilisateur.

**Facilité de Maintenance (maintenabilité)** : les logiciels ayant une longue durée de vie, doivent être conçus et réalisés de façon à minimiser les coûts des corrections d'erreurs non décelés dans les phases antérieures ainsi que ceux des adaptations aux évolutions éventuelles. On en distingue ainsi 3 types :

1. *Maintenance corrective* : pour corriger les erreurs faites lors du développement et non décelées par les tests.
2. *Maintenance adaptative* : pour adapter les fonctionnalités du logiciel aux changements technologiques *soft & hard*.
3. *Maintenance évolutive* : pour réaliser et intégrer au logiciel de nouvelles fonctionnalités exigées par l'évolution de son environnement.

La maintenabilité exige que le développement du logiciel respecte certains critères (qualités auxquelles s'intéressent les développeurs)

- Conception Modulaire assurant un forte cohésion intra-modulaires et faible couplage inter-modulaires
- Structuration du code source
- Lisibilité du code source

- Extensibilité (Facilité d'extension)
- Documentation technique lisible, structurée, extensible ...
- Portabilité (Utilisation d'un langage standardisé, Indépendance vis-à-vis du matériel et du système d'exploitation).

**D'autres qualités spécifiques :** Sûreté de fonctionnement, sécurité informatique, ... Remarque : Il est difficile d'optimiser tous ces facteurs en même temps car certains s'excluent mutuellement.

**Exemple 1.2.** *offrir une meilleure interface utilisateur peut réduire l'efficacité. Il est clair que dans ce cas, on doit effectuer des choix antinomiques. On est amené à trouver des compromis entre ces objectifs lors de la définition des besoins du système logiciel.*

**Exemple 1.3.** *Dans les systèmes temps réel, l'efficacité est de première importance. Toutefois, ceci risque d'accroître les coûts de manière exponentielle.*

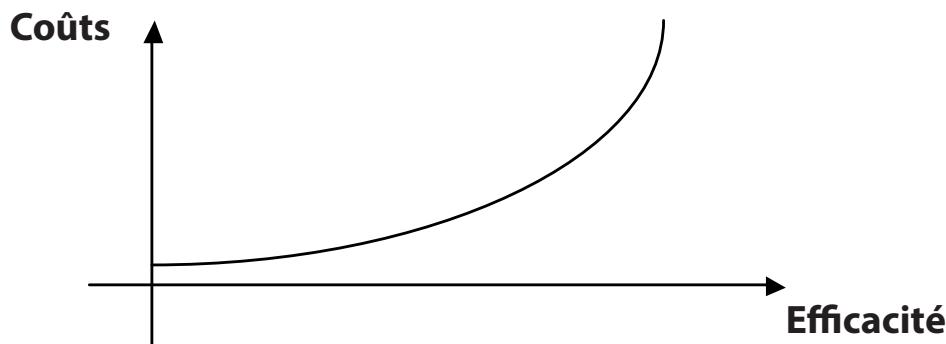


FIGURE 1.2:

## CYCLE DE VIE D'UN LOGICIEL

---

### 2.1 Les étapes d'un cycle de vie de logiciels

Comme toute science de l'ingénieur, le génie logiciel perçoit la fabrication des programmes comme un processus à étapes. Le premier article important sur ce sujet fut celui de W.W Royce intitulé "*Managing the development of large software systems*", IEEE WESCON, en août 1970 où il a cherché à caractériser :

- Chacune des étapes successives de la fabrication depuis la commande du logiciel par le maître d'ouvrage jusqu'à sa mise en exploitation. L'ensemble de ces étapes forme ce qu'on appelle le cycle de vie du logiciel
- La logique d'enchaînement des différentes étapes.

### 2.2 Les différents modèles de cycle de vie

#### 2.2.1 Modèle de la cascade

Royce a proposé un premier modèle du cycle de vie (dit de la cascade).

**Caractéristiques du modèle :**

- Cycle de vie linéaire sans aucune évaluation entre le début du projet et la validation
- Le projet est découpé en phases successives dans le temps et à chaque phase correspond une activité principale bien précise produisant un certain nombre de livrables
- On ne passe à l'étape suivante que si les résultats de l'étape précédente sont jugés satisfaisants.
- L'activité d'une étape se réalise avec les résultats fournis par l'étape précédente ; ainsi, chaque étape sert de contrôle du travail effectué lors de l'étape précédente.
- Chaque phase ne peut remettre en cause que la phase précédente ce qui, dans la pratique, s'avère insuffisant.
- L'élaboration des spécifications est une phase particulièrement critique : les erreurs de spécifications sont généralement détectées au moment des tests, voire au moment de la livraison du logiciel à l'utilisateur. Leur correction nécessite alors de reprendre toutes les phases du processus.

**Remarque 2.1.** Bien que les étapes de vie en cascade apparaissent comme un enchaînement séquentiel de phases distinctes, en réalité ces étapes se recouvrent et provoquent des retours d'information.

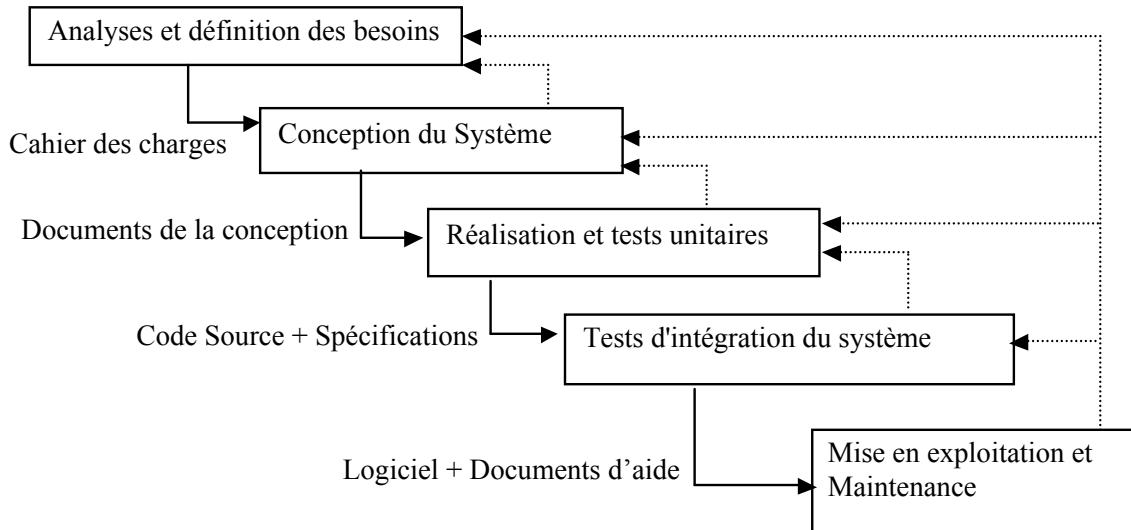


FIGURE 2.1: Le cycle de vie en " Cascade " (waterfall model)

Ce modèle est mieux adapté aux **petits projets** ou à ceux dont les spécifications sont bien connues et fixes. Depuis de nombreuses améliorations et modifications y ont été apportées.

Il existe de nombreux modèles de cycle de vie, les plus courants comportent les phases suivantes :

**Analyse et définition des besoins :** Cette étape comporte deux activités : *Analyse des besoins* et la *spécification globale*. Dans la 1<sup>ère</sup> activité, on étudie le **domaine de l'application**, ainsi que l'**état actuel de l'environnement du futur système** afin d'en déterminer les frontières, le rôle, les ressources disponibles et requises, les contraintes d'utilisation et de performance, etc. Cette étude est menée en collaboration avec les experts du domaine d'application et les futurs utilisateurs.

Quant à la *spécification*, son rôle est d'établir une première description du futur système en se basant sur les **données de l'analyse des besoins** ainsi que **des considérations techniques** et de **faisabilité informatique**. Son résultat est une description de ce que doit faire le logiciel (fonctionnalités) en évitant des décisions prématurées de réalisation.

Cette activité est fortement liée à l'analyse des besoins avec laquelle des échanges importants sont nécessaires. C'est pourquoi, ces deux activités sont souvent regroupées dans une même étape dont les documents produits forment le **cahier des charges**.

**Conception du système :** Après avoir décomposé le système en parties matériel et logiciel selon l'analyse des besoins, on s'occupe dans cette étape à enrichir la description du logiciel de détails d'implémentation afin d'aboutir à une description très proche d'un programme. Elle se déroule souvent en deux étapes : conception architecturale et conception détaillée. L'étape de conception architecturale consiste à décomposer le logiciel en composants plus simples en précisant les interfaces et les fonctions de chaque composant. L'étape de conception détaillée fournit pour chaque composant une description de la manière dont les fonctions du composant sont réalisées : algorithmes, représentations des données.

**Réalisation et tests unitaires :** Cette activité consiste à passer du résultat de la conception détaillée à un ensemble de programme ou de composants de programmes. Notons que tout en développant les composants d'un logiciel, les tests unitaires sont menés pour permettre de vérifier que ces unités répondent à leurs spécifications.

**Tests d'intégration du système :** L'intégration consiste à assembler les composants du logiciel pour obtenir un système exécutable (avec la possibilité de gérer plusieurs variantes du logiciel). Ensuite, on réalise des tests globaux pour être sûr que les besoins logiciels ont été satisfaits. Après quoi, le système est livré au client.

**Mise en exploitation et Maintenance :** Normalement c'est l'étape la plus longue dans le cycle de vie du logiciel. Une fois que le système est installé et mis en service, l'activité maintenance intervient pour corriger les erreurs éventuelles qui n'ont pas été découvertes lors des phases antérieures et/ou aussi améliorer la réalisation des unités du système et à augmenter ses fonctionnalités au fur et à mesure que de nouveaux besoins apparaissent.

**Remarque 2.2.** L'un des buts de l'étape de tests est de s'assurer que le système construit répond aux attentes des utilisateurs. Cette activité s'appelle la validation. Par contre la vérification (l'autre activité apparentée à la validation) consiste à s'assurer que les descriptions successives du logiciel et le logiciel lui-même satisfont la spécification globale.

Le tableau 2.2 (fourni par Boehm, 1975) constitue une approximation très réaliste des coûts relatifs à chaque étape.

Il ressort de ce tableau que les coûts les plus importants sont liés aux activités de spécification/conception et de tests. Une réduction des coûts de développement peut être obtenue en entreprenant des efforts supplémentaires au niveau de ces étapes de début et de fin de ce cycle.

#### Particularité de l'étape de maintenance :

- Cette étape peut entraîner des changements dans toutes les étapes précédentes : l'analyse des besoins, la conception et la réalisation du système ou mettre en évidence la nécessité de faire des tests supplémentaires.

TABLEAU 2.1: Approximation des coûts relatifs à chaque étape du cycle de vie

Type de système	Coût de la phase (%)			
	Besoins/Conception	Réalisation	Test	
Systèmes de commande	46	20	34	
Systèmes embarqués	34	20	46	
Systèmes d'exploitation	33	17	50	
Systèmes scientifiques	44	26	30	
Systèmes de gestion	44	28	28	

- Généralement les coûts de la maintenance des systèmes dont la durée de vie est longue, dépasse de loin les coûts de développement d'un facteur qui peut aller de 2 à 4. Exemple : *le coût de développement d'un système avionique était de 30 dollars/instruction et le coût de maintenance de 4000 dollars/instruction*
- La plus grosse part de ces coûts provient des modifications des besoins et non pas des erreurs. D'où l'importance de couvrir au mieux la définition des besoins des utilisateurs. A cet effet, d'autres variantes du modèle du cycle de vie plus évolutives ont été proposées.

### 2.2.2 Modèle en V

Ce modèle rend explicite le fait que les premières étapes du développement qui ont trait surtout à la construction du logiciel, doivent préparer les dernières étapes qui font intervenir essentiellement les activités de validation et de vérification<sup>1</sup>.

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

En plus des flèches continues qui reflètent l'enchaînement séquentiel des étapes du modèle de la cascade, on remarque l'ajout de flèches discontinues qui représentent le fait qu'une partie des résultats de l'étape de départ est utilisée directement par l'étape d'arrivée, par exemple, à l'issue de la conception architecturale, le protocole d'intégration et les jeux de test d'intégration doivent être complètement décrits.

L'avantage d'un tel modèle est d'éviter d'énoncer une propriété qu'il est impossible de vérifier objectivement une fois le logiciel réalisé.

**Remarque 2.3.** *Le cycle en V est le cycle qui a été normalisé, il est largement utilisé, notamment en informatique industrielle et télécoms. Idéal quand les besoins sont bien connus,*

1. Avec les jeux de tests préparés dans la première branche, les étapes de la deuxième branche peuvent être mieux préparées et planifiées.

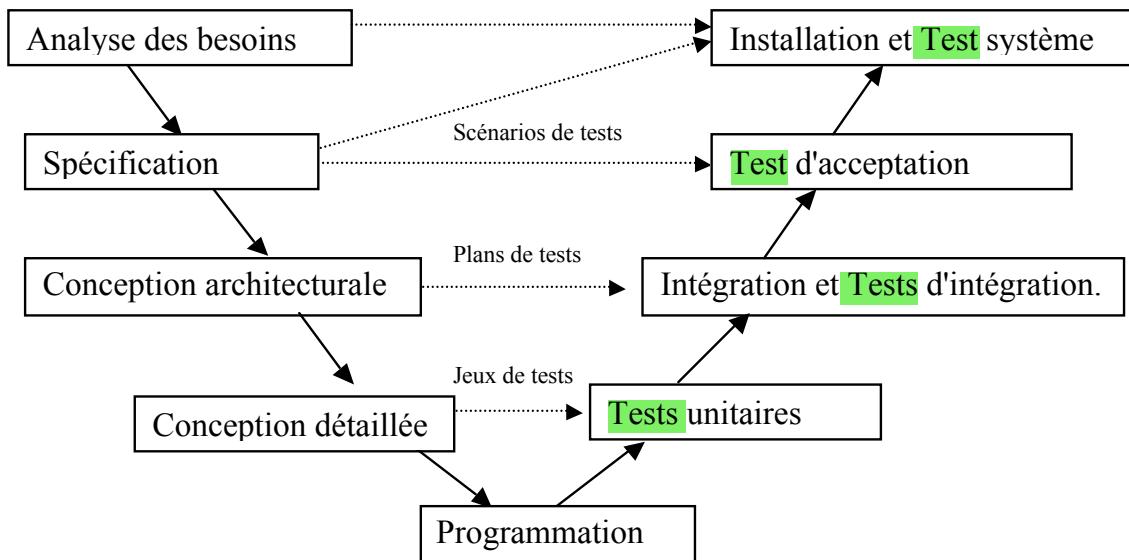


FIGURE 2.2: Le cycle de vie en V

quand l'analyse et la conception sont claires. Il est adapté aux projets de taille et de complexité moyenne.

### 2.2.3 Approches évolutives : Maquettage & Prototypage

La principale difficulté de l'activité de validation est due à l'imprécision des besoins et des caractéristiques du système à développer

#### 2.2.3.1 Maquettes (ou prototype rapide, jetable)

Cette approche consiste donc à produire rapidement une *maquette* (*prototype jetable*) qui constitue une ébauche du futur système.

Avec cette approche, on est capable de définir plus explicitement et de manière plus cohérente les besoins des usagers, de détecter les fonctions manquantes et identifier et améliorer les fonctions complexes, comme elle permet de démontrer la faisabilité et l'utilité de l'application.

Le prototype est ensuite abandonné et le système réel est construit.

**Remarque 2.4.** Jeter le prototype est dû aux imperfections propres à tout prototype car la rapidité de développement est souvent réalisée au détriment de la maintenabilité du logiciel (ce qui est intolérable pour les gros systèmes logiciels)

Faut-il développer un prototype d'un futur système ou bien développer ce dernier et ensuite procéder aux modifications voulues ? Cela dépend en grande partie aux coûts inhérents à chaque approche.

### 2.2.3.2 Prototypes (non jetables)

Un prototype répond à des objectifs différents : on cherche à construire un système éventuellement très incomplet, mais dans son dimensionnement réel de façon à pouvoir faire des essais en vraie grandeur. On détermine d'abord un ensemble minimum de fonctions elles-mêmes incomplètes de façon à réaliser un premier incrément du logiciel dont on se sert pour analyser le comportement du logiciel. L'utilisateur fournit en retour des informations au concepteur qui modifie le prototype. Ce processus d'évolution de prototypes continue jusqu'à ce que l'utilisateur soit satisfait du système livré.

### 2.2.4 Modèle par incréments

Dans les modèles spirale, en V ou en cascade, un logiciel est décomposé en composants développés séparément et intégrés à la fin du processus.

Dans le modèle par incréments, seul un sous ensemble des composants d'un système est développé à la fois. Dans un premier temps un *logiciel noyau* est développé, puis successivement, les *incréments* sont développés et intégrés. Chaque incrément est développé selon l'un des modèles précédents. Notons qu'il peut aussi y avoir un chevauchement entre les différents processus de développement de ces incréments.

Les avantages de ce type de modèle sont :

- Chaque développement est moins complexe,
- Les intégrations sont progressives,
- Il peut y avoir des livraisons et des mises en services après chaque intégration d'incrément.
- Il permet de bien lisser (répartir) dans le temps l'effort de développement et les effectifs par rapport à ce qu'on rencontre dans les modèles en cascade et en V.

Néanmoins, ces modèles présentent des risques importants comme celui de voir remettre en cause le noyau ou les incréments précédents, un autre risque est d'être incapable d'intégrer un incrément.

Donc, son utilisation doit être faite avec précaution :

- La spécification du noyau, des incréments et de leurs interactions doit être faite globalement au début du projet.
- Les incréments doivent être aussi indépendants que possible aussi bien fonctionnellement qu'au niveau des calendriers de développement.

## 2.2.5 Modèle en Spirale

Pour corriger les travers de la démarche linéaire sont apparus des modèles dits en spirales, proposé par B. Boehm en 1988, où les risques, quels qu'ils soient, sont constamment traités au travers de bouclages successifs :

Chaque cycle de la spirale se déroule en quatre phases représentées par des quadrants :

1. Détermination des objectifs du cycle, des alternatives pour les atteindre, des contraintes, à partir des résultats des cycles précédents ou s'il n'y a pas, d'une analyse préliminaire des besoins,
2. Analyse des risques, évaluation des alternatives, éventuellement maquettage,
3. Développement et vérification de la solution retenue,
4. Revue des résultats et planification du cycle suivant.

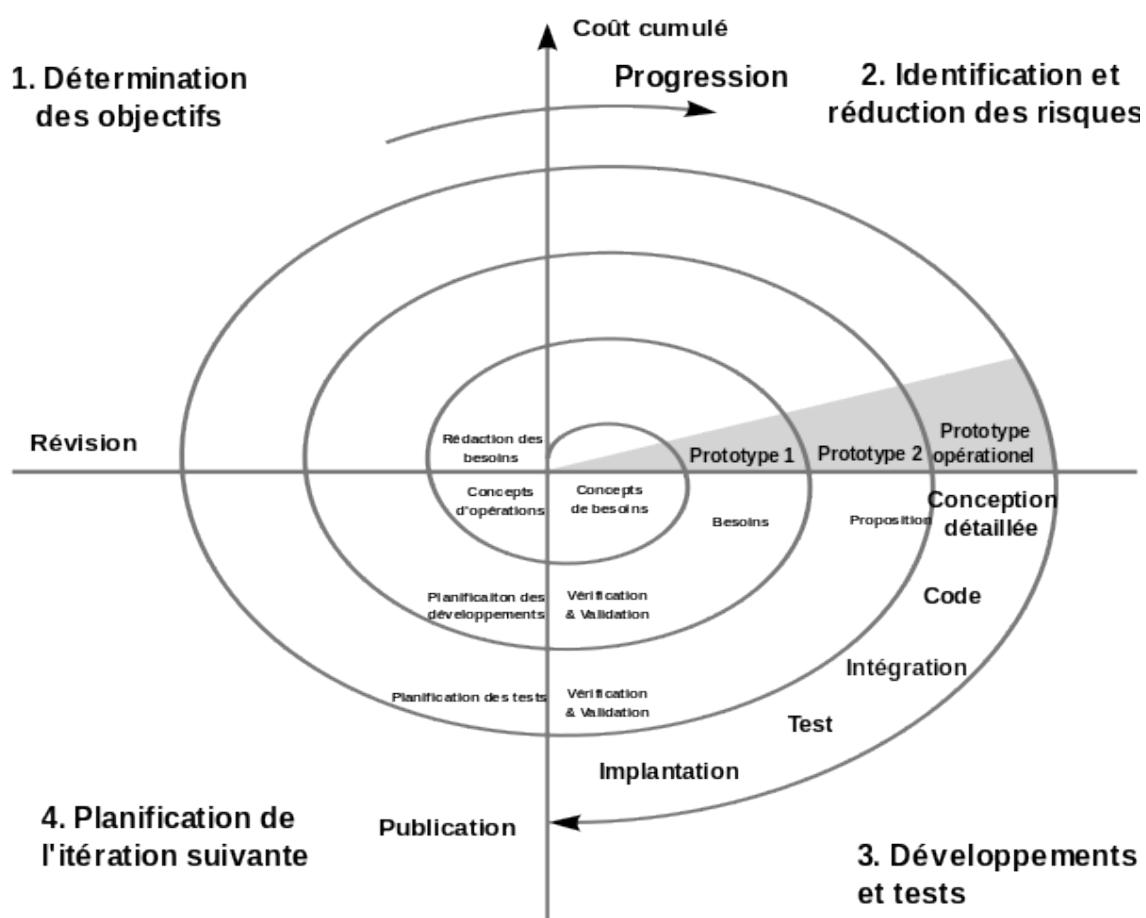


FIGURE 2.3: Le cycle de vie en spirale

Le quadrant 3 correspond à un développement ou à une portion de développement classique, et un des modèles précédents peut s'appliquer : son choix peut faire partie des alternatives à évaluer. L'originalité de ce "super" modèle est d'**encadrer le développement proprement dit par des phases consacrées à la détermination des objectifs et à l'analyse de risque.**

Un développement de ce modèle commence par une analyse préliminaire de besoins qui est affinée au cours des premiers cycles, en prenant en compte les contraintes et l'analyse des risques.

Le modèle utilise systématiquement des maquettes, qui durant ces cycles sont de nature exploratoire. Les troisièmes quadrants des cycles suivants correspondant à de la conception, les choix étant guidés par des maquettes expérimentales. Le dernier cycle se termine par la fin d'un processus classique. La mise en œuvre de modèle demande des compétences et un effort importants. En plus il s'agit d'un modèle moins expérimenté que les précédents et est moins documenté.

TABLEAU 2.2:

<b>Les risques majeurs du développement de logiciel</b>	<b>Solutions suggérées</b>
Défaillance du personnel	Embauche de personnel de haut niveau, adéquation entre profil et fonction, esprit d'équipe, formation mutuelle ...
Calendrier et budget irréalistes	Estimation détaillée des coûts et délais, développement incrémental, réutilisation ...
Développement de fonctions mal appropriées	Analyse de l'organisation, analyse de la mission, formulation des concepts opérationnels, revues d'utilisateurs, ...
Développement d'interfaces utilisateurs mal appropriées	Maquettage, scénarios et revues d'utilisateurs, analyse des tâches
Volatilité des besoins	Seuil élevé de modification, masquage d'information, développement incrémental où les derniers incrémentations sont les plus changeantes.
Problèmes de performances	Simulations, modélisations, essais et mesures, maquettage.
Exigences démesurées par rapport à la technologie	Analyses techniques de faisabilité, maquettage.

## 2.2.6 Modèles hybrides

Il ne fait aucun doute que l'adoption d'une approche évolutive du développement de certaines classes de systèmes peut conduire à des produits qui répondent mieux aux besoins des utilisateurs. C'est le cas, en particulier, des nouvelles applications qui n'ont pas d'équivalent non automatique. Dans de tels cas, la formulation des besoins est en effet très difficile.

Cependant, pour les systèmes basés sur une procédure manuelle bien connue, il est moins évident que l'approche évolutive serait d'un bon rapport coût - performance. Boehm (1984) semble indiquer que le contrôle et l'intégration du changement, en particulier dans le cas des grands systèmes, sont souvent plus difficiles lorsqu'un modèle évolutif est utilisé. Il en conclut que la meilleure solution globale peut être composée de plusieurs approches, l'analyse du risque encouru déterminant l'approche adoptée pour chaque sous-système particulier :

- Utilisation du prototypage pour les spécifications à hauts risques
- Modèle classique pour les parties bien connues.

# UML

## 3.1 Modélisation

### 3.1.1 Qu'est ce qu'un modèle ?

Un modèle est une **description abstraite** d'un système, destinée à en préciser certaines caractéristiques.

- Il permet de **réduire la complexité** d'un phénomène en éliminant les détails qui n'influencent pas son comportement de manière significative.
- Il permet de **simuler** le système étudié. Un modèle représente le système étudié et reproduit ses comportements.
- Il reflète ce que le concepteur croit important pour la **compréhension** et la **prédiction** du phénomène modélisé, les limites du phénomène modélisé dépendent des objectifs du modèle.

Principe : "Un beau dessin vaut mieux qu'un long discours". Seulement s'il est compris par tous de la même manière.

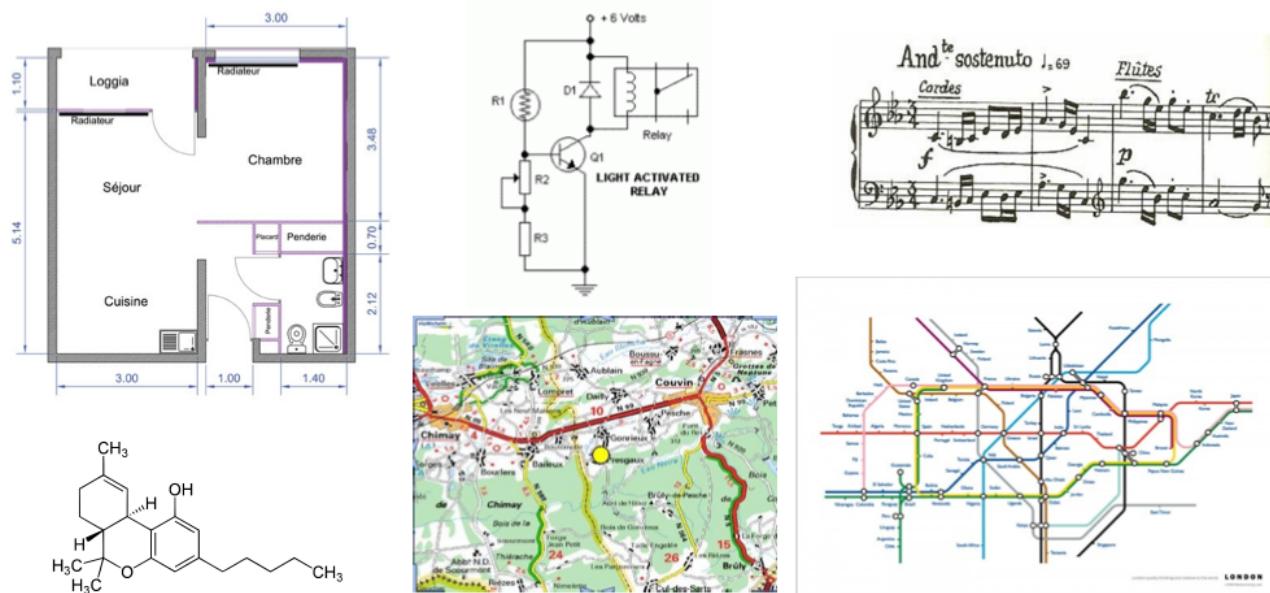


FIGURE 3.1: Modélisation

### Quelques exemples de modèles :

- **Modèle météorologique** : à partir de données d'observation (satellite ...), permet de prévoir les conditions climatiques pour les jours à venir.
- **Modèle économique** : peut par exemple permettre de simuler l'évolution de cours boursiers en fonction d'hypothèses macro-économiques (évolution du chômage, taux de croissance, ...).
- **Modèle démographique** : définit la composition d'un échantillon de population et son comportement, dans le but de fiabiliser des études statistiques, d'augmenter l'impact de démarches commerciales, etc.

### 3.1.2 Rôles d'un modèle en Génie Logiciel

Un modèle est une simplification de la réalité qui permet de mieux comprendre le système à développer. Il permet :

- De **visualiser** le système comme il est ou comme il devrait l'être.
- De **valider** le modèle vis à vis des clients
- De **spécifier** les structures de données et le comportement du système.
- De fournir un **guide** pour la construction du système.
- De **documenter** le système et les décisions prises

En pratique, un modèle est représenté par un ensemble de documents (diagrammes, textes, ?).

Il existe un standard industriel pour l'écriture de différents types de modèles : UML (Unified Modeling Language).

### 3.1.3 Langages de modélisation

Un langage de modélisation doit définir :

- La **sémantique** des concepts ;
- Une **notation** pour la représentation de concepts ;
- Des **règles** de construction et d'utilisation des concepts.

*Des langages à différents niveaux de formalisation*

- **Langues naturelles** : qui évoluent hors du contrôle d'une théorie. Ex : Français, Anglais, ...
- **Langages formels** (Z,B,VDM) : le plus souvent mathématiques, au grand pouvoir d'expression et permettant des preuves formelles sur les spécifications ;
- **Langages semi-formels** (MERISE, UML, ...) : le plus souvent graphiques, au pouvoir d'expression moindre mais plus faciles d'emploi.

*L'industrie du logiciel dispose de nombreux langages de modélisation :*

- Adaptés aux systèmes **procéduraux** (MERISE...);
- Adaptés aux systèmes temps réel (ROOM, SADT...);
- Adaptés aux systèmes à **objets** (OMT, Booch, UML, ...).

### 3.1.4 Histoire des modélisations par objets

Les méthodes utilisées dans les années 1980 pour organiser la programmation impérative (notamment Merise) étaient fondées sur la modélisation **séparée** des données et des traitements.

Lorsque **la programmation par objets prend de l'importance au début des années 1990**, la nécessité d'une méthode qui lui soit adaptée devient évidente. Plus de cinquante méthodes apparaissent entre 1990 et 1995 (Booch, Classe-Relation, Fusion, HOOD, OMT, OOA, OOD, OOM, OOSE, etc.) mais aucune ne parvient à s'imposer. En 1994, le consensus se fait autour de trois méthodes : **OMT**, **Booch** et **OOSE** dont la fusion a donné naissance au langage UML, sans négliger les apports d'autres techniques de modélisation..

## 3.2 UML

UML (*Unified Modeling Language* en anglais, soit **langage de modélisation objet unifié**).

La notation UML est une fusion des notations de Booch, OMT, OOSE et d'autres notations.

- La méthode OMT (*Object Modeling Technique -Technique de Modélisation Objet*) inventée par James Rumbaugh fournit une représentation graphique des aspects statique, dynamique et fonctionnel d'un système ;
- La méthode BOOCH inventée par Grady Booch, définie pour le Department of Defense, introduit le concept de package (élément d'organisation des modèles) ;
- La méthode OOSE (*Object Oriented Software Engineering*) inventée par Ivar Jacobson fonde l'analyse sur la description des besoins des utilisateurs (cas d'utilisation, ou "use cases").

UML est le résultat d'un large consensus d'experts reconnus. Par la suite, de nombreux acteurs industriels ont adopté UML et participent à son développement. Microsoft, Oracle, DEC, HP, Rational, Unisys etc. s'associent alors à l'effort et proposent UML 1.0 à l'OMG (*Object Management Group*) qui l'accepte en novembre 1997 dans sa version 1.1. La version d'UML en cours en 2011 est UML 2.4.1 qui s'impose plus que jamais en tant que langage de modélisation standardisé pour la modélisation des logiciels.

### Principales étapes de la définition d'UML :

- **1991** : première édition de Modélisation et conception orientées objet basée sur OMT, Object Modeling Technique, issue de la R&D de General Electric.
- **1994** : James Rumbaugh rejoint Rational et travaille avec Grady Booch à la fusion des notations OMT et Booch.
- **1995** : Ivar Jacobson rejoint Rational et intègre Objectory au travail d'unification.
- **1997** : l'OMG (Object Management Group) accepte UML, proposé par Rational, comme standard de modélisation objet.
- **2001** : révision par l'OMG d'UML 1.
- **2004** : adoption d'UML 2.0
- **2008** : UML 2.2
- **Aout 2011** : UML 2.4.1

Les concepteurs de la notation ont recherché avant tout la **simplicité**; UML est **intuitif**, **homogène** et **cohérent**. Les symboles embrouillés, redondants ou superflus ont été éliminés en faveur d'un meilleur rendu visuel.

#### 3.2.1 Objectifs d'UML

- Proposer un **langage visuel** de modélisation
- Utilisable par toutes les méthodes
- Adapté à toutes les phases du développement
- Compatible avec toutes les techniques de réalisation
- Proposer des mécanismes d'extension et de spécialisation pour pouvoir étendre les concepts de base
- Être indépendant des langages particuliers de programmation
- Proposer une base formelle pour comprendre le langage de modélisation
- Encourager l'application des outils Orientés Objets (OO)
- Supporter les concepts de développement de haut-niveau tels que les frameworks, les patterns et les composants
- Intégrer les résultats de la pratique

#### 3.2.2 Les diagrammes d'UML

Un diagramme donne à l'utilisateur un moyen de visualiser et de manipuler des éléments de modélisation. UML définit des diagrammes **structurels** et **comportementaux** pour représenter respectivement des vues **statiques** et **dynamiques** d'un système.

UML 2.0<sup>1</sup> s'articule autour de 13<sup>2</sup> types de diagrammes, chacun d'eux étant dédié à la représentation des concepts particuliers d'un système logiciel.

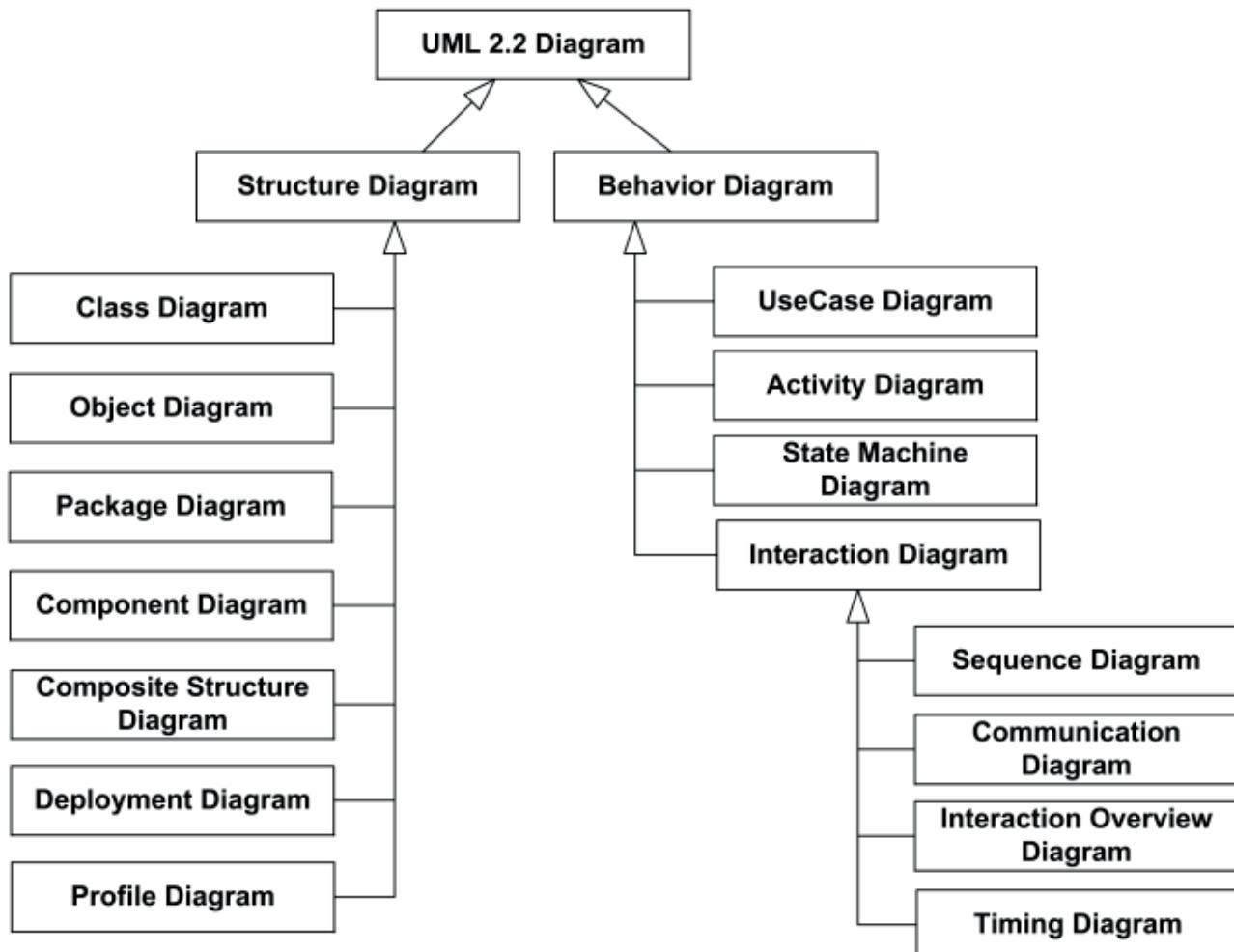


FIGURE 3.2: La hiérarchie des diagrammes UML 2.0 sous forme d'un diagramme de classes.

#### – DIAGRAMMES COMPORTEMENTAUX :

- **Diagramme de cas d'utilisation** Le diagramme de cas d'utilisation est utilisé dans l'activité de spécification des besoins. Il montre les interactions fonctionnelles entre les acteurs et le système à l'étude. Il représente les fonctions du système du point de vue des utilisateurs.
- **Diagramme d'états** Le diagramme d'états représente le cycle de vie commun aux objets d'une même classe. Ce diagramme complète la connaissance des classes en analyse et en conception en montrant les différents états et transitions possibles des objets d'une classe à l'exécution.

1. UML v2.0 date de 2005. Il s'agit d'une version majeure apportant des innovations radicales et étendant largement le champ d'application d'UML.

2. Un 14ème diagramme : **Diagramme de profils** a été introduit dans la version UML 2.2

- **Diagramme d'activité** Le diagramme d'activité représente les règles d'enchaînement des actions et décisions au sein d'une activité ou le comportement d'un cas d'utilisation, ou un processus métier.
- **DIAGRAMMES D'INTERACTIONS :**
  - **Diagrammes de séquence** Le diagramme de séquence représente la succession chronologique des opérations réalisées par un acteur. Il indique les objets que l'acteur va manipuler et les opérations qui font passer d'un objet à l'autre. Les diagrammes de séquence servent d'abord à développer en analyse les scénarios d'utilisation du système. Plus tard les diagrammes de séquence et de communication permettent de concevoir les méthodes des classes.
  - **Diagrammes de communication** Un diagramme de communication est un graphe dont les nœuds sont des objets et les arcs (numérotés selon la chronologie) les échanges entre objets. En fait, diagramme de séquence et diagramme de communication sont deux vues différentes mais logiquement équivalentes (on peut construire l'une à partir de l'autre) d'une même chronologie. Ce sont des diagrammes d'interaction.
  - **Diagramme global d'interaction (*interaction overview*)** Il fusionne les diagrammes d'activité et de séquence pour combiner des fragments d'interaction avec des décisions et des flots.
  - **Diagramme de temps** Il fusionne les diagrammes d'états et de séquence pour montrer l'évolution de l'état d'un objet au cours du temps.
- **DIAGRAMMES STRUCTURELS :**
  - **Diagramme de classes** Le diagramme de classes est le point central dans un développement orienté objet. En analyse, il a pour objet de décrire la structure statique des entités manipulées par les utilisateurs en termes classes, associations, interfaces, attributs, opérations, généralisations, etc. En conception, le diagramme de classes représente la structure d'un code orienté objet.
  - **Diagramme d'objets** Le diagramme d'objets est un instantané, une photo d'un sous-ensemble des objets d'un système à un certain moment du temps. C'est probablement le diagramme le moins utilisé d'UML.
  - **Diagramme de composants** Le diagramme de composants montre les unités logicielles à partir desquelles on a construit le système informatique, ainsi que leurs dépendances (avec leurs interfaces fournies et requises.).
  - **Diagramme de déploiement** Le diagramme de déploiement montre le déploiement physique des artefacts (éléments concrets tels que fichiers, exécutable, etc.) sur les ressources matérielles.

- **Diagramme de packages** Le diagramme de packages montre l'organisation logique du modèle et les relations entre packages. Il permet de structurer les classes d'analyse et de conception, mais aussi les cas d'utilisation.
- **Diagramme de structure composite** Comme le diagramme des packages, il montre les dépendances des éléments, mais au niveau de l'exécution. Il montre l'organisation interne d'un élément statique complexe sous forme d'un assemblage de parties, de connecteurs et de ports.
- **Diagramme de profils** Depuis UML 2.2, permet de spécialiser, de personnaliser pour un domaine particulier un meta-modèle de référence d'UML.

Une façon de mettre en œuvre UML est de considérer différentes vues qui peuvent se superposer pour collaborer à la définition du système :

- **Vue des cas d'utilisation** : C'est la description du modèle vu par les acteurs du système. Elle correspond aux besoins attendus par chaque acteur (c'est le **QUOI** et le **QUI**).
- **Vue logique** : c'est la définition du système vu de l'intérieur. Elle explique comment peuvent être satisfaits les besoins des acteurs (c'est le **COMMENT**).
- **Vue d'implantation** : cette vue définit les dépendances entre les modules.
- **Vue des processus** : c'est la vue temporelle et technique, qui met en oeuvre les notions de tâches concurrentes, stimuli, contrôle, synchronisation, etc.
- **Vue de déploiement** : cette vue décrit la position géographique et l'architecture physique de chaque élément du système (c'est le **OÙ**).

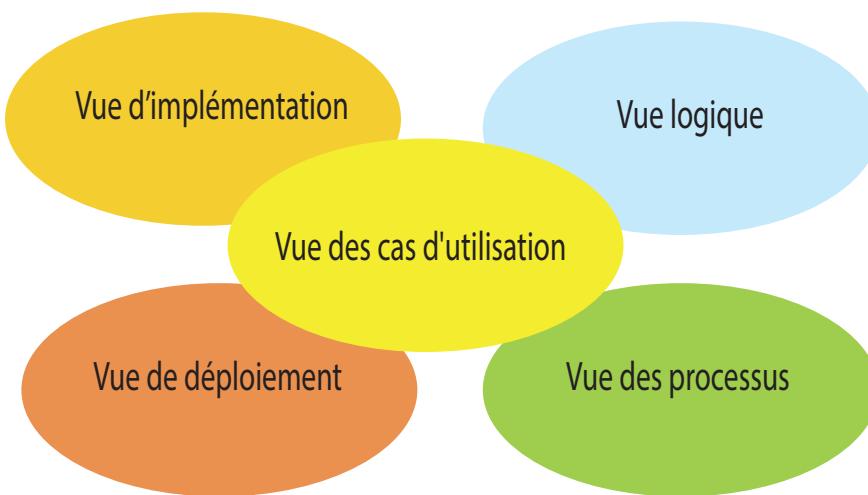


FIGURE 3.3: Vues d'UML.

### Exemple d'utilisation des diagrammes

- Spécification

- **Diagrammes de cas d'utilisation** : besoins des utilisateurs
- **Diagrammes de séquence** : scénarios d'interactions entre les utilisateurs et le logiciel, vu de l'extérieur
- **Diagrammes d'activité** : enchaînement d'actions représentant un comportement du logiciel
- **Conception**
  - **Diagrammes de classes** : structure interne du logiciel
  - **Diagrammes d'objet** : état interne du logiciel à un instant donné
  - **Diagrammes états-transitions** : évolution de l'état d'un objet
  - **Diagrammes de séquence** : scénarios d'interactions avec les utilisateurs ou au sein du logiciel
  - **Diagrammes de composants** : composants physiques du logiciel
  - **Diagrammes de déploiement** : organisation matérielle du logiciel

## PRINCIPES GÉNÉRAUX D'ANALYSE ET SPÉCIFICATION DES BESOINS

---

### 4.1 Analyse & Définition des Besoins

Les problèmes qui sont soumis aux ingénieurs logiciels sont souvent d'une complexité extrême. Comprendre la nature exacte d'un problème s'avère très complexe particulièrement s'il est nouveau et qu'aucun système non automatique n'existe pour servir de modèle au logiciel à développer.

Il est important de préciser le niveau de détail des besoins exprimés par l'utilisateur. L'information concernant ce problème doit être assemblée et analysée pour donner une définition compréhensible du problème qui servira pour concevoir et réaliser le logiciel.

Le processus visant à établir **quelles fonctionnalités le système doit fournir** et **les contraintes auxquelles il sera soumis**, s'appelle l'analyse et la définition des besoins qui est la première grande étape du cycle de vie du logiciel.

**Remarque 4.1.** *Le fait de présenter le besoin d'un logiciel pour gérer la comptabilité d'une entreprise, ne suffit pas à l'ingénieur logiciel de concevoir ce système. Il faudra rassembler et analyser toutes les informations concernant le problème et une définition compréhensible devra être produite et qui servira par la suite à concevoir et à réaliser le logiciel.*

**Remarque 4.2.** *Il faut faire la distinction entre un **but** et un **besoin**. Ce dernier peut être testé tandis que le but est une caractéristique plus générale que devrait posséder le système. Exemple : un but pourrait être que le système soit agréable à utiliser (critère subjectif). Par contre un besoin associé à ce but pourrait être que toutes les commandes utilisateur soient activées par menu.*

**Remarque 4.3.** *La tâche de définition des besoins peut parfois être confiée à un analyste dans le cas où un système manuel doit être automatisé mais souvent cela est fait en concertation avec l'ingénieur logiciel. Par contre, pour les systèmes complexes, souvent c'est l'ingénieur logiciel qui se charge de cette tâche.*

Les résultats de cette 1<sup>ère</sup> phase sont regroupés dans un document de définition des besoins appelé **le cahier des charges** du logiciel qui donc, doit décrire le système que l'on veut bâtir, et ce de manière aussi **précise, complète et consistante** que possible. Toute information non pertinente, superflue ou incomplète doit être éliminée en faisant des analyses critiques.

## 4.2 Cahier des Charges

- C'est un document qui contient des indications concernant les phases suivantes du cycle de vie.
- C'est un ensemble de propriétés ou de contraintes, décrites de façon précise, qu'un système logiciel doit satisfaire.
- Ce n'est pas un document de conception ; il doit décrire ce que le système doit faire sans spécifier comment il doit le faire.
- Les besoins exprimés dans ce document doivent constituer un ensemble complet et cohérent (aucun besoin ne doit être en conflit avec un autre déjà exprimé).

En pratique, ces propriétés sont extrêmement difficiles à vérifier, particulièrement si les besoins sont exprimés dans un langage naturel. Cependant, des notations plus formelles existent qui permettent dans une certaine mesure, de faire des vérifications de cohérence automatiques.

**Remarque 4.4.** *Cette vérification n'est pas facile ; par exemple le cahier de charges d'un système de défense à base de missiles contenait plus de 8000 besoins distincts couvrant 2500 pages.*

Les critères que devrait vérifier tout cahier de charges (selon Heninger, 1980), sont les suivants :

1. il doit spécifier uniquement le comportement externe du système.
2. il doit spécifier les contraintes de réalisation.
3. il doit être facile à mettre à jour.
4. il doit servir d'outil de référence aux programmeurs de maintenance du système.
5. il doit contenir des indications concernant les étapes ultérieures du cycle de vie du système.
6. il doit spécifier les réponses acceptables aux événements non désirables.

## 4.3 Structure d'un Cahier des Charges

Pour pouvoir satisfaire les critères cités précédemment, un cahier des charges doit être structuré comme suit :

1. **Introduction** : décrit la raison d'être d'un tel système et place celui-ci dans son contexte, en donnant brièvement ses fonctions et en présentant une justification de son organisation. L'introduction doit également préciser la structure du document et décrire les notations utilisées.
2. **Matériel** : si le système est réalisé sur un matériel spécial, cet équipement et ses interfaces doivent être décrits. Si un matériel standard est utilisé, les configurations minimale et optimale sous lesquelles le système peut s'exécuter, doivent être définies.

3. **Modèle Conceptuel** : Ce modèle doit définir une vue à très haut niveau des fonctions majeures du système et de leurs relations. Ce modèle est bien décrit par une notation graphique (exemple : modèle de flot de données).
4. **Besoins fonctionnels** : les besoins fonctionnels du système (les services fournis à l'utilisateur) doivent être décrits. Suivant la nature de ces besoins, la notation utilisée peut être un langage naturel, un langage semi-formel, un langage formel ou bien un mélange de ces notations.
5. **Les besoins en matière de base de données** : l'organisation logique des données manipulées par le système et leurs relations doivent être décrites dans cette section.
6. **Besoins non fonctionnels** : les contraintes sous lesquelles le logiciel doit opérer, doivent être exprimées et reliées aux besoins fonctionnels. Exemple : contraintes de temps, espace mémoire, représentation des données, langage de programmation.
7. **Information destinée à la maintenance** : cette section doit décrire les hypothèses fondamentales sur lesquelles le système est fondé ainsi que les changements prévus du fait de l'évolution du matériel, des besoins des utilisateurs, etc. Dans la mesure du possible, les fonctions et les contraintes particulièrement sujettes aux changements doivent être indiquées explicitement.
8. **Glossaire** : doit définir les termes techniques utilisés dans le document et ce de manière précise et sans faire aucune présupposition sur l'expérience ou la formation du lecteur.
9. **Index** : il est préférable de fournir plus d'une seule sorte d'index (alphabétique, par chapitre, des fonctions, ?).

**Remarque 4.5.** *Du fait que les besoins sont susceptibles de changer, il est essentiel que le document d'analyse des besoins soit organisé de façon qu'il puisse évoluer facilement. Faute de quoi, des changements pourront intervenir dans le système sans qu'ils soient enregistrés dans le cahier des charges. Cela provoquera un défaut de synchronisation entre le programme et sa documentation créant ainsi des problèmes immenses aux programmeurs de maintenance.*

## 4.4 Modèle Conceptuel du Système

Une fois l'analyse initiale des besoins réalisée, l'étape suivante consiste à produire un modèle conceptuel du système qui est une vue du système à très haut niveau, dans laquelle les fonctions majeures voulues par l'utilisateur sont identifiées et leurs relations documentées.

Pour les systèmes très simples, ce modèle peut n'exister que dans l'esprit de l'ingénieur responsable de la définition des besoins car celui-ci comprend le système dans son ensemble et

connaît les fonctions qui doivent être fournies ainsi que les contraintes imposées à l'exécution de ces fonctions.

Par contre, pour les systèmes complexes, il est nécessaire de définir très tôt un modèle du système qui soit précis et explicite et d'utiliser ce modèle pour en comprendre le fonctionnement.

Les notations les plus efficaces pour la description du modèle conceptuel du système, sont graphiques. En effet, dessins et diagrammes sont en général compréhensibles par des utilisateurs mêmes non spécialistes en génie logiciel.

**Exemple :** Considérons un système bureautique composé d'un courrier électronique, d'un tableur et de services de préparation de documents et de recherche d'informations.

**1<sup>ère</sup> étape Identifier les fonctionnalités principales et leurs relations.** Les fonctionnalités principales (courrier électronique, tableur, préparation de documents, recherche d'informations) peuvent être identifiées, ainsi que les fonctions de support système (communications et bases de données). Le modèle conceptuel de haut niveau est illustré par le graphe de la figure 4.1.

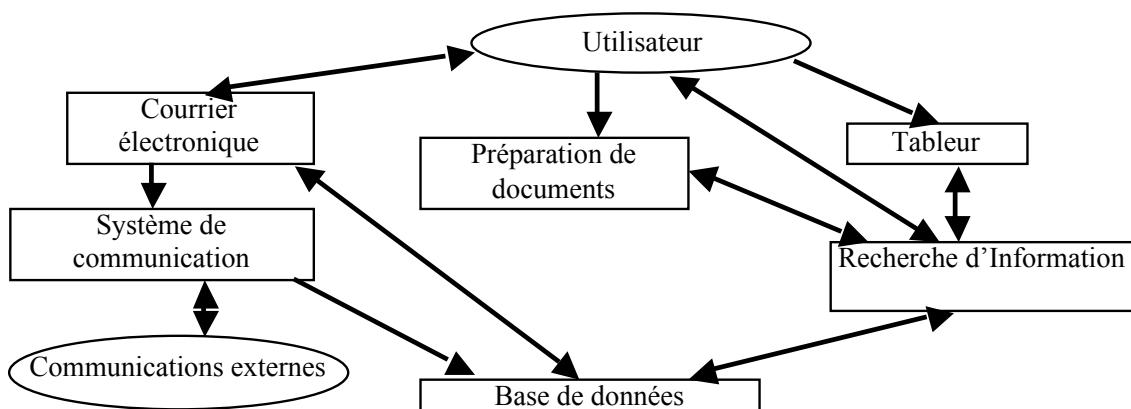


FIGURE 4.1: Modèle conceptuel de haut niveau

**2<sup>ème</sup> étape** L'étape suivante du processus de modélisation consiste à prendre chaque fonction principale et en définir un modèle conceptuel. Exemple : Fonction Courrier (voir figure 4.2).

Un modèle similaire peut être dérivé pour chaque fonction principale ; ces modèles peuvent être affinés à plusieurs niveaux de détails si nécessaire. Le modèle conceptuel fournit alors un cadre sur lequel la définition des besoins plus détaillée et plus complète peut s'appuyer mais à condition de ne pas influer trop directement sur la phase de conception.

**Remarque 4.6.** Si le système est complètement nouveau, le modèle conceptuel peut être incomplet car on ne connaît pas exactement les fonctions souhaitées. Les besoins logiciels dérivés

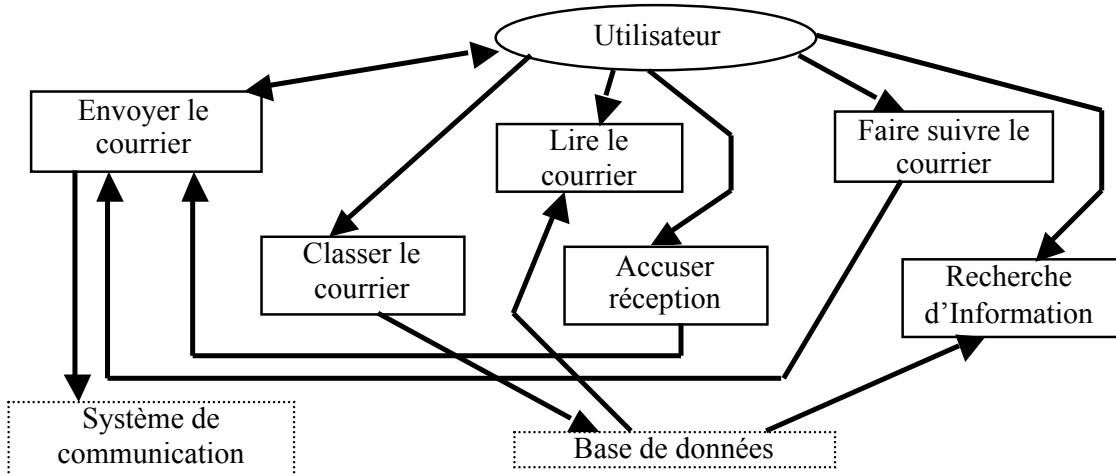


FIGURE 4.2: Fonction Courrier

*d'un tel modèle sont inévitablement imprécis. Les besoins plus précis ne peuvent être formulés qu'après avoir fait des expériences de conception de tels systèmes et dont les résultats permettent au retour d'établir un modèle conceptuel plus complet.*

## 4.5 Besoins fonctionnels

Habituellement, la plus grande partie du document de définition des besoins logiciels est consacrée à la définition des besoins fonctionnels.

Les besoins fonctionnels définissent les services attendus par l'utilisateur. En général, l'utilisateur ne se soucie pas de savoir comment ces services seront réalisés. L'ingénieur logiciel doit donc éviter l'introduction de concepts de réalisation dans cette section du document de définition des besoins.

En principe, les besoins fonctionnels doivent constituer un ensemble complet et cohérent. La complétude signifie que tous les services requis par l'utilisateur doivent être spécifiés, et la cohérence que les définitions des besoins ne doivent pas être contradictoires.

Dans les systèmes grands et complexes, il est impossible d'atteindre les propriétés de complétude et de cohérence dans la version initiale du document de définition des besoins.

Au fur et à mesure que les problèmes sont découverts pendant les revues ou plus tard dans le cycle de vie, le document de définition des besoins, doit être modifié.

Il y a trois façons d'exprimer les besoins fonctionnels :

1. Dans un langage naturel.
2. Dans un langage structuré comportant quelques règles sans définition précise de sa syntaxe et sa sémantique.

3. Dans un langage de spécification formel ayant une définition rigoureuse de sa syntaxe et sa sémantique.

Le langage naturel est le plus utilisé car il est le plus expressif et le plus compréhensible par les utilisateurs et les développeurs à la fois. En pratique, l'utilisation des langages formels est principalement réservée à la spécification de logiciels critiques, ou restreinte aux parties critiques du système.

#### **4.5.1 Définition des besoins en langage naturel**

- Présenter les besoins dans des paragraphes constitués de texte en langage naturel.
- Les besoins sont imprécis, descriptifs et de très haut niveau (ce qui est le cas des systèmes novateurs).
- Les besoins fonctionnels et non fonctionnels et les buts ne sont pas clairement distingués.
- Dans un paragraphe donné, on peut regrouper l'expression de plusieurs besoins distincts dans une seule phrase. Ceci rend les vérifications de complétude et de cohérence très difficiles.

**Exemple :** La BD doit permettre l'accès aux objets d'une même version par l'utilisation d'un nom incomplet (simplement en connaissant le contexte racine de la version).

**Problèmes :** Bien que les langages naturels constituent probablement le seul moyen d'exprimer des besoins de haut niveau, des défauts sont constatés lors d'expression des besoins à l'aide de paragraphes non structurés :

- Ils reposent sur l'expérience linguistique partagée du lecteur et du rédacteur du document. Ce dernier suppose que les termes utilisés ont le même sens pour lui et pour le lecteur. Cela conduit à des ambiguïtés dangereuses.
- Ils ne permettent pas d'exprimer l'architecture fonctionnelle d'un système de façon claire et concise. L'architecture fonctionnelle (Schoman et Ross, 77) est une description des activités du système et des entités en interaction à l'intérieur du système.
- Ils sont trop souples (différencient des besoins proches, ...).
- Ils ne permettent pas de séparer efficacement les besoins.

#### **4.5.2 Définition des besoins en langage structuré**

Pour corriger les désavantages des langages naturels, un bon nombre de notations ont été développées pour exprimer les besoins de manière détaillée et précise.

Ces notations reposent sur le langage naturel pour définir les besoins de façon expressive. Cependant au lieu de l'utiliser de façon non structurée, ils imposent une certaine structure en

limitant les expressions du langage naturel qu'il est permis d'utiliser, et pour certains de ces langages, en les enrichissant d'une notation graphique.

**Exemple :** PSL/PSA, SADT, RSL, UML.

- SADT (Shoman et Ross, 1977) : Technique d'analyse structurée reposant essentiellement sur le graphisme. Elle est destinée non seulement à exprimer les besoins mais aussi à partitionner, structurer, exprimer et communiquer des idées.
- RSL (Bell, 1977) : est une méthode (de spécification des systèmes temps réel) dont les instructions sont traitables automatiquement. L'information dérivée de ces instructions est assemblée dans une base de données appelée le modèle sémantique abstrait du système (MSAS). Un ensemble d'outils automatiques traite l'information contenue dans MSAS pour générer des simulateurs, produire des rapports et vérifier la cohérence et la complétude des besoins exprimés.

#### **4.5.3 Langages de spécification**

Ce sont des langages qui reposent sur des bases théoriques (mathématiques) solides, leur syntaxes et sémantiques étant définies d'une manière formelle et rigoureuse. Leur avantage est de permettre une analyse et vérification des propriétés voulues de manière automatique. Par contre, leur nature formelle qui exige de solides connaissances théoriques pré-requises, rend leur adoption très difficile par les développeurs de logiciels. De plus, ces langages rencontrent des problèmes techniques (explosion combinatoire) qui ne sont pas entièrement maîtrisés.

### **4.6 Besoins non fonctionnels**

Un besoin non fonctionnel correspond à une restriction ou à une contrainte placée sur une des fonctions du système.

**Exemple :** contraintes sur le temps de réponse maximum, limitations sur l'espace mémoire nécessaire, restrictions sur la représentation des données.

Bien que les besoins fonctionnels et non fonctionnels soient tous deux susceptibles d'évoluer, les besoins non fonctionnels sont particulièrement affectés par les modifications de la technologie du matériel (surtout dans les projets de grands systèmes ayant une durée de développement et d'exploitation de plusieurs années).

Les changements matériels qui peuvent se produire lorsque le logiciel est en cours de développement, peuvent être anticipés. Les besoins non fonctionnels qui dépendent du matériel peuvent être spécifiés en supposant des performances qui ne seront effectives qu'en fin de projet. Ce type d'anticipation ne peut être fait pour des changements concernant la période d'explo-

tation. En conséquence, il est important que ces besoins liés au matériel soient spécifiés de telle façon qu'ils puissent être facilement modifiés.

Les besoins non fonctionnels peuvent rentrer en conflit et interagir les uns avec les autres et même avec des besoins fonctionnels.

**Exemple :** conflit entre vitesse d'exécution et occupation mémoire. Donc, il faut que ces conflits apparaissent clairement lors de la définition des besoins non fonctionnels afin de trouver les compromis possibles.

## 4.7 Validation des Besoins

Une fois que les besoins sont définis, il faut les valider comme suit :

- La **cohérence** des besoins doit être établie. Aucun besoin ne peut être en conflit avec un autre besoin.
- L'ensemble des besoins exprimés doit être **complet**. Il doit comprendre toutes les fonctions et contraintes souhaitées par l'utilisateur du système.
- Les besoins doivent être **réalistes** vis à vis de la technologie des matériels et logiciels. Il est possible d'anticiper quelque peu l'évolution du matériel mais pour ce qui est du logiciel, les développements sont beaucoup moins prévisibles.
- Les besoins des utilisateurs doivent être **valides**. Un utilisateur peut penser qu'un système doit fournir certains services, mais une analyse poussée peut faire apparaître la nécessité de fournir des fonctionnalités supplémentaires ou différentes.

**Remarque 4.7.** *Les erreurs dans cette étape seront propagées aux phases de conception et réalisation du système. Pour les corriger, il faut procéder à des modifications qui s'avèrent souvent très coûteuses. Boehm (1974) rapporte que pour quelques grands systèmes, il a été nécessaire de récrire jusqu'à 95% du code pour tenir compte des changements dans la définition des besoins.*

Une revue de définition des besoins constitue le moyen de validation le plus efficace. Pendant une revue, les besoins sont étudiés à la fois par les utilisateurs et par les développeurs afin de déceler toute anomalie ou contradiction. Ce travail est facilité s'il existe des outils d'analyse.

**Analyseur de mots-clés : (Boehm;1974)** Etant donné un ensemble de mots clés tels que "protection", "mémoire", "accès", ... le système extrait les paragraphes définissant des besoins contenant ces mots clés. Les paragraphes extraits sont alors comparés pour vérifier que les besoins ne sont pas en conflit.

**Remarque 4.8.** *D'autres outils automatiques permettent de vérifier qu'il n'existe pas de termes différents pour désigner le même objet, ou comparent les entrées et les sorties des fonctions.*

**Remarque 4.9.** Des outils appelés simulateurs permettent, à partir de descriptions formelles (mathématiques) d'un futur système, de simuler son exécution afin de vérifier certaines propriétés importantes.

**Problème :** L'étape de validation des besoins, nécessite une étroite coopération de la part des utilisateurs. Or, beaucoup d'entre eux ne comprennent pas clairement leurs besoins et ne peuvent pas comparer un besoin effectif à sa traduction fonctionnelle surtout dans le cas d'un nouveau système grand et complexe. Ils ne peuvent alors valider leurs besoins qu'en évaluant un logiciel existant.

**Prototypage** Ceci à conduit à l'adoption de l'approche évolutive du développement des systèmes en utilisant la technique du prototypage (jetable ou non). On présente à l'utilisateur un système incomplet à partir d'une spécification grossière et on procède à des expérimentations pour améliorer la spécification. Pour la suite, deux options se posent : le prototype (jetable) peut être abandonné pour reconstruire complètement le système réel à partir de zéro ou bien le prototype (non jetable) est lui même transformé progressivement en un produit final.

**Prototypage de l'interface utilisateur** Avec l'avènement des interfaces utilisateurs graphiques ( Macintosh d'Apple, Windows pour les PC, X-Windows pour les plate-formes Unix ), l'effort engagé dans la conception et la réalisation d'une interface utilisateur représente une partie significative des coûts de développement d'une application.

D'où l'approche "Conception centrée sur l'utilisateur" qui prône le développement d'un prototype et l'implication de l'utilisateur dans la phase de conception de l'interface.

## 4.8 Diagramme de cas d'utilisation

Avant de développer un système, il faut savoir précisément à **QUOI** il devra servir, c'est à dire à quels besoins il devra répondre.

Le diagramme *Use Case* (ou de Cas d'utilisation) répond à la question "**qu'est-ce que les utilisateurs attendent ?**", c'est-à-dire les relations entre les acteurs et les fonctionnalités du système d'information.

Le diagramme de Cas d'utilisation doit aussi permettre de répondre à la question **Qui fait quoi ?**

### 4.8.1 Cas d'utilisation

Un cas d'utilisation (*use case*) décrit la fonctionnalité du système logiciel qu'un acteur doit exécuter pour obtenir un certain résultat ou pour atteindre un objectif. Les cas d'utilisation

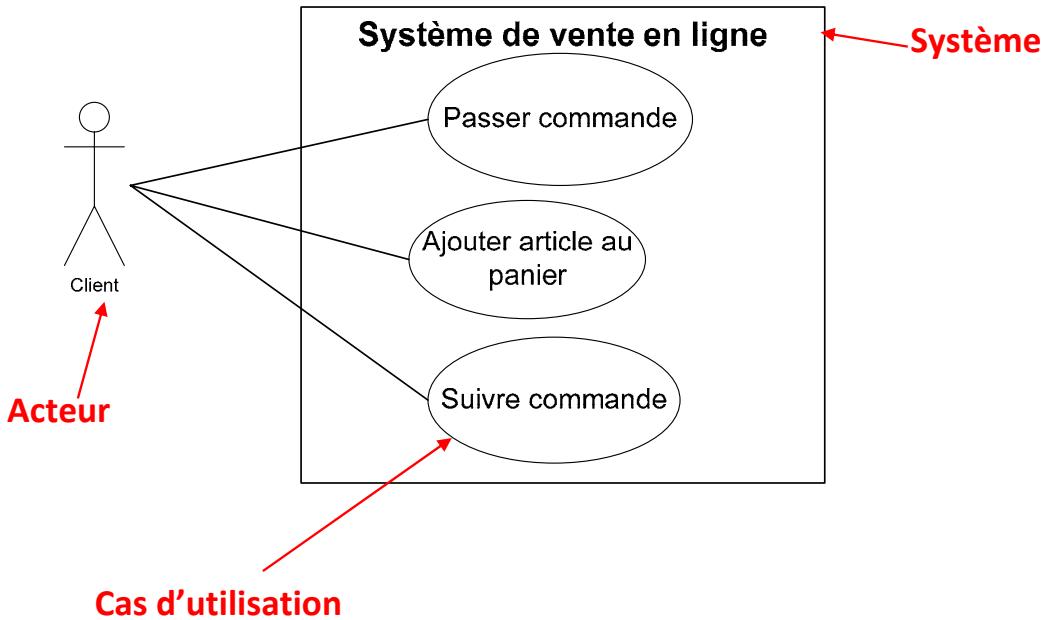


FIGURE 4.3: Exemple de diagramme de cas d'utilisation

doivent permettre de discuter avec l'utilisateur futur des fonctionnalités du système logiciel sans se perdre dans les détails.

Les cas d'utilisation sont généralement modélisés sous forme d'**ellipse**. Le nom peut figurer à l'intérieur de l'ellipse ou au-dessous. Les cas d'utilisation peuvent éventuellement être représentés par un rectangle doté d'un pictogramme d'ellipse.

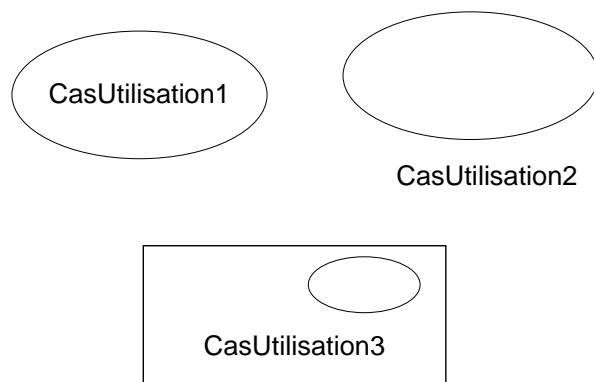


FIGURE 4.4: Notations graphiques possibles d'un cas d'utilisation

Une **ligne** entre un acteur et un cas d'utilisation signifie qu'une communication est établie. Elle est modélisée sous forme d'**association** en UML.

Le **système observé** (*subject*) est modélisé dans le diagramme de cas d'utilisation sous forme de grand rectangle comprenant tous les cas d'utilisation.

### 4.8.2 Acteur

Un acteur (*actor*) est un **rôle** joué par l'utilisateur du système logiciel. Les acteurs peuvent être des personnes physiques comme des systèmes automatisés. Ils se trouvent obligatoirement à l'**extérieur** du système.

Le diagramme de cas d'utilisation (*use case diagram*) donne, à un niveau d'abstraction supérieur, un bon aperçu du système logiciel et de son interface. Les acteurs sont souvent spécifiés sous forme de **personnages stylisés** (Figure 4.5). Ils peuvent également être représentés par un rectangle doté du stéréotype `actor` ou par un pictogramme (par exemple un symbole d'ordinateur) – Figure 4.6.



FIGURE 4.5: Acteur sous forme de **personnages stylisés**

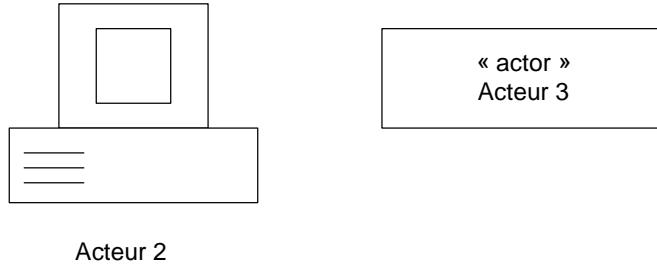


FIGURE 4.6: Acteurs

**Remarque 4.10.** *Un acteur correspond à un rôle, pas à une personne physique.*

- Une même personne physique peut être représentée par plusieurs acteurs si elle a plusieurs rôles.
- Si plusieurs personnes jouent le même rôle vis-à-vis du système, elles seront représentées par un seul acteur.

**EXEMPLE** Monsieur Mustafa, garagiste de son état, passe le plus clair de son temps dans le rôle du mécanicien, mais peut à l'occasion jouer le rôle de vendeur. Le Samedi, il joue le rôle du client et entretient sa voiture personnelle (Figure 4.7).

En plus des utilisateurs, les acteurs peuvent être :

- Des périphériques manipulés par le système (imprimantes...);

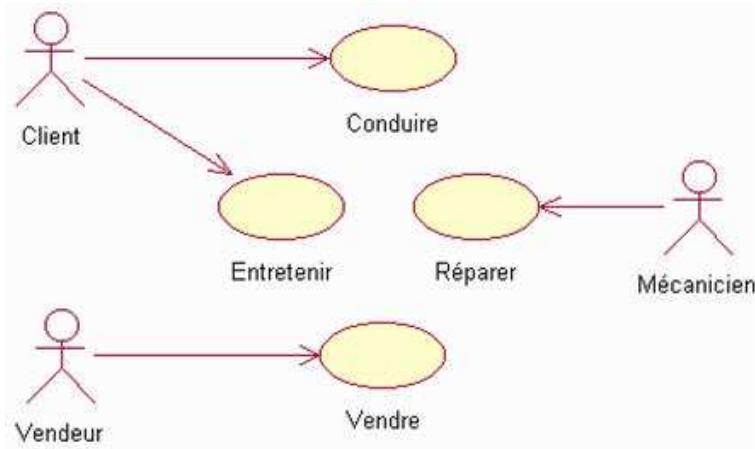


FIGURE 4.7: Exemple de diagramme de cas d'utilisation

- Des logiciels déjà disponibles à intégrer dans le projet ;
- Des systèmes informatiques externes au système mais qui interagissent avec lui, etc.

Pour faciliter la recherche des acteurs, on se fonde sur les frontières du système.

#### ACTEURS PRINCIPAUX ET SECONDAIRES

- L'acteur est dit **principal** pour un cas d'utilisation lorsque l'acteur est à l'initiative des échanges nécessaires pour réaliser le cas d'utilisation (C'est lui qui déclenche le cas d'utilisation).
- Les acteurs **secondaires** sont sollicités par le système alors que le plus souvent, les acteurs principaux ont l'initiative des interactions. Le plus souvent, les acteurs secondaires sont d'autres systèmes informatiques avec lesquels le système développé est inter-connecté

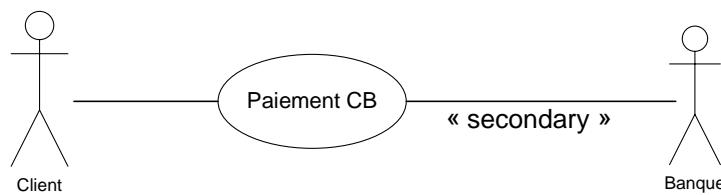


FIGURE 4.8: Acteurs principaux et secondaires

#### RELATION ACTEUR-ACTEUR

Une seule relation possible : la généralisation/spécialisation

### 4.8.3 Relations entre cas d'utilisation et acteurs (association)

Les acteurs impliqués dans un cas d'utilisation lui sont liés par une **association**. Un acteur peut utiliser plusieurs fois le même cas d'utilisation (dans la figure 4.10, \* indique le nombre d'instances possibles de l'association).

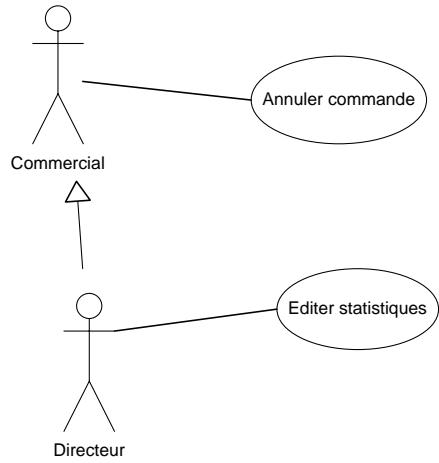


FIGURE 4.9: Relation de généralisation entre acteurs

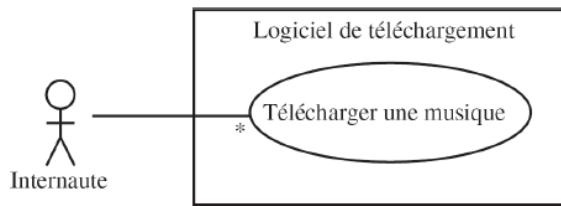


FIGURE 4.10: Relations entre cas d'utilisation et acteurs

**Multiplicité** : Nombre de fois où l'acteur peut déclencher le cas

- \* : une infinité de fois (pas représenté en général)
- [n..m] : entre n et m fois
- n : exactement n fois

#### 4.8.4 Relations entre cas d'utilisation

Afin d'optimiser la formalisation des besoins en ayant recours notamment à la réutilisation de cas d'utilisation, trois relations peuvent être décrites entre cas d'utilisation.

##### 4.8.4.1 Relation include

**Inclusion** : le cas d'utilisation de base en incorpore explicitement un autre, de façon **obligatoire**.

Un cas A inclut un cas B (**A "include" B**) :

- Le comportement décrit par le cas A inclut le comportement du cas B.
- Lorsque A est sollicité, B l'est obligatoirement, comme une partie de A.
- A implique B (B est nécessaire pour A)

- Le cas d'utilisation B n'est pas optionnel : il est **toujours** nécessaire pour la bonne exécution de A.

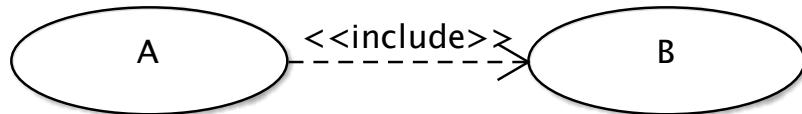


FIGURE 4.11: Exemple de la relation *include*

**Remarque 4.11.** Contrairement à la relation *extend*, l'exécution du cas d'utilisation B n'est dépendante d'aucune condition.

La relation *include* évite la description multiple du même comportement.

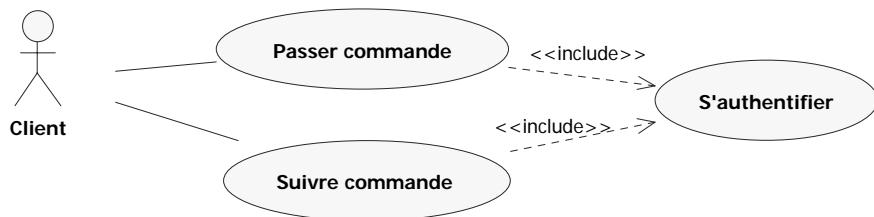


FIGURE 4.12: Exemple de la relation *include*

Quand un cas est trop complexe (faisant intervenir un trop grand nombre d'actions), on peut procéder à sa décomposition en cas plus simples.

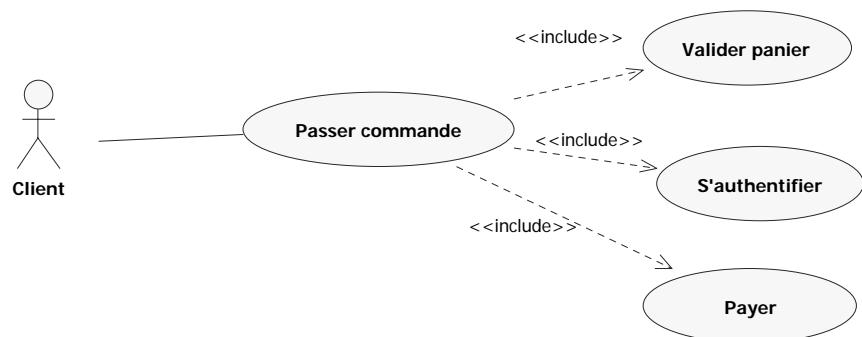


FIGURE 4.13: Exemple de la relation *include*

#### 4.8.4.2 Relation extend

**Extension :** On utilise principalement cette relation pour séparer le comportement **optionnel** (les variantes) du comportement obligatoire.

B étend A (**B "extend" A**) :

- Le comportement de A peut être étendu par le comportement de B
- Le cas d'utilisation A est complété par le cas d'utilisation B.
- Le cas d'utilisation A décrit la fonctionnalité de base, le cas d'utilisation B spécifie les extensions.
- Le cas d'utilisation A peut être exécuté seul ou avec les extensions.
- Exécuter A peut **éventuellement** entraîner l'exécution de B

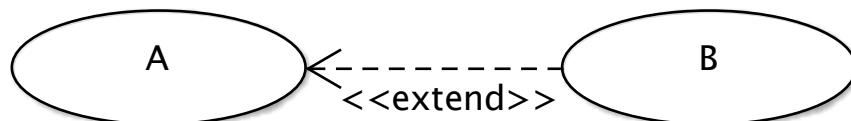


FIGURE 4.14: Exemple de la relation *include*

Les points d'extension (*extension point*) indiquent les emplacements auxquels le cas d'utilisation A de base est complété.

Pour qu'une extension soit ajoutée, il faut qu'une condition soit remplie. Cette condition peut au besoin être spécifiée sous forme de note ou de commentaire attaché à la relation *extend*.

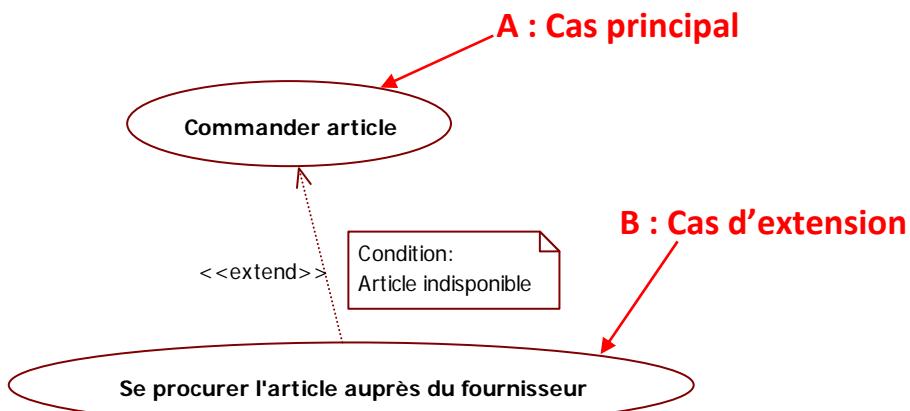


FIGURE 4.15: Exemple de la relation *extend*

#### 4.8.4.3 GÉNÉRALISATION

Il est en outre possible de modéliser une généralisation entre deux cas d'utilisation ou deux acteurs. Comme pour les classes, l'élément spécialisé hérite de toutes les propriétés de l'élément

général. Dans la figure 4.16, le cas A est une généralisation du cas B (B est une sorte de A).

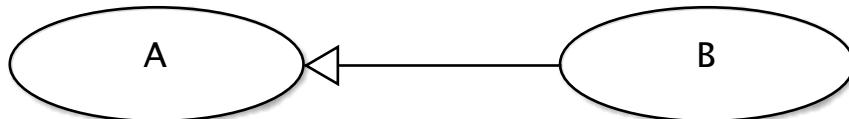


FIGURE 4.16: Exemple de la relation *include*

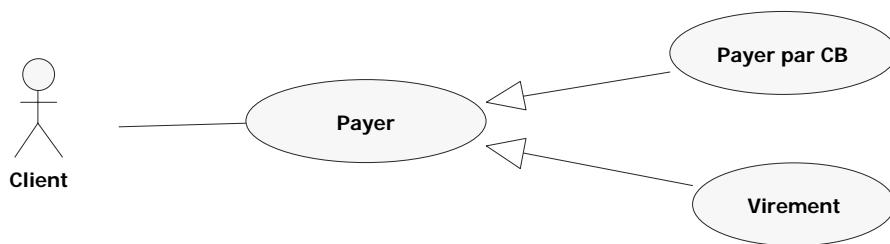


FIGURE 4.17: Exemple de généralisation entre deux cas d'utilisation et deux acteurs

#### 4.8.5 Recenser les cas d'utilisation

Il n'y a pas une manière mécanique et totalement objective de repérer les cas d'utilisation.

Il faut se placer du point de vue de chaque acteur et déterminer comment il se sert du système, dans quels cas il l'utilise, et à quelles fonctionnalités il doit avoir accès.

Il faut éviter les redondances et limiter le nombre de cas en se situant au bon niveau d'abstraction (par exemple, ne pas réduire un cas à une seule action).

Il ne faut pas faire apparaître les détails des cas d'utilisation, mais il faut rester au niveau des grandes fonctions du système.

#### 4.8.6 Identifier les scénarios

Lors de l'étape de détermination des besoins fonctionnels, un scénario représente une séquence d'interactions entre le système et ses acteurs. Le système est alors considéré comme une boîte noire.

Un scénario décrit une exécution particulière d'un cas d'utilisation du début à la fin. Il correspond à une sélection d'enchaînements du cas d'utilisation.

On peut distinguer plusieurs types de scénarios :

- **nominaux** : ils réalisent les postconditions du cas d'utilisation, d'une façon naturelle et fréquente ;
- **alternatifs** :
  - (*Scénario de succès*) ils remplissent les postconditions du cas d'utilisation, mais en empruntant des voies détournées ou rares ;
  - (*Scénario d'erreur*) : ne réalisent pas les postconditions du cas d'utilisation.

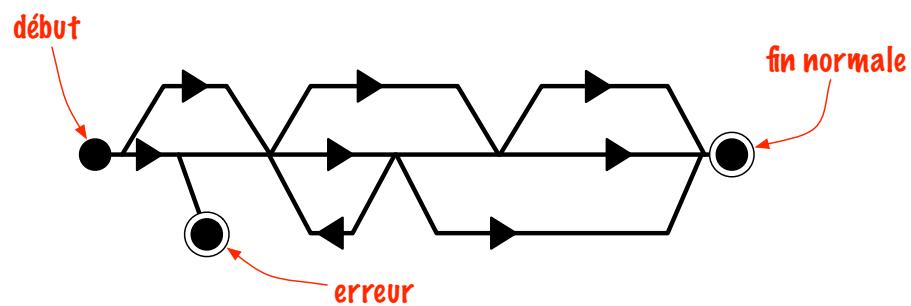


FIGURE 4.18: Représentation des variantes d'un cas d'utilisation

Chaque scénario est composé d'étapes qui peuvent être de trois sortes :

1. Un message d'un acteur vers le système
2. Une validation ou un changement d'état du système
3. Un message du système vers un acteur

Les étapes sont numérotées séquentiellement afin de pouvoir facilement indiquer par la suite les alternatives possibles.

Comme les alternatives doivent se brancher sur le scénario nominal, la convention de numérotation des étapes prend toute son importance. Ainsi à une étape  $X$ , une première alternative se notera  $XA$ . Elle identifiera d'abord la condition qui provoque l'alternative, puis la réponse du système.

#### NE CHERCHEZ PAS L'EXHAUSTIVITÉ DES SCÉNARIOS !

Un scénario correspond à l'exécution d'un ou de plusieurs enchaînements, joignant le début du cas d'utilisation à une fin normale ou non. Il est clair que la combinatoire des enchaînements fait exploser le nombre de scénarios potentiels ! Nous ne pourrons donc pas tous les décrire. Il faudra faire des choix, essayer de trouver le meilleur rapport « qualité/prix », c'est-à-dire l'ensemble minimal de scénarios permettant de couvrir toutes les actions/réactions du système. Cela revient à définir une famille de scénarios qui empruntent au moins une fois toutes les branches d'exécution du cas d'utilisation.

#### 4.8.7 Spécification du cas d'utilisation (*use case template*)

**Un nom du cas d'utilisation** – Le nom compte deux ou trois mots (qu'est ce qui est fait ?). Utiliser un verbe à l'infinitif.

**Une description** – Définition globale de l'objectif en cas d'exécution réussie du processus. Résumé permettant de comprendre l'intention principale du cas d'utilisation.

**Acteurs** – Rôles des personnes ou d'autres systèmes qui déclenchent le cas d'utilisation ou qui y participent (Acteur principal ou déclencheur et Acteur(s) secondaire(s)).

**Pré-condition** – État attendu avant l'exécution du cas d'utilisation.

**Post-condition (succès)** – État attendu après l'exécution réussie du cas d'utilisation, c'est-à-dire le résultat du cas d'utilisation.

**Le scénario nominal** – Celui qui satisfait les objectifs des acteurs par le chemin le plus direct de succès

**Alternatives** – Comprennent tous les autres scénarios, de succès (fin normale) ou d'échec (erreur)

**Rubriques optionnelles** – Très souvent, les exigences non fonctionnelle et les contraintes de conception se rapportent spécifiquement à un cas d'utilisation plutôt qu'au système dans sa totalité (ex. performance, sécurité, ergonomie, le nombre de personnes exécutant ce cas d'utilisation dans une journée type, ...)

### 4.9 Diagramme d'activité

Le diagramme d'activités permet de représenter la **dynamique** du système. C'est un graphe orienté qui décrit un enchaînement de traitements qui peut être soumis à des **branchements** ou à des **synchronisations**.

Les diagrammes d'activité offrent une manière graphique et non ambiguë pour modéliser les traitements. Un **traitement** peut être une **action** (une opération élémentaire) ou une **activité**, pouvant elle-même être décrite à un niveau plus fin.

Le diagramme d'activité est une variante des diagrammes d'états-transitions, ce diagramme met l'accent sur les activités, leurs relations et leurs impacts sur les objets.

Un diagramme d'activité est utilisé pour :

- **Spécifier une opération** (décrire la logique d'une opération). Le diagramme d'activité a une représentation proche de l'**organigramme**. Cette représentation le rend facilement intelligible et beaucoup plus accessible que le diagramme d'états-transitions. Il permet pour une classe donnée, d'exprimer graphiquement une activité (donc une méthode) en un

algorithme avec la possibilité d'une traduction directe dans un langage de programmation O.O

- Modéliser le **déroulement d'un cas d'utilisation**. Le diagramme d'activité est particulièrement adapté à la **description des cas d'utilisation**. Plus précisément, il vient illustrer et consolider la description textuelle des cas d'utilisation

#### 4.9.1 Éléments d'un diagramme d'activité

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

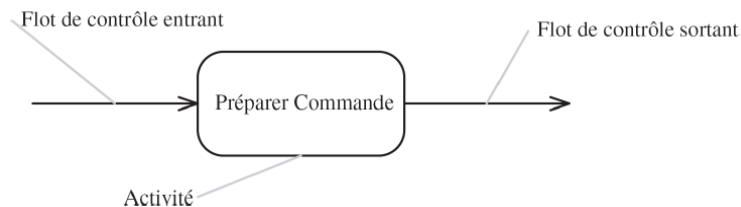


FIGURE 4.19: Activité

**Nœud initial** Marque le début de l'activité. Plusieurs noeuds initiaux indiquent des flots concurrents (exécution en parallèle)



FIGURE 4.20: Nœud initial

**Nœud final** Termine entièrement l'activité. Tous les flots sont terminés



FIGURE 4.21: Nœud final

**Nœud fin de flot** Termine uniquement le flot en question, pas les flots concurrents

**Nœud de décision** Représente un choix entre plusieurs flots

- Un flot d'entrée et plusieurs flots de sortie
- On ne peut choisir qu'un flot de sortie

Les **gardes** ([condition]) : les transitions entre activités peuvent être gardées par des conditions booléennes, mutuellement exclusives. Les gardes sont des labels des transitions



FIGURE 4.22: Nœud fin de flot

dont elles valident le déclenchement. Une condition peut être matérialisée par un losange d'où sortent plusieurs transitions.

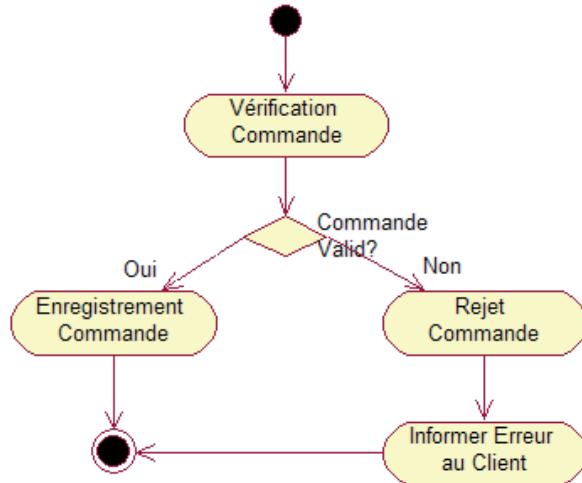


FIGURE 4.23: Transition Alternative

**Nœud de fusion** Un nœud de fusion est un nœud de contrôle qui rassemble plusieurs flots alternatifs entrants en un seul flot sortant. Il n'est pas utilisé pour synchroniser des flots concurrents (c'est le rôle du nœud d'union – fork) mais pour accepter un flot parmi plusieurs.



FIGURE 4.24: Nœud de décision et de fusion

**Synchronisation** Les diagrammes d'activités représentent les synchronisations d'activités au moyen de barres de synchronisation

- Lorsque la barre de synchronisation a plusieurs transitions en **sortie**, on parle de transition de type **fork** qui correspond à une **duplication du flot de contrôle** en plusieurs flots indépendants.
- Lorsque la barre de synchronisation a plusieurs transitions en **entrée**, on parle de transition de type **join** qui correspond à un **rendez-vous** entre des flots de contrôle.

**Itération** Deux manières de représenter l'itération, comme le montre la figure 4.26.

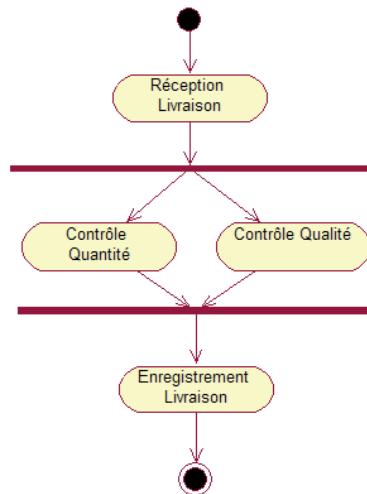


FIGURE 4.25: Synchronisation disjonctive et conjonctive

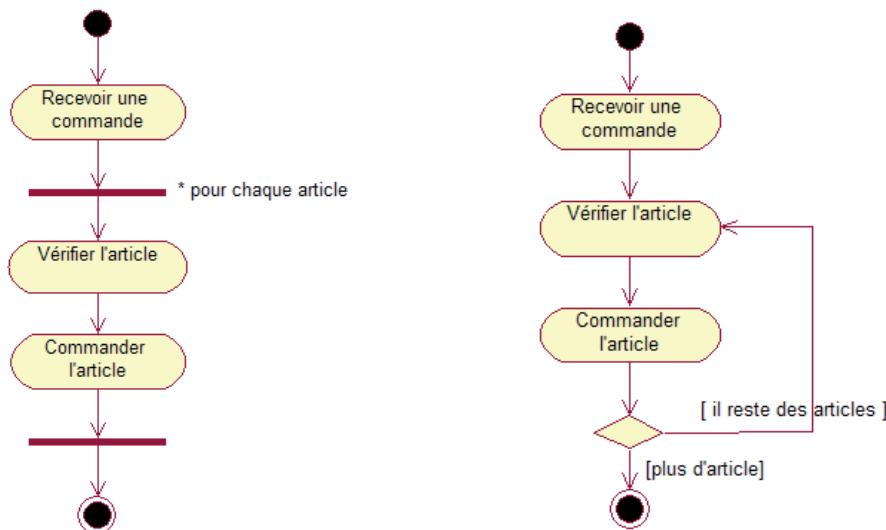


FIGURE 4.26: Itération : Il y a deux manières de représenter l'itération

**Couloirs d'activités (*Swimlanes*)** Les diagrammes d'activités peuvent être découpés en couloirs d'activités pour montrer les différentes responsabilités au sein d'un mécanisme logiciel ou d'une organisation. Chaque responsabilité est assurée par un ou plusieurs objets et chaque activité est allouée à un couloir donné

**Objets dans un diagramme d'activités** Il est possible de faire apparaître des objets dans un diagramme d'activités, soit au sein des couloirs d'activités, soit en dehors de ces couloirs

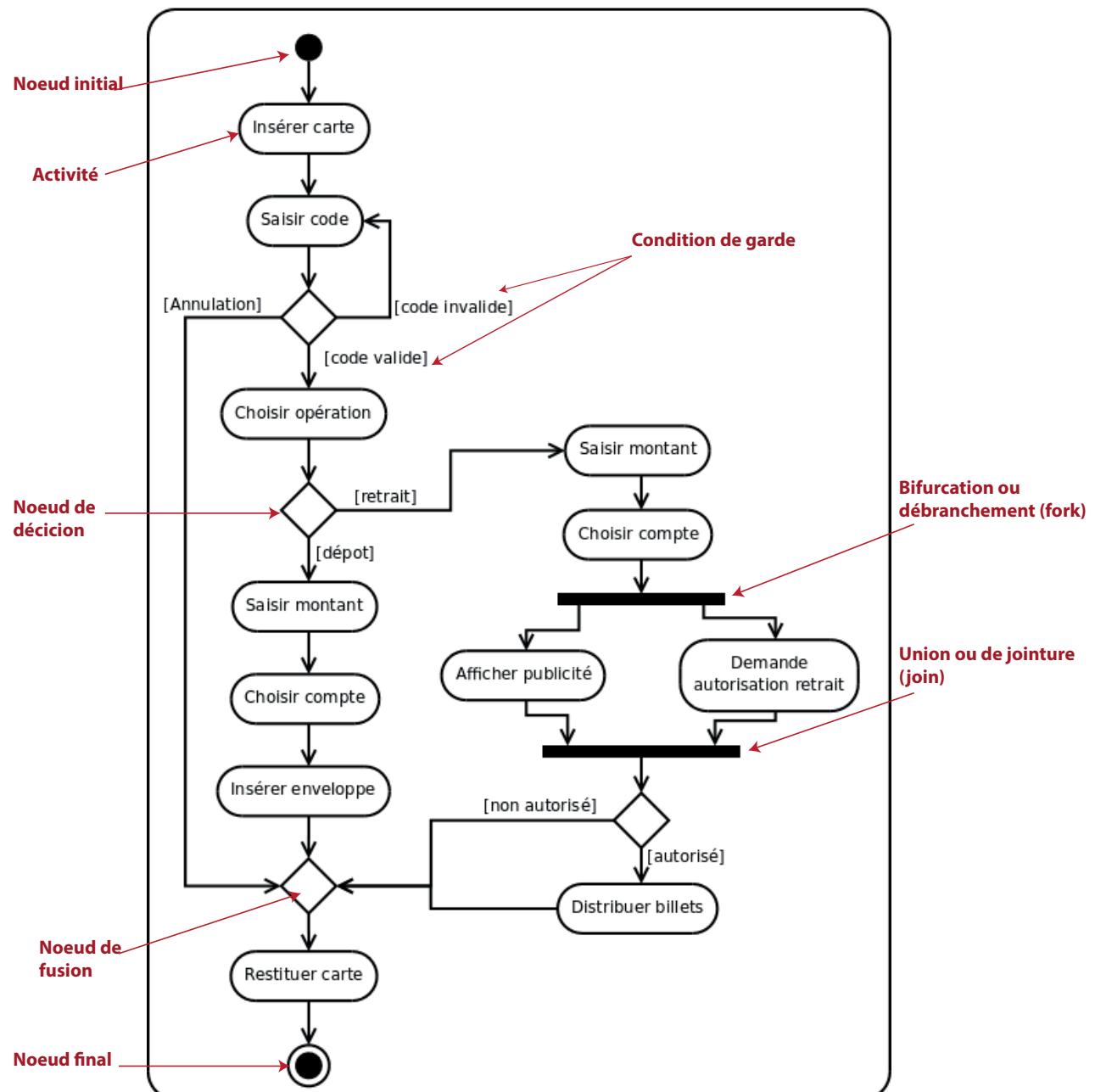
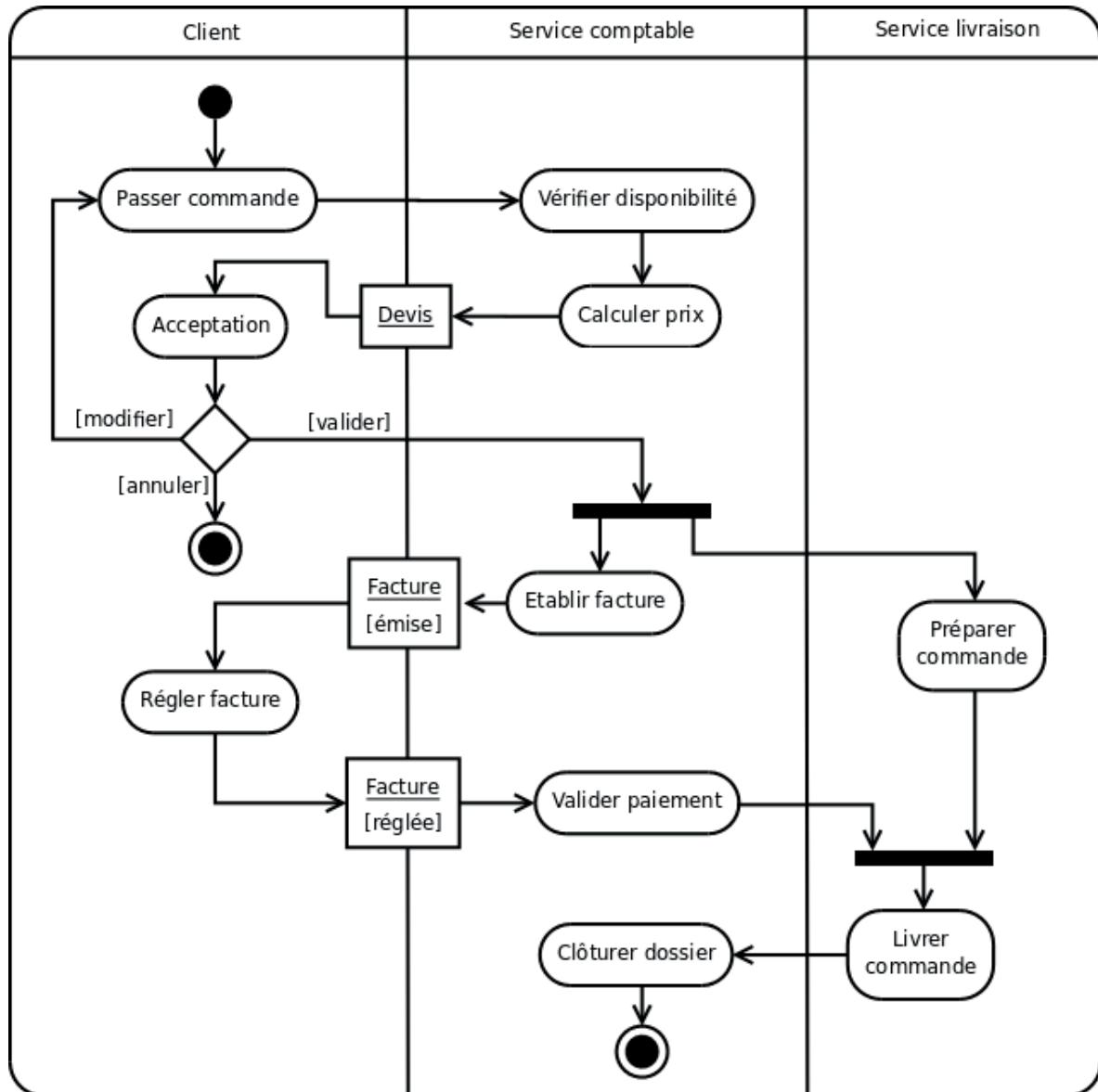


FIGURE 4.27: Éléments d'un diagramme d'activité


 FIGURE 4.28: Couloirs d'activités (*Swimlanes*)