Université de Mostaganem
Faculté des Sciences Exactes et Informatique

Polycopié

Génie logiciel

Dr. Meriem Abid

2018 - 2019

Table des matières

Avant-propos

Longtemps, les informaticiens qui souhaitaient créer un logiciel considéraient qu'il suffisait de savoir programmer pour le mettre en œuvre. Cette idée reçue, a néanmoins atteint ses limites assez rapidement. En effet, avec l'évolution rapide des technologies issues du monde de l'informatique et de l'électronique ainsi que la complexité croissante des systèmes informatiques, savoir programmer ne suffit plus. Les informaticiens chargés de mettre en place un logiciel sont désormais obligés de se focaliser davantage sur les étapes en amont et en aval de la programmation et à exploiter la technologie de façon plus rationnelle et plus organisée.

L'organisation du travail et des efforts est un passage obligé à tout projet de mise en œuvre d'un logiciel complexe et impliquant de nombreuses fonctionnalités et ressources. Cette organisation vise à mettre davantage « le métier » au cœur des préoccupations afin de le comprendre et de proposer des solutions adaptées à ses besoins et non l'inverse.

De nombreuses années de recherche et d'expérimentation ont été nécessaires pour se convaincre de la nécessité d'organiser le travail de mise en œuvre d'un logiciel comme pour n'importe quel produit industriel. Ces recherches ont donné naissance à un nouveau domaine de l'informatique : le génie logiciel.

Objectifs du cours

Ce cours, dispensé aux étudiants en parcours de licence informatique, vise à initier les étudiants à de nouvelles techniques nécessaires à la mise en œuvre de projets logiciels complexes.

De nombreux exemples sont fournis afin d'amener l'étudiant à constater, par lui-même, qu'il ne suffit pas de savoir programmer pour faire un bon logiciel et qu'il est nécessaire de s'initier au domaine de l'ingénierie du logiciel.

Dans ce cours, nous allons plus particulièrement sensibiliser les étudiants à l'importance de la modélisation lors de la mise en œuvre de projets informatiques complexes.

Nous nous focaliserons sur la modélisation orientée objet à travers l'étude du langage de modélisation unifié UML. Une initiation au domaine de la gestion de projet est également apportée car il s'agit, le plus souvent, de méthodes capables d'améliorer la mise en œuvre et le suivi de projets complexes comme celui de la création d'un logiciel. Ainsi, l'étudiant aura une vue globale des différents aspects à considérer pour mettre en place des projets informatiques d'envergure.

L'étudiant devra également apprendre à mieux parler « métier » et pas uniquement informatique pour mieux cerner les besoins des clients et répondre à leurs attentes. En ce sens, la

maîtrise du langage UML est un réel atout car ce dernier est un très bon outil de communication au-delà des nombreux moyens techniques mis à disposition.

A qui s'adresse ce document

Ce polycopié s'adresse à la fois aux étudiants du premier cycle de formation informatique mais également aux enseignants qui souhaitent s'initier au domaine du génie logiciel.

En plus des cours, de nombreux exemples sont donnés afin de consolider les différentes notions étudiées ainsi qu'une série d'exercices corrigés et non corrigés et qui font pour la plupart l'objet de travaux dirigés.

Organisation du document

Le présent polycopié est structuré en plusieurs chapitres. Le chapitre 1 aborde les notions générales liées au génie logiciel telles que le contexte de sa naissance, les cycles de vie des logiciels mais également des notions en gestion de projet.

Le chapitre 2 se penche davantage sur la notion de modélisation et son importance lors de la mise en œuvre d'un logiciel. Nous nous focaliserons plus particulièrement sur la notion de modélisation orientée objet et nous étudierons plus en détails le langage UML et ses différents diagrammes.

Le chapitre 3 aborde le diagramme de cas d'utilisation qui fait partie des diagrammes de comportement et qui est l'un des tous premiers diagrammes à réaliser lors de la mise en œuvre d'un logiciel.

Le chapitre 4 s'intéresse aux diagrammes de classes et d'objets qui représentent les diagrammes structurels et qui sont au cœur d'une démarche de modélisation orientée objet.

Les chapitre 5 et 6 sont consacrés à l'étude de diagrammes UML modélisant la vue dynamique d'un système. Dans le chapitre 5, nous nous intéresserons aux diagrammes d'interaction et plus particulièrement aux diagrammes de séquence et de communication. Dans le chapitre 6, nous aborderons le diagramme d'états-transitions.

Enfin, ce polycopié se termine par une conclusion générale ainsi que les corrigés de certains exercices proposés à la fin des différents chapitres.

Chapitre 1

Introduction au génie logiciel

L'objectif d'un système informatique est d'automatiser le traitement de l'information. Néanmoins, bien souvent les systèmes informatiques posent problèmes au lieu de faciliter et d'accélérer les tâches.

Un système informatique est constitué principalement de deux parties bien distinctes. D'un côté, nous avons le matériel composé essentiellement d'une unité de traitement, d'une unité de stockage et de périphériques. De l'autre côté, nous avons le logiciel dont l'objectif est d'offrir des fonctionnalités répondant à différents besoins. Si on y regarde de plus près, on peut constater que le matériel constituant un système informatique est maintenant relativement fiable, avec des coûts de fabrication de plus en plus réduits et un marché de plus en plus standardisé. Cela a d'ailleurs été prédit par la fameuse « loi de Moore » qui porte sur l'évolution de la puissance de calcul et la complexité du matériel informatique.

Les problèmes liés à l'informatique sont donc essentiellement dus à des problèmes logiciels. On peut alors se poser la question de savoir pourquoi est-ce les logiciels qui posent problèmes. est-ce si difficile de fabriquer un logiciel ?

Pour répondre à ces questions, nous allons, dans ce chapitre, nous intéresser au logiciel et à la façon de le construire à l'aide des principes du « génie logiciel ». Le génie logiciel est une des nombreuses spécialités de l'informatique dont le but est d'améliorer le processus de mise en œuvre d'un logiciel.

L'objectif de ce premier chapitre est d'introduire les principes fondamentaux du génie logiciel. Pour cela, ce chapitre est organisé comme suit. Tout d'abord, nous mettrons en lumière les principales motivations qui ont conduit à la naissance du génie logiciel. Nous analyserons par la suite les différentes étapes qui constituent le cycle de vie d'un logiciel ainsi que la manière dont ces étapes sont agencées.

En outre, quelques notions de gestion de projet, qui représente la partie organisationnelle accompagnant tout projet logiciel, seront également abordées. Enfin, nous conclurons par une réflexion sur la meilleure façon d'organiser ces différentes étapes et feront un point sur les méthodes actuelles dites « Agiles » et ce qu'elles apportent au domaine du génie logiciel.

1.1 Qu'est-ce qu'un logiciel ?

Avant de parler de génie logiciel, il est primordial de comprendre qu'est-ce qu'un logiciel. En effet, jusqu'à présent, c'est la notion de programmes qui prédominait. Mais alors, quelle est la différence entre un programme et un logiciel ?

Un programme est souvent défini comme une suite de données et d'instructions écrites dans un langage de programmation particulier (C, Java, Python, ...) et décrivant un algorithme dont l'objectif est de répondre à une problématique relativement simple et qui est susceptible d'être exécutée par un ordinateur.

Un logiciel est, quant à lui, un ensemble à la fois de programmes mais également de tout le nécessaire pour les rendre opérationnels (fichiers de configuration, images bitmaps, procédures automatiques,...). Il regroupe donc de nombreuses fonctionnalités et adresse par conséquent des problématiques bien plus larges qu'un simple programme.

Une autre différence notable est qu'un programme est généralement écrit par un seul individu alors que dans le cas d'un logiciel, il s'agit de la fabrication collective d'un système complexe qui a fait l'objet d'efforts à la fois en matière de conception, de programmation mais également de tests et qui adresse un problème jugé difficile.

Nous distinguons, généralement, deux principales catégories de logiciels : les logiciels applicatifs et les logiciels systèmes. Un logiciel de traitement de texte ou un logiciel de gestion de la paie sont des exemples de logiciels applicatifs.

Parmi les exemples de logiciels systèmes les plus connus et les plus utilisés nous pouvons citer le système d'exploitation. Un système d'exploitation offre de nombreuses fonctionnalités qui servent à :

- manipuler le matériel informatique,
- diriger le logiciel,
- organiser les fichiers
- gérer la mémoire,
- faire l'interface avec l'utilisateur.
- etc.

1.2 Importance du logiciel

Les logiciels occupent une place de plus en plus importante dans le monde où nous vivons. Aujourd'hui, les économies de tous les pays développés sont dépendantes des logiciels. Le logiciel est, en effet, impliqué dans de nombreux domaines critiques et stratégiques tels que le transport (terrestre, aérien, espace), la télémédecine, les processus industriels (nucléaire, armement), la finance,... où la fiabilité, la sûreté et la sécurité ne sont plus des critères négligeables.

Si des dysfonctionnements au niveau des logiciels venaient à se produire, cela pourrait avoir des conséquences plus ou moins graves selon les secteurs touchés, allant de pertes économiques à des catastrophes graves. Par conséquent, l'ingénierie du logiciel a une place importante et une lourde responsabilité dans le bon fonctionnement des équipements et des institutions.

1.3 Origine du génie logiciel : la crise du logiciel

Le génie logiciel doit son origine à la première grave crise du logiciel ou « crise du logiciel » à la fin des années 1960. Malheureusement, d'autres crises ont suivi. Ces crises sont le fruit d'un manque de qualité des systèmes produits qui ont conduit à des catastrophes aussi bien humaines qu'économiques comme l'illustrent les quelques exemples célèbres (cf. section 1.3.1).

1.3.1 Quelques exemples célèbres

Nous avons choisi quelques exemples célèbres pour illustrer la crise du logiciel.

Perte de la sonde Mariner 1 en 1962 :

Cette sonde spatiale qui devait effectuer un passage à 5.000 km de la planète Vénus a été détruite peu après son lancement à cause d'une déviation de sa trajectoire. La cause de cette déviation est une simple erreur de transcription manuelle d'un symbole mathématique dans la spécification d'un programme, plus exactement une barre suscrite manquante. Cette erreur est surnommée le trait d'union le plus cher de l'histoire.

Explosion de la fusée Ariane V en 1996

Lors du vol inaugural de la fusée Ariane V, celle-ci se brise et explose en plein vol seulement 36,7 secondes après son décollage. La cause de cette explosion est un bug informatique causant un dépassement d'entier dans les registres mémoire des calculateurs électroniques utilisés par le pilote automatique. Un logiciel utilisé sous Ariane IV a été intégré dans Ariane V sans nouvelles validations. Le coût de cette catastrophe est estimé à un demi milliard de dollars.

Perte de la sonde Mars Climate Orbiter en 1998 :

Après 7 mois de voyage, cette sonde spatiale a été détruite en s'approchant d'un peu trop près de la planète Mars. L'erreur était due à une confusion d'unités entre deux équipes distinctes. Coût : 120 Milliards de dollars.

Le bug de l'an 2000 :

Le passage à l'année 2000 a suscité de sérieuses inquiétudes à cause de problèmes de conception et donc de programmation portant sur le format de la date sur deux chiffres au lieu de quatre. Cette erreur de conception a nécessité parfois de revoir en profondeur l'architecture des systèmes d'information, en particulier lorsque certains composants critiques ne pouvaient pas être réparés parce qu'ils étaient trop anciens et n'étaient plus maintenus. Il a donc souvent fallu remplacer complètement des systèmes d'information, généralement spécifiques, par des progiciels du marché compatibles « an 2000 ». Les systèmes plus récents ont pu être réparés par conversion. Au final, aucun problème critique ne s'est produit. Cependant, des sommes considérables, se chiffrant en centaines de milliards de dollars dans le monde, ont dû être dépensées pour prévenir tout incident lors du passage à l'an 2000.

1.3.2 La crise du logiciel en chiffres

La crise du logiciel s'illustre également à travers des chiffres établis dans des rapports et qui pointent les nombreux écueils rencontrés. Ainsi, une étude réalisée par Standish Group sur 50000 projets entre 2011 et 2015 montre les chiffres illustrés dans le tableau 1.1.

	2011	2012	2013	2014	2015
Succès*	29%	27%	31%	28%	29%
Mitigé*	49%	56%	50%	55%	52%
Echec*	22%	17%	19%	17%	19%

TABLE 1.1 – Crise du logiciel en chiffres

*Succès : livré à temps, sans surcoût et client satisfait.

*Mitigé : client non satisfait ou dépassement du coût ou des délais.

*Echec : abandonné en cours de route

1.3.3 Les causes de cette crise

Plusieurs causes peuvent expliquer de tels chiffres. Les problèmes de communication peuvent être une cause importante de dysfonctionnement comme l'attestent les exemples célèbres cités en section 1.3.1. Ces problèmes se traduisent par une mauvaise compréhension entre les clients porteurs du besoin et les individus qui souhaitent répondre à ces besoins. Plusieurs aspects peuvent mettre à mal cette communication entre client/utilisateur et développeurs comme le jargon métier employé par le client et qui est incompréhensible par les informaticiens ou vice-versa.Ces problèmes apparaissent également entre individus au sein d'une même équipe et bien évidemment s'accentuent à mesure que l'équipe est importante.

La réflexion métier et l'expression du besoin des clients peuvent également s'avérer pas assez mature à mesure que le projet avance. Ceci entrainera non seulement des problèmes au niveau des étapes de spécification et de conception mais aussi une évolution très fréquente (trop de modifications) du logiciel.

De même, l'équipe chargée du développement peut négliger parfois l'évolution de la perception d'un problème qui n'est pas répercutée à temps. Le logiciel ne pourra ainsi pas répondre au besoin évolutif des clients et peut entrainer un réexamen complet des étapes d'analyse de besoins et de conception débouchant sur des problèmes de délais et de coûts importants.

De manière générale, l'examen des causes de cette crise du logiciel est instructif car cela montre que les échecs ne proviennent pas uniquement de l'informatique, mais également de la maîtrise d'ouvrage (i.e. le client).

Pour ces raisons, le développement de logiciels dans un contexte professionnel doit suivre souvent des règles strictes encadrant la conception et permettant le travail en groupe et la maintenance du code. Ces règles sont décrites dans cette nouvelle discipline qu'est : le génie logiciel.

1.4 Naissance du génie logiciel

Avant 1968, la production de logiciel s'apparentait presqu'exclusivement à la programmation où l'intelligence est le facteur déterminant. Cette idée reçue et encore répandue a conduit à de graves crises du logiciel qui se traduisent principalement par :

— l'augmentation des coûts ;
— les difficultés de maintenance et d'évolution ;
— la non-fiabilité ;
— le non-respect des spécifications ;
— le non-respect des délais

En 1968, suite à la 1ère crise du logiciel, une conférence s'est tenue à Garmisch-Partenkirchen, sous le parrainage de l'OTAN. En effet, l'informatique souffrait d'une mauvaise image de marque car elle coûtait très cher et désorganisait les entreprises au lieu de les aider. Les acteurs de l'informatique ont abouti à la conclusion qu'une approche empirique de développement de logiciel ne suffisait pas.

C'est lors de cette conférence que l'expression « Génie Logiciel » (ou Software Engineering) a été introduite pour la première fois. L'objectif étant de transformer le processus de création de logiciel afin de ne pas dépasser les coûts, les délais et d'obtenir un logiciel satisfaisant les qualités attendues. Dès lors, comme pour toute activité industrielle, le génie logiciel doit prendre en charge les aspects techniques, méthodologiques et managériaux lors de l'élaboration d'un logiciel.

1.4.1 Définition du génie logiciel

Le génie logiciel peut être définit de plusieurs façons. Nous en avons sélectionné deux.

Définition 1 : « Le génie logiciel est l'application d'une approche systématique, disciplinée et quantifiable au développement, à l'exploitation et à la maintenance d'un logiciel c'est-à-dire l'application de l'ingénierie au logiciel » [1].

Définition 2 : Le Génie Logiciel 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.

1.4.2 Motivation

Le domaine du génie logiciel a pour objectif de répondre à un problème qui s'énonçait en deux constatations :

— d'une part le logiciel n'était pas fiable,
— d'autre part, il était incroyablement difficile de réaliser dans des délais prévus des logiciels satisfaisant leur cahier des charges.

1. La norme IEEE 610.12

En résumé le génie logiciel s'intéresse à la manière dont le code source d'un logiciel est spécifié puis produit afin d'optimiser le coût de développement du logiciel.

L'importance d'une approche méthodologique s'est imposée à la suite de la crise de l'industrie du logiciel à la fin des années 1960 et continue de l'être puisque la crise du logiciel n'a jamais réellement cessée.

1.5 Les différents aspects du génie logiciel

Le génie logiciel regroupe plusieurs aspects indispensables à la fabrication de produits logiciels (voir figure 1.1) :

L'aspect qualité vise à s'assurer de la conformité des besoins et de l'adéquation avec l'usage attendu à travers la mise en place de plusieurs tâches telles que l'assurance qualité, le contrôle de qualité, etc. Cet aspect sera détaillé en section 1.5.1.

L'aspect organisationnel nécessaire à la bonne conduite de projet dont l'objectif est principalement de maîtriser les coûts et les délais en découpant le projet en plusieurs tâches, en planifiant les tâches et en affectant les ressources nécessaires à chaque tâches. Cet aspect sera traité plus en détail dans la section 1.5.2.

L'aspect technique qui représente les différentes étapes techniques nécessaires à la fabrication d'un logiciel et qui s'étend du recueil des besoins jusqu'à la maintenance. Cet aspect sera détaillé en section 1.5.3.

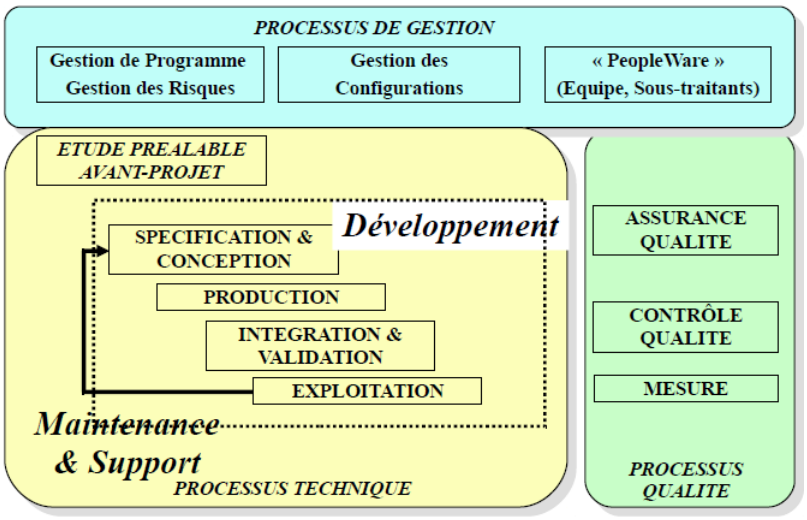


FIGURE 1.1 – Aspects du Génie Logiciel

1.5.1 Aspect qualité

La gestion de la qualité lors du processus de mise en œuvre d'un logiciel est une préoccupation présente tout au long du cycle de vie d'un logiciel.

La gestion de la qualité doit garantir une certaine adéquation entre les besoins du client (exprimés de façon explicites) et les caractéristiques d'un produit logiciel.

Elle implique un contrôle strict des productions intermédiaires, afin d'éviter de détecter trop tardivement un problème de qualité et qui sera bien plus difficile à résoudre lorsque le produit logiciel sera finalisé.

Les qualités attendues d'un logiciel sont nombreuses et dépendent, entre autres, du domaine d'application et des outils utilisés. Parmi ces qualités, nous pouvons citer :

- La validité : C'est l'aptitude d'un produit logiciel à remplir exactement ses fonctions, définies par le cahier des charges et les spécifications.
- La fiabilité et la robustesse : C'est l'aptitude d'un produit logiciel à fonctionner dans des conditions anormales.
- L'extensibilité (ou maintenabilité) : C'est la facilité avec laquelle un logiciel se prête à sa maintenance, c'est-à-dire à une modification ou à une extension des fonctions qui lui sont demandées.
- La réutilisabilité : C'est l'aptitude d'un logiciel à être réutilisé, en tout ou en partie, dans de nouvelles applications (lié à la modularité).
- La compatibilité : C'est la facilité avec laquelle un logiciel peut être combiné avec d'autres logiciels.
- L'efficacité : C'est l'utilisation optimale des ressources matérielles.
- La portabilité : C'est la facilité avec laquelle un logiciel peut être transféré sous différents environnements matériels et logiciels.
- La vérifiabilité : C'est la facilité de préparation des procédures de test.
- La sécurité : C'est l'aptitude d'un logiciel à protéger son code et ses données contre toute forme d'attaques (accès non autorisés, déni de service, etc.).
- La facilité d'emploi : C'est la facilité d'apprentissage, d'utilisation, de préparation des données, d'interprétation des erreurs et de rattrapage en cas d'erreur d'utilisation.

Ces qualités peuvent être regroupés selon que l'on soit développeur ou utilisateur du logiciel.

En outre, certains critères sont parfois contradictoires, Il est donc difficile de les retrouver en même temps au sein d'un produit logiciel. Des compromis doivent être trouvés en fonction du contexte.

1.5.2 Aspects organisationnels : conduite de projet

Lors de la mise en œuvre d'un logiciel, il est nécessaire de s'appuyer sur une bonne gestion de projet afin d'atteindre le but fixé. La conduite de projet représente l'aspect organisationnel lors de la mise en œuvre d'un logiciel et fait intervenir plusieurs tâches parmi lesquelles :

— l'estimation des coûts et des délais
— la planification des tâches (diagrammes PERT et Gantt, etc.)
— le suivi (gestion de l'équipe, des échéances et fournitures à remettre au client, ...).
— la gestion des risques (mesures préventives ou correctives)
— la gestion des configurations (gestion des différents composants, de leurs versions et de la cohérence globale des versions, maîtrise des évolutions, ...).
— la gestion des documents associés aux différentes phases de la production du logiciel

Les objectifs d'une bonne conduite de projet consiste, d'une part, à éviter les dépassements de délai et de coût. D'autre part, à faire un logiciel de qualité.

La conduite de projet est un domaine à part entière qui nécessite une étude approfondie. Nous avons choisi, dans ce polycopié, de nous intéresser à la planification des tâches à travers l'étude des diagrammes PERT et Gantt.

1.5.2.1 La planification dans la conduite de projet

La phase de planification revêt une importance toute particulière lors de la mise en œuvre d'un logiciel. Elle permet un agencement des tâches suite à un découpage du projet en différentes tâches Elle implique également une meilleure identification des besoins et un contrôle plus important du suivi de projet.

La mise en œuvre de méthodes pour la planification des tâches vise à résoudre les problèmes suivants :

- Quel est le temps nécessaire pour réaliser l'ensemble du projet ?
- A quelle date doit commencer chaque tâche ?
- Quelles sont les tâches critiques ?

Il existe une multitude de méthodes mises en œuvre pour la planification des tâches. Nous allons en étudier deux : les diagrammes PERT et Gantt. Il existe également plusieurs outils informatiques concernant la planification des tâches parmi lesquels :

— Microsoft Project,
— GanttProject,
— Microsoft Excel ou autre tableur.

PERT (Programm Evaluation Review Technique) :

L'objectif de cette méthode est de réduire la durée totale d'un projet par une analyse détaillée des tâches ou activités élémentaires et de leur enchaînement.

La méthode s'appuie en grande partie sur une représentation graphique qui permet de bâtir un « diagramme (ou graphe) PERT ».

Il existe deux grandes familles de diagramme PERT, le PERT potentiel-étapes et le PERT potentiel-tâches. La première (potentiel-étapes), et la plus ancienne, ne sera pas détaillée dans ce cours car elle est moins souple et moins utilisée. Nous nous focaliserons donc sur le PERT à potentiel-tâches dont un exemple est illustré par la figure 1.2. Comme nous pouvons le voir, dans cette représentation les tâches sont les sommets du graphe. Les arcs liant les sommets et toujours orientés dans le sens du défilement du temps représentent, pour leur part, les relations de dépendance qui existent entre les tâches.

Méthodologie de construction d'un diagramme PERT potentiel-tâches :

— Établir la liste des tâches (suite au découpage du projet en plusieurs tâches).
— Déterminer les antériorités : tâches antérieures qu'il est nécessaire de terminer pour pouvoir faire une tâche.
— Déterminer les niveaux d'exécution ou rang des tâches.
— Construire le graphe PERT

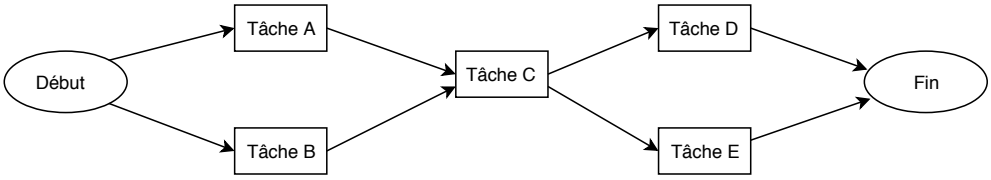


FIGURE 1.2 – Graphe PERT des potentiels-tâches

— Calculer la durée du projet, les dates début et de fin des tâches au plus tôt et au plus tard.
— Déterminer le chemin critique
— Mettre en évidence les marges

Niveau d'exécution d'une tâche :

Pour déterminer le rang (ou niveau) d'exécution d'une tâche, il faut procéder comme suit :

Les tâches sans antécédents constituent le niveau 1. On identifie, ensuite, les tâches dont les antécédents sont du niveau 1. Ces tâches constituent le niveau 2, et ainsi de suite...

Calcul des dates au plus tôt, au plus tard :

Les dates au plus tôt (début (DTO)/fin (FTO)) sont obtenues en cumulant la durée des tâches qui les précèdent sur la séquence la plus longue. Pour les calculer, il faut procéder comme suit :

— On initialise le sommet DEBUT avec une date au plus tôt = O
— date au plus tôt de la tâche j = Maximum (date au plus tôt de i + durée $T_{i,j}$) pour tous les prédécesseurs i de j

Les dates au plus tard (Début (DTA)/fin (FTA)) sont les dates auxquelles doivent être exécutées les tâches sans remettre en cause la durée optimale de fin de projet. Pour les calculer, il faut pour cela procéder comme suit :

— On initialise à l'étape terminale le dernier sommet par la date au plus tard. celle-ci doit être égale à la date au plus tôt
— date au plus tard i = Minimum (date au plus tard de j - durée $T_{i,j}$) pour tous les successeurs j de i.

Chemin critique :

C'est le chemin dont la succession des tâches donne la durée d'exécution la plus longue. Pour toutes les tâches du chemin critique, les dates au plus tôt et au plus tard coïncident. Ces tâches n'admettent donc aucune marge. Un quelconque retard d'une tâche critique impactera la durée totale du projet.

Diagramme de Gantt :

Le diagramme de Gantt est un graphique (ou chronogramme) qui consiste à placer les tâches chronologiquement en fonction des contraintes techniques de succession (contraintes d'antériorités).

L'axe horizontal des abscisses représente le temps et l'axe vertical des ordonnées les tâches.

On représente chaque tâche par un segment de droite dont la longueur est proportionnelle à sa durée.

L'origine du segment est calée sur la date de début au plus tôt de l'opération (« jalonnement au plus tôt ») et l'extrémité du segment représente la fin de la tâche. On trace le plus souvent le diagramme de Gantt au plus tôt ou « jalonnement au plus tôt » et éventuellement au plus tard « jalonnement au plus tard ».

Ce type de graphe a l'avantage d'être très facile à lire, mais présente l'inconvénient de ne pas représenter l'enchaînement des tâches. Cette méthode est généralement utilisée en complément du graphe PERT.

1.5.3 Aspects techniques

Le processus technique d'élaboration du logiciel est constitué de plusieurs étapes qui forment ce que l'on appelle le cycle de vie du logiciel. Contrôler et maîtriser la succession de ces étapes est primordial pour obtenir le logiciel attendu.

Le génie logiciel touche au cycle de vie des logiciels i.e. toutes les étapes du développement d'un logiciel, de sa conception à sa disparition.

L'objectif d'un tel découpage est, d'une part, de définir des jalons intermédiaires permettant la validation du développement logiciel, c'est-à-dire la conformité du logiciel avec les besoins exprimés. D'autre part, de vérifier le processus de développement, c'est-à-dire l'adéquation des méthodes mises en œuvre.

L'origine de ce découpage provient du constat que les erreurs ont un coût d'autant plus élevé qu'elles sont détectées tardivement dans le processus de réalisation.

Le cycle de vie doit permettre de détecter les erreurs au plus tôt et ainsi de maîtriser la qualité du logiciel, les délais de sa réalisation et les coûts associés.

1.6 Cycle de vie d'un logiciel

Les principales étapes constituant le cycle de développement d'un logiciel peuvent être décrites comme suit.

Étape 0 : Étude de faisabilité

Cette étape consiste à déterminer si le développement proposé vaut la peine d'être mis en œuvre, compte tenues des attentes et de la difficulté de développement. Elle peut également impliquer une étude de marché afin de déterminer s'il existe un marché potentiel pour le produit logiciel en question.

Étape 1 : Analyse des besoins et Spécification

Dans cette étape, il s'agit de déterminer les fonctionnalités que doit posséder le logiciel. Cela implique tout d'abord une analyse des besoins du client dans le but de collecter ses exigences. Cela aboutit généralement à un document appelé le cahier des charges. On spécifie ensuite ce que le logiciel doit faire (et pas comment il va le faire) à travers l'écriture de différents modèles, cela aboutit à un document appelé les spécifications fonctionnelles.

Étape 2 : Organisation du projet

Dans cette étape, il s'agit de déterminer comment on va développer le logiciel, cela implique : Une analyse des coûts afin d'établir une estimation du prix du projet. La planification des tâches et l'établissement d'un calendrier avec pour chaque tâche ses contraintes (date de début, date de fin, tâche critique, ..) ainsi que la répartition des tâches. Dans cette étape, on peut également déterminer les actions qui permettront de s'assurer de la qualité du produit fini, c'est l'assurance qualité du logiciel.

Étape 3 : Conception

C'est lors de cette étape qu'est déterminée la façon dont le logiciel fournit les fonctionnalités attendues. Plusieurs sous-étapes sont impliquées lors de la phase de conception. On distingue :

Conception architecturale : il s'agit au départ de déterminer la structure du système et de procéder à la décomposition du logiciel en « composants », de prévoir la manière et l'ordre d'intégration de ces différents composants (incréments) et de concevoir des tests associés à l'intégration.

Conception des interfaces : cette étape permet de déterminer la façon dont les différentes parties du système agissent entre elles.

Conception détaillée : c'est dans cette étape qu'il faut déterminer les algorithmes pour les différentes parties du système et concevoir des tests qui valideront les implémentations.

Étape 4 : Implémentation

C'est dans cette étape que le logiciel est codé dans un langage de programmation. L'implémentation constitue l'étape prédominante du cycle de vie d'un logiciel et s'effectue fréquemment au détriment des autres étapes. Pourtant, comme le montre la plupart des études, il s'avère toujours plus coûteux de revenir sur des erreurs liés à l'analyse de besoin ou à la conception dans les phases en aval.

Étape 5 : Test

Cette étape comporte plusieurs catégories de tests à effectuer :

Tests unitaires : il s'agit d'exécution contrôlée des tests de chaque procédure/méthode tels que définis préalablement. Les tests unitaires sont effectués par les développeurs.

Test d'intégration : ces tests visent à contrôler l'étape d'intégration.

Test d'acceptation (ou test système) : ces tests sont réalisés par le fournisseur. Il s'agit de tests globaux, en conditions réelles d'exploitation.

Recette : ces tests sont réalisés par le client. En effet, même si des tests unitaires et d'intégration ont été effectués, il est important que la maîtrise d'ouvrage procède à ses propres tests dits « tests métiers ». La recette se déroule en 2 phases :

(1) la recevabilité qui permet au client acquéreur de vérifier que tous les composants commandés ont bien été livrés.

(2) la qualification pendant laquelle le client utilisateur va vérifier que tous les composants livrés fonctionnent correctement et répondent aux spécifications.

Étape 6 : Déploiement (mise en production)

Il s'agit du déploiement sur le site du client du logiciel.

Étape 7 : Maintenance

Elle comprend toutes les actions correctives (maintenance corrective) et évolutives (maintenance évolutive) sur le logiciel.

La maintenance corrective implique la gestion et la correction des bugs.

La maintenance évolutive est nécessaire lorsqu'il y a par exemple des changements de systèmes d'exploitation ou de matériels mais également des problèmes de performance ou d'évolution des fonctionnalités.

L'étape de maintenance nécessite la mise ne œuvre de tests de non-régression d'une version à l'autre afin de s'assurer que les fonctionnalités existantes sont toujours valides malgré les changements. La maintenance est, sans doute, l'une des étapes les plus douloureuses et les plus coûteuses !

D'autant plus douloureux qu'il peut y avoir un ensemble de versions à maintenir simultanément. En outre, le coût de maintenance représente 2 à 4 fois le coût de développement.

1.7 Modèles de cycle de vie d'un logiciel

Un modèle de cycle de vie spécifie la manière dont les différentes étapes formant le cycle de vie d'un logiciel sont agencées.

La séquence et la présence de chacune de ces étapes dans le cycle de vie dépendent donc du choix d'un modèle de cycle de vie entre le client et l'équipe de développement.

1.7.1 Modèles linéaires

sont des modèles prédictifs qui s'appuient sur la planification en amont des différentes phases lors de la mise en œuvre du logiciel.

Cette planification très précise implique que les différentes étapes doivent être accomplies de façon rigoureuse ce qui peut rendre parfois difficile la prise en compte de certaines évolutions. Les modèles linéaires sont principalement les modèles en cascade et en V.

1.7.1.1 Principe du modèle en cascade (ou Waterfall)

Ce modèle est l'un des premiers modèles proposés de cycle de vie. Inspiré du modèle de Royce (1970), le modèle en cascade répond aux principes suivants :

— Enchaînement séquentiel des différentes étapes
— Chaque étape se termine par la production de certains livrables ou logiciels
— Une étape ne peut démarrer que si la précédente est finie

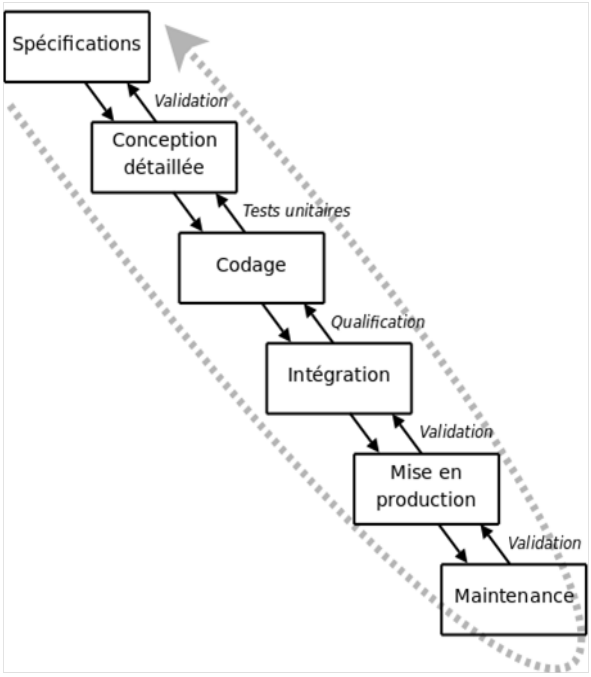


FIGURE 1.3 – Modèle en cascade

Avantages :

Ce modèle offre les avantages suivants :

— Facile à comprendre et à utiliser
— Procédé structuré idéal pour la gestion et le suivi de projets
— Idéal lorsque la qualité est plus importante que les coûts et les délais

Inconvénients :

Même s'il offre de nombreux avantages, le modèle en cascade possède les inconvénients suivants :

— Très faible implication du client : alors que les besoins des clients sont très rarement stables et clairement définis
— Le produit n'est visible qu'à la fin, les risques se décalent donc vers la fin
— La vérification du bon fonctionnement du système est trop tardive : lors de la phase d'intégration, ou pire, lors de la mise en production.

Contexte d'utilisation :

L'utilisation du modèle en cascade est adaptée à des projets de petites tailles lorsque les besoins des clients sont connus et stables et qu'ils ne risquent pas d'évoluer trop rapidement. En outre, le modèle en cascade est intéressant à utiliser lorsque la technologie est bien maîtrisée et le produit bien connu ; par exemple quand il s'agit de la création d'une nouvelle version d'un produit existant ou du portage d'un produit sur une autre plateforme.

1.7.1.2 Principe du modèle en V :

Tout comme le modèle en cascade, le modèle en V est également linéaire. Ce dernier est une variante du modèle en cascade qui met plus l'accent sur la vérification et la validation.

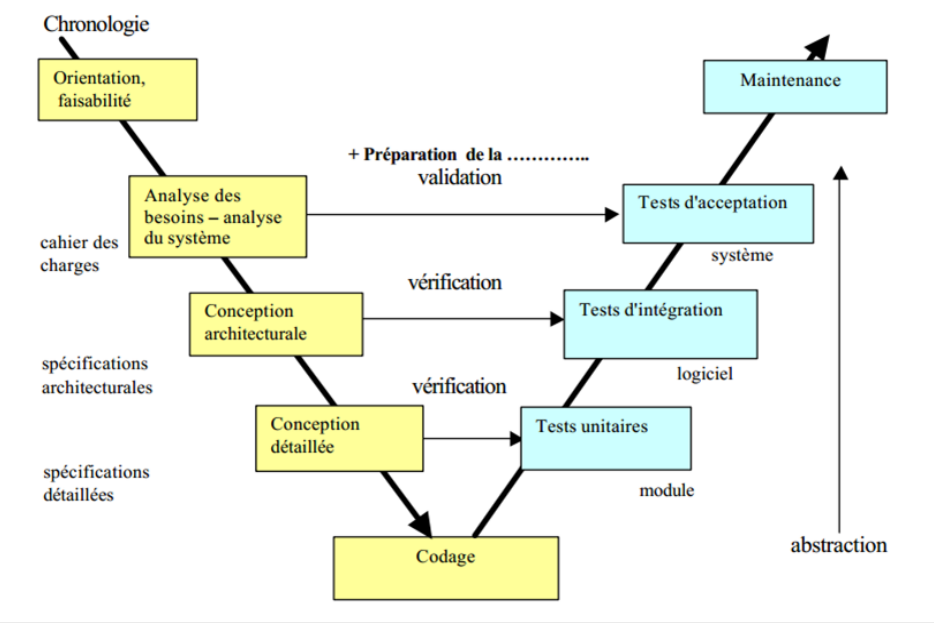


FIGURE 1.4 – Modèle de cycle de vie en V

Le modèle en V tient davantage compte de la réalité puisque le processus de développement n'est pas réduit à un simple enchaînement de tâches séquentielles. Le V est parcouru de gauche à droite et c'est lors des phases descendantes que l'on produit les guides pour les phases montantes. Toutefois, même si une réflexion est apportée aux étapes de validation et de vérification en amont (voir figure 1.4), les activités de construction les précèdent quand même.

Avantages :

- Le modèle en V permet de s'intéresser dès les premières étapes de développement de logiciel aux phases de validation et de vérification.
  - C'est en phase de spécification que l'on se préoccupe de leur validation,
  - C'est en phase de conception globale que l'on se préoccupe des tests d'intégration
  - C'est en phase de conception détaillée que l'on prépare les tests unitaires.

Inconvénients :

— Un modèle parfois difficile à appliquer rigoureusement dans la pratique
— Un modèle toujours séquentiel avec une validation tardive du système
— Les validations intermédiaires n'empêchent pas la transmission des insuffisances

Contexte d'utilisation :

Tout comme le modèle en cascade, le modèle en V est adapté aux problèmes bien connus et idéal quand les besoins sont stables.

1.7.2 Modèles par prototypage

Utilisé lors des phases amont du projet, à savoir l'analyse des besoins et la spécification fonctionnelle, ce modèle permet une validation par expérimentation et une plus grande implication du client. Ce modèle offre une version d'essai du logiciel soit pour tester les différents concepts et exigences, soit pour montrer aux clients les fonctions que l'on veut mettre en œuvre. Lorsque le client a donné son accord, le développement peut commencer (en suivant par exemple un modèle de cycle de vie linéaire).

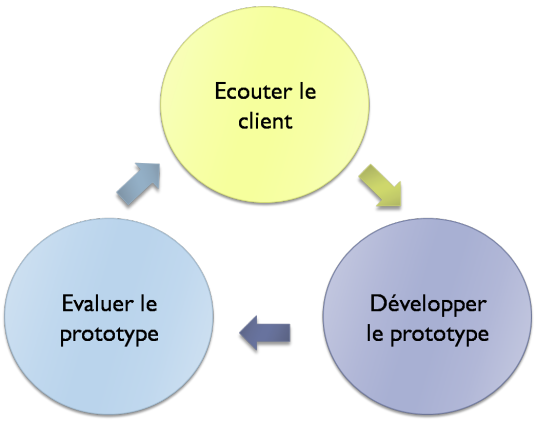


FIGURE 1.5 – modèle par prototypage

Ainsi, les efforts consacrés au développement d'un prototype sont le plus souvent compensés par ceux gagnés à ne pas développer de fonctions inutiles.

Deux sous-modèles dérivent du modèle par prototypage :

1. Prototypage par expérimentation
2. Prototypage évolutif

1.7.2.1 Prototypage par expérimentation :

Ce modèle est généralement utilisé au niveau de la phase de conception afin de s'assurer de la faisabilité de parties critiques et également de valider des options de conception. Souvent inutile, le prototype résultant est abandonné (voire jeté) après développement.

1.7.2.2 Prototypage par évolution :

Dans ce modèle, la première version du prototype est un embryon. Il s'agit d'un développement par incrément et où chaque prototype fait l'objet d'un cycle spécification-conception-implémentation-test. Dans ce modèle, on retrouve la notion de cycle incrémental.

Avantages :

Le modèle par prototypage offre les avantages suivants :
- Permet une implication plus élevée du client
- S'adapte plus rapidement aux changements des besoins
- Permet d'avoir une meilleure vision du produit

Inconvénients :

Même s'il possède plusieurs avantages, le modèle par prototypage peut entraîner un code peu ou mal structuré. Par ailleurs, le processus peut ne jamais s'arrêter.

Contexte d'utilisation :

Le modèle par prototypage peut être combiné avec des modèles linéaires (en cascade ou en V) pour mieux clarifier les besoins du client. En outre, ce modèle est particulièrement indiqué dans les cas suivants :
- Lorsque les besoins du client varient ou nécessitent des clarifications
- Lorsque des livraisons rapides sont exigées par le client

1.7.3 Modèles en spirale

Proposé par Boehm en 1988, le modèle en spirale est basé sur le prototypage évolutif et sur la gestion de risque. A chaque cycle, il s'agit de recommencer les étapes suivantes :
- consultation des clients
- analyse des risques
- conception
- implémentation
- tests
- planification du prochain cycle

Chaque boucle de la spirale représente une phase du processus du logiciel. La boucle la plus interne concerne la faisabilité du système, la boucle suivante concerne la conception, etc. Chaque boucle de la spirale est décomposée en quatre secteurs :

1. Détermination des objectifs,
2. Identification des risques et leur réduction,
3. Développement et validation/vérification,
4. Planification de la boucle suivante.

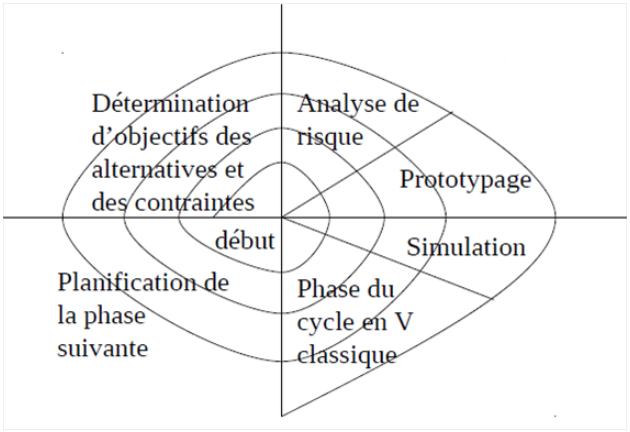


FIGURE 1.6 – Modèle en spirale

Avantages :

Le modèle en spirale permet une évaluation continue du procédé de mise en œuvre du logiciel. Il offre ainsi la possibilité d'avoir un feedback rapide des clients. Par ailleurs, grâce à une meilleure gestion des risques, le modèle en spirale permet une minimisation de l'impact des risques sur le projet.

Inconvénients :

Le modèle en spirale est complexe. La spirale peut s'éterniser d'autant que l'évaluation des risques peut prendre beaucoup de temps. En outre, les développeurs doivent être réaffectés pendant les phases de non développement.

Contexte d'utilisation :

Le modèle en spirale est indiqué dans les cas suivants :
- Lorsque le risque du projet est considérable
- Lorsque le prototype est nécessaire

Il est également particulièrement intéressant pour les nouveaux produits.

1.7.3.1 Bilan des modèles traditionnels

Les modèles traditionnels sont théoriquement complets mais proposent des approches de développement rigides et figées. Ils supposent deux choses :

1. l'analyse est capable de spécifier correctement les besoins
2. et que ces besoins sont stables

Or, quelques chiffres extraits du livre de C. Larman[2] concernant les fonctionnalités spécifiées/cycle en cascade montrent que :
- 45% d'entre eux ne sont JAMAIS utilisées
- 19% sont Rarement utilisées

2. C. Larman : *UML 2 et les design patterns* (2005). Campus Press

— 16% sont Parfois utilisées
— 13% sont Souvent utilisées
— 7% sont Toujours utilisées

Ceci explique pourquoi, sur le terrain, on constate que 90% des dépenses concernent la maintenance et l'évolution.

Toujours selon la même source, la répartition des coûts de maintenance par secteur est la suivante :

— Extensions utilisateur : 41,8%
— Correction d'erreurs : 21,4%
— Modification format de données : 17,4%
— Modification de matériel : 6,2%
— Documentation : 5,5%
— Efficacité : 4%

Comme le montre les chiffres sur la répartition des coûts de maintenance, le principal poste de dépense est celui de l'extension des fonctionnalités pour les utilisateurs.

Ceci implique que dans la plupart des cas, la phase d'analyse n'est pas capable de spécifier correctement ces besoins car ces derniers ne sont pas forcément stables, ni bien formulés.

1.7.4 Modèles Agiles en vogue

Les modèles Agile sont des modèles itératifs et adaptatifs qui visent à prendre davantage en compte les changements et les évolutions (besoins des clients, évolution technologique,...). Les méthodes Agiles sont d'ailleurs présentées comme une alternative aux méthodes traditionnelles considérées comme lourdes et bureaucratiques.

L'approche agile s'inscrit dans une démarche d'amélioration du développement logiciel par la pratique. Cette approche s'est grandement popularisée suite à la publication du manifeste agile (Agile manifesto). Ce manifeste repose sur quatre valeurs (illustrées par figure 1.7) et douze principes.

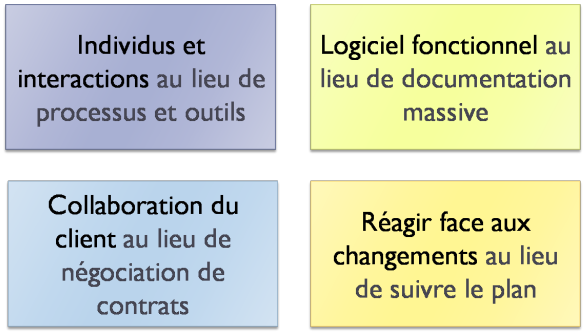


FIGURE 1.7 – Valeurs Agiles

1.7.4.1 Valeurs agiles

Les quatre valeurs agiles sont décrites conne suit :

1. Individus et interactions au lieu de processus et outils : signifie que les collaborateurs et que le travail d'équipe sont la clé du succès et qu'il est plus important de construire une équipe que d'offrir une surabondance d'outils.

2. Logiciel fonctionnel au lieu de documentation massive : même si un code sans documentation est une catastrophe, il est parfois pire de produire trop de documents ou des documents qui ne reflètent pas réellement le code. Il est donc important de produire des documents fidèles et aussi courts que possibles.

3. Collaboration du client au lieu de négociation de contrats : de façon générale, les produits réussis sont ceux qui ont impliqués de façon régulière les clients.

4. Réagir face aux changements au lieu de suivre le plan : il est très difficile (voire impossible) de prévoir dès le début la totalité du logiciel. De multiples changements liés à la technologie et surtout aux besoins des clients peuvent intervenir. Il est donc préférable de faire des plannings courts mais très détaillés et de planifier de façon plus souple le travail à moyen et à long terme.

Les valeurs et principes du Manifeste agile sont défendus par l'Agile Alliance.

1.7.4.2 Principes du manifeste

Parmi les douze principes agiles figurent :
- Toujours satisfaire le client à travers des livraisons rapides et continues
- Livrer fréquemment un système fonctionnel
- Les clients et les développeurs doivent collaborer
- La première mesure d'avancement c'est un logiciel fonctionnel
- Le développement doit être durable et à un rythme constant
- La bonne conception et l'excellence technique augmentent l'agilité
- Simplifier au maximum
- Les meilleures architectures, besoins et conceptions proviennent d'équipes qui s'organisent d'elles-mêmes

1.7.4.3 Quelques méthodes Agiles

Les méthodes agiles ne sont pas apparues avec l'Agile manifesto en 2001 mais celui-ci détermine leur dénominateur commun et consacre le terme d'« Agile » pour les référencer.

Parmi les principales méthodes agiles, nous pouvons citer :
- eXtreme Programmaing (XP)
- Scrum
- Rapid Application Developement (RAD)
- Adaptative Software Development (ASD)

1.8 Cycle de vie idéal

Il n'y a pas de modèle idéal pour le processus de fabrication d'un logiciel, tout dépend.

Le modèle en cascade ou en V est risqué pour les développements innovants car les spécifications et la conception risquent d'être inadéquates et souvent remises en cause. Le modèle incrémental est risqué car il ne donne pas beaucoup de visibilité sur le processus complet. Le modèle en spirale est un canevas plus général qui inclut l'évaluation des risques mais qui peut être lourd et complexe à mettre en place.

Souvent, un même projet peut mêler différents modèles, comme le modèle par prototypage pour les sous-systèmes à haut risque et le modèle en V pour les sous-systèmes bien connus et à faible risque.

Si un modèle idéal de développement logiciel devait exister, ce dernier devra permettre de :
- Bien comprendre les demandes des utilisateurs finaux
- Tenir compte des changements du cahier des charges
- Empêcher la découverte tardive de défauts sérieux dans le projet
- Traiter au plus tôt tous les points critiques du projet
- Bien communiquer (avec le client, entre les membres d'une équipe,...)
- Bien maîtriser la complexité
- Favoriser la réutilisation
- Définir une architecture robuste
- Faciliter le travail en équipe

1.8.1 Recueil de bonnes pratiques

Dans la vraie vie, il est souvent bien difficile de respecter les différentes étapes de cycle de vie d'un logiciel qu'ils soient classiques (cycles linéaires, en spirale,...) ou agiles. Bien souvent, se sont les contraintes de délai qui priment et qui conditionnent grandement le travail. Les documents de spécification et de conception sont peu précis, ambigus voire absents. Le codage est peu documenté. Les tests unitaires sont fait au vol et perdus alors que les tests de non régression sont trop rares lors de la phase d'intégration.

Pour éviter ces écueils et être pragmatique il faut mettre en place un recueil de bonnes pratiques et bien s'organiser tout en restant souple et en s'adaptant aux projets. Car si les gros projets nécessitent une bonne organisation celle-ci peut être lourde à mettre en place pour de plus petits projets. Il est donc préférable de suivre l'esprit plutôt que de respecter à la lettre les plans.

Un recueil de ces bonnes pratiques peut être résumé comme suit :
- Développement de manière itérative
- Développement à base de composants centrés sur l'architecture
- Pilotage par les risques
- Gestion des exigences
- Maîtrise des modifications
- Évaluation continue de la qualité
- Modélisation visuelle

La modélisation visuelle est au cœur de ce cours puisque la suite des chapitres est consacrée à l'étude du langage de modélisation unifié UML.

1.9 Conclusion

Un logiciel n'est pas uniquement composé d'une suite d'instructions mais d'un ensemble d'éléments qui permettent de le spécifier, de le concevoir, de le tester, de l'exploiter et de le faire évoluer.

La construction d'un logiciel est un exercice relativement complexe impliquant de nombreuses parties à la fois humaines, matérielles et technologiques. Dans ce chapitre, nous sommes revenus sur les principales motivations ayant conduit à la naissance de cette spécialité au croisement de l'informatique et des sciences de l'ingénieur.

L'organisation de la construction d'un logiciel est donc primordiale et doit être basée sur un plan d'actions et une organisation méthodique afin d'aboutir à un produit répondant aux objectifs fixés au départ avec le client et ce en respectant les délais et les coûts fixés au préalable.

Dans la suite de ce cours, nous allons étudier l'importance de la modélisation dans la maîtrise de la complexité grandissante des projets informatiques. Nous nous intéresserons plus particulièrement à la modélisation objet à travers l'étude du langage de modélisation unifié UML.

1.10 Exercices

Exercice 1 :

1. Le génie logiciel est une spécialité de l'informatique qui peut être définie comme :
   a) la modélisation orientée objet des besoins du client pour la mise en œuvre de logiciels informatiques
   b) une méthodologie pour la mise en œuvre de projets informatiques
   c) les bonnes pratiques pour programmer des logiciels informatiques
   d) Aucune de ces réponses
2. Le cycle de vie du logiciel s'arrête :
   a) Après les tests unitaires
   b) Lorsque les développeurs terminent l'implémentation
   c) Lorsque le logiciel est conçu
   d) Jamais car il doit être maintenu
3. Lors de la réalisation d'un logiciel important, impliquant peu le client, et où les technologies sont maitrisées, il vaut mieux opter pour le modèle de cycle de vie :
   a) en cascade
   b) en V
   c) en spirale
   d) en Y
4. Les qualités requises pour ce logiciel sont résumées comme suit :
   — facilité de maintenance
   — facilité d'utilisation
   — performance
   — extensibilité
   — réutilisabilité
   Parmi ces qualités quelles sont celles attendues par les utilisateurs et celles attendues par les développeurs.

Exercice 2 :

Soient les tâches suivantes qui constituent un projet et leurs durées : A (3), B (4), C (2), D (3), E (4). Les antériorités sont les suivantes : A enclenche C, A enclenche D, B enclenche E, C enclenche E.

1. Remplir le tableau suivant :

Tâche (pour faire)	Durée	Tâche(s) antérieure(s) (il faut faire)

2. Déterminer le niveau d'exécution de chaque tâche.
3. Calculer les dates de début et de fin au plus tôt et au plus tard pour chaque tâche
4. En déduire le diagramme PERT des potentielles tâches ainsi que le chemin critique

Exercice 3 :

Soit le tableau suivant :

Tâche	A	B	C	D	E	F	G	H	I	J	K	L	M
Durée	1	2	1	3	2	5	2	5	2	1	4	5	4
Antériorité	-	-	A	-	B	E	C,D	-	H	-	I,J	F,G	K,L

1. Déterminer les dates au plus tôt de chacune des tâches
2. Représenter le diagramme Gantt au plus tôt
3. En combien de jours est réalisé le projet ?
4. Quelles sont les tâches critiques ?

Exercice 4 : Compléter les exercices 1.10 et 6.24 par un diagramme de Gantt au plus tôt et un digramme PERT respectivement.

Exercice 5 :

Pour réaliser une mousse au chocolat, il faut suivre les étapes de la recette suivante :

Tout d'abord, faire fondre le chocolat (T1).

Pendant ce temps, il faut battre les blancs d'œufs en neige pendant 5 minutes (T2) après avoir séparé les blancs des jaunes pour chaque œuf (T3).

Ensuite, il faut battre le sucre avec les jaunes d'œufs pendant 3 minutes (T4). Une fois le chocolat fondu,le rajouter au mélange jaune-sucre (T5).

Rajouter délicatement les blancs battus en neige au mélange jaune-sucre-chocolat (T6) puis mettre au frais pendant 4 heures(T7).

1. En suivant la recette, compléter le tableau suivant en déterminant pour chaque tâche sa durée et les tâches qui lui sont antérieures(le(s) tâche(s) avant) :

Tâche	T1	T2	T3	T4	T5	T6	T7
Durée (minutes)	10		2		1	8	
Antériorité			-				

2. Déterminer le niveau d'exécution de chaque tâche.
3. Calculer les dates de début et de fin au plus tôt et au plus tard pour chaque tâche. En déduire la durée totale de réalisation d'une mousse au chocolat.
4. Tracer le diagramme de PERT et en déduire le chemin critique

Chapitre 2

Modélisation en UML

La complexité grandissante des logiciels exige la définition de nouvelles méthodes de travail dont l'objectif consiste à maîtriser cette complexité à la fois fonctionnelle et technique de ces systèmes.

La nécessité de cette maîtrise oblige à la modélisation des propriétés et des comportements des logiciels dès les premières phases de leur cycle de développement et notamment dès les phases d'analyse de besoins et de spécification.

Face à la démocratisation de la programmation orientée objet, les méthodes de modélisation classiques (telle que MERISE) ont rapidement montré certaines limites et de très nombreuses méthodes objet ont vu le jour (Merise 2, OMT, Booch, ...).

Dans ce chapitre, nous allons aborder la notion de modélisation et son importance lors de la mise en œuvre d'un logiciel. Nous nous attarderons plus particulièrement sur la modélisation orientée objet et sur ces principaux concepts. Enfin, nous nous pencherons sur UML, le principal langage de modélisation orientée objet, à travers sa genèse ainsi que son évolution et ses principaux diagrammes.

2.1 Notions de modélisation

La modélisation est une abstraction du monde réel afin d'en garder les aspects que l'on veut étudier mais également d'en maîtriser la complexité (figure 2.1).

Différentes phases dans la mise en œuvre d'un logiciel (de l'analyse des besoins jusqu'à la conception détaillée voire la programmation) nécessitent des modèles.

Les objectifs poursuivis dans une démarche de modélisation sont de permettre une meilleure compréhension de systèmes complexes qu'on peut difficilement comprendre dans leur intégralité. En outre, la modélisation offre un support de communication qui peut s'avérer judicieux s'il est précis et sans ambigüité. Ceci permet de tenir compte des points de vue et de l'expérience de chacun tout en minimisant les interprétations possibles.

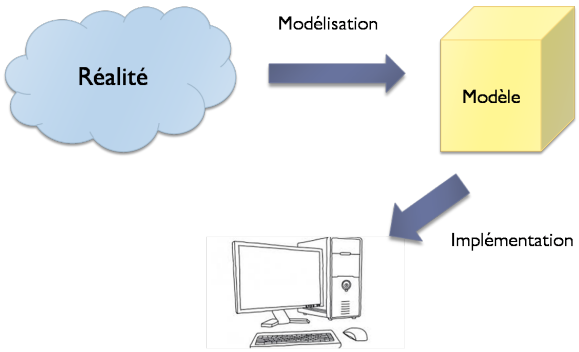


FIGURE 2.1 – Modélisation

2.1.1 Qu'est-ce qu'un modèle ?

Un modèle est une abstraction de la réalité dans le but de mieux appréhender celle-ci. L'objectif étant de faire ressortir les éléments importants tout en éliminant les détails non nécessaires.

Un même sujet d'étude peut avoir plusieurs modèles, chacun donnant un point de vue différent sur le sujet.Un modèle permet ainsi de représenter un sujet d'études (représentativité) s'appliquant à plusieurs cas de ce sujet d'étude (généricité) et incarnant un point de vue sur ces cas (abstraction).

En informatique, un modèle permet de mieux appréhender des programmes complexes ou encore d'explorer des solutions afin de les tester, de les valider voire de montrer à un client ce que sera l'application au préalable.

2.1.2 Approches de modélisation

Il existe différentes approches de modélisation pour décrire un logiciel et son fonctionnement.

Deux principales approches de modèles sont : l'approche fonctionnelle (ou structurée) et l'approche objet. Dans l'approche fonctionnelle, la modélisation est réalisée à partir de fonctions que doit réaliser le système. C'est donc une approche plutôt intuitive. Dans l'approche orientée objet, on identifie les objets manipulés par le système, avec leurs états et leurs comportements.

2.1.2.1 Approche de modélisation fonctionnelle :

L'approche fonctionnelle trouve son origine dans les langages procéduraux (C, Ada, Fortran,...). Elle met en évidence les fonctions à assurer et propose une approche hiérarchique et modulaire qui consiste à découper le problème en blocs indépendants plus facile à traiter (figure 2.2).

L'architecture du système est donc dictée par la réponse au problème (i.e. la fonction du système).

L'approche fonctionnelle dissocie le problème de la représentation des données du problème du traitement de ces données. Ce genre d'approches présente un caractère intuitif fort et

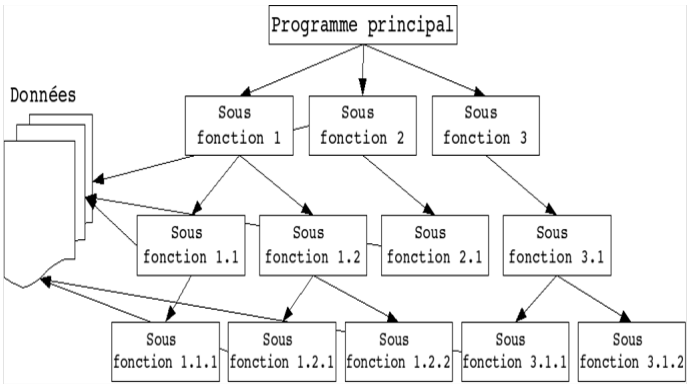


FIGURE 2.2 – Approche fonctionnelle

permet une grande réutilisation des fonctions. Néanmoins, l'approche fonctionnelle n'a pas que des avantages. En cas d'évolution du logiciel, la maintenance devient très vite complexe du fait du chaînage entre les fonctions. Par ailleurs, la séparation entre les données et leurs traitements augmente le risque d'incohérence. Cela peut également engendrer un important effort de maintenance à chaque fois qu'une portion de code utilisera de nouvelles structures de données.

2.1.2.2 Approche de modélisation orientée objet :

Les écueils rencontrés par l'approche fonctionnelle ont fait émerger une nouvelle approche qui considère une entité autonome regroupant un ensemble de propriétés et de traitements associés. Cette entité appelée objet donne donc son nom à cette approche.

Dans l'approche objet, on considère le logiciel comme une collection d'objets dissociés, identifiés et possédant des caractéristiques cohérentes (figure 2.3).

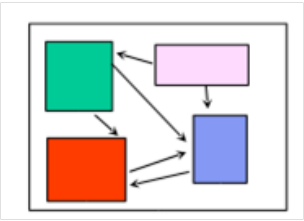


FIGURE 2.3 – De quoi se compose le système ?

Ces caractéristiques sont soient des attributs caractérisant l'état de l'objet, soient des méthodes caractérisant son comportement, c'est-à-dire l'ensemble des actions qu'il peut réaliser.

La fonctionnalité du logiciel émerge alors de l'interaction entre les différents objets qui le constituent.

Le fait que cette approche regroupe les données et leurs traitements associés au sein d'un unique objet induit une forte cohérence entre traitements et données.

En outre, l'approche objet offre l'avantage de faciliter l'évolution d'applications complexes contrairement à l'approche fonctionnelle qui n'est pas adaptée au développement d'applica-

tions qui évoluent sans cesse et dont la complexité croit continuellement. Cependant, l'approche objet reste moins intuitive que l'approche fonctionnelle.

2.1.2.3 Comparatif synthétique entre l'approche fonctionnelle et l'approche objet :

1. Gestion des données :

L'approche objet est une approche orientée donnée car il y a une forte cohérence entre traitements et données qui se trouvent être au même endroit à savoir l'objet.

L'approche structurée privilégie, quant à elle, une organisation des données postérieure à la découverte des fonctions. Le risque d'incohérence entre traitements et données est par conséquent plus élevé.

2. Gestion de l'évolution des besoins :

En approche objet, l'évolution des besoins se présente comme un changement de l'interaction des objets. Cela signifie que s'il y a une modification des données, seul l'objet concerné sera modifié. Toutes les fonctions à modifier sont bien identifiées : ce sont ses méthodes.

Dans une approche fonctionnelle, l'évolution des besoins entraîne souvent une profonde remise en question de la topologie. La décomposition des unités de traitement (du programme principal aux sous-fonctions) est directement dictée par ces besoins. Une modification des données entraîne donc une modification d'un nombre important de fonctions éparpillées et difficiles à identifier dans la hiérarchie de cette décomposition.

2.2 Concepts de base de l'approche orientée objet

L'approche objet repose sur plusieurs concepts dont les principaux sont :

1. L'objet et la classe
2. L'encapsulation
3. L'héritage
4. Le polymorphisme

2.2.1 L'objet et la classe

Un objet est défini par des informations à la fois sur son état (attributs) et sur ses comportements (méthodes). Une classe est un type de données abstrait qui précise des caractéristiques (attributs et méthodes) communes à toute une famille d'objets et qui permet de créer (instancier) des objets possédant ces caractéristiques. Un objet est une instance de classe c'est-à-dire une occurrence d'un type abstrait (figure 2.4).

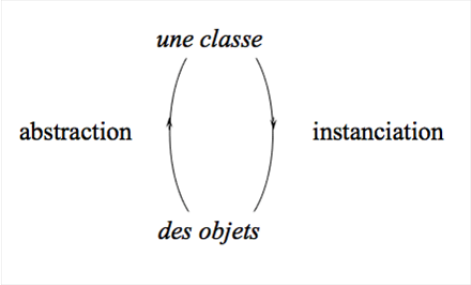


FIGURE 2.4 – Relation entre objet et classe

2.2.2 L'encapsulation

L'encapsulation consiste à masquer les détails d'implémentation d'un objet, en définissant une interface. L'interface est la vue externe d'un objet, elle définit les services accessibles (offerts) aux utilisateurs de l'objet (figure 2.5).

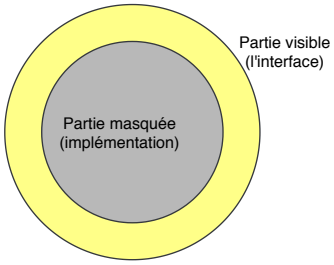


FIGURE 2.5 – Principe d'encapsulation

L'encapsulation garantit l'intégrité des données, car elle permet d'interdire, ou de restreindre, l'accès direct aux attributs des objets. De plus, l'encapsulation facilite l'évolution d'une application car elle stabilise l'utilisation des objets : on peut modifier l'implémentation des attributs d'un objet sans modifier son interface.

2.2.3 L'héritage

L'héritage est un mécanisme de transmission des caractéristiques d'une classe (ses attributs et ses méthodes) vers une sous-classe. L'héritage permet ainsi de définir de nouvelles classes à partir de classes existantes.

L'héritage décrit une relation entre une classe générale (classe de base, classe parent ou super classe) et une classe spécialisée (sous-classe, classe enfant ou classe dérivée).

Une classe peut être spécialisée en d'autres classes, afin d'y ajouter des caractéristiques spécifiques ou d'en adapter certains comportements. Plusieurs classes peuvent être généralisées en une classe qui les factorise, afin de regrouper les caractéristiques communes d'un ensemble de classes.

La spécialisation et la généralisation permettent de construire des hiérarchies de classes.
L'héritage évite la duplication et encourage la réutilisation.

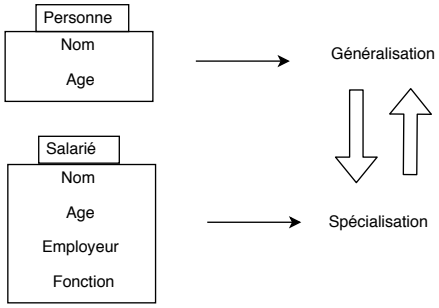


FIGURE 2.6 – Exemple d'héritage

L'héritage peut être simple, si une classe fille n'hérite que d'une seule classe mère, ou multiple, si elle hérite de plusieurs classes mères.

2.2.4 Le polymorphisme

Le polymorphisme représente la faculté d'une même opération à s'exécuter différemment suivant le contexte de la classe où elle se trouve. Le polymorphisme permet ainsi d'augmenter la généricité du code.

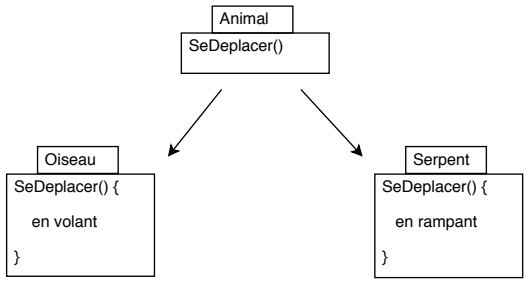


FIGURE 2.7 – Exemple de polymorphisme

2.3 UML, qu'est-ce que c'est ?

Le langage de modélisation unifié, de l'anglais Unified Modeling Language (UML), est un langage de modélisation graphique qui est couramment utilisé en développement logiciel et en conception orientée objet. UML est défini par l'Object Management Group (OMG) [1]

1. L'OMG est un organisme à but non lucratif, créé en 1989 à l'initiative de grandes sociétés (HP, Sun,Unisys, American Airlines, Philips...). Aujourd'hui, l'OMG fédère plus de 850 acteurs du monde informatique. Son rôle est de promouvoir des standards qui garantissent l'interopérabilité entre applications orientées objet, développées sur des réseaux hétérogènes. L'OMG propose

comme un « langage visuel dédié à la spécification, la construction et la documentation des artefacts d'un système logiciel ».

2.3.1 Naissance du standard UML

Dans les années 90, suite à la crise du logiciel et aux limites rencontrées par l'approche fonctionnelle, on dénombrait plus de 50 méthodes objet (OMT, OOD, OOA, OOM, OOSE,...) mais aucune ne s'est réellement imposée. Devant ce foisonnement de méthodes, l'OMG a eu comme objectif de définir une notation standard utilisable par différentes méthodes toutes basées sur l'objet.

Un premier consensus a pu être trouvé autour de trois méthodes (figure 2.8) :

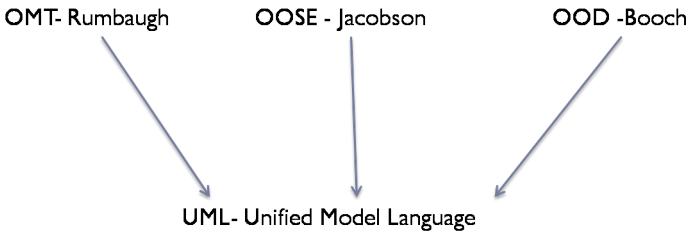


FIGURE 2.8 – Rappel historique d'UML

L'OMT (Object Modelling Technique) , de James Rumbaugh, qui s'appuie sur des vues statiques, dynamiques et fonctionnelles d'un système.

L'approche OOD (Object-Oriented Design) de Grady Booch qui fournit des vues logiques et physiques du système.

'approche OOSE (Object Oriented Software Engineering) d'Ivar Jacobson qui couvre tout le cycle de développement et qui repose sur l'analyse des besoins des utilisateurs.

Issu de l'unification de ces trois méthodes, c'est ainsi qu'est né UML. Les travaux sur ce langage ont continué avec son adoption par de grands acteurs industriels comme HP, Microsoft, Oracle ou encore Unisys. Ce travail a abouti en 1997 à la norme OMG UML 1.1.

UML 2.0 date de 2005, il s'agit d'une version majeure apportant plusieurs innovations et étendant les champs d'application d'UML. Dans la suite de ce polycopié, on notera UML 1 pour les versions UML 1.x et UML 2 pour les versions UML 2.x

2.3.2 UML, un langage, non une méthode

UML est un langage de modélisation au sens de la théorie des langages. Il contient de ce fait les éléments constitutifs de tout langage, à savoir : des concepts, une syntaxe et une sémantique.

notamment l'architecture CORBA (Common Object Request Broker Architecture), un modèle standard pour la construction d'applications à objets distribués (répartis sur un réseau)

De plus, UML s'appuie sur une forme visuelle de notation fondée sur des diagrammes. L'intérêt majeur d'UML est qu'il permet d'exprimer et d'élaborer des modèles objet, indépendamment de tout langage de programmation

UML est un langage qui permet de représenter des modèles, mais il ne définit pas le processus d'élaboration des modèles.

UML n'est en aucun cas une méthode. Une méthode propose un processus, qui régit notamment l'enchaînement des activités d'une entreprise. Or, UML n'a pas été pensé pour faire cela.

UML doit être considéré plutôt comme une sorte de boite à outils qui permet d'améliorer la conception de logiciels complexes tout en s'intégrant à différents processus de développement et ce en toute transparence. En effet, comme UML n'impose aucune méthodologie de travail particulière à suivre, il permet de préserver les approches et les modes de fonctionnement courants.

2.3.3 Avantages d'UML

L'une des principales qualités d'UML est le fait qu'il soit un langage graphique ce qui lui permet d'être relativement facile à comprendre.

En outre, UML offre une certaine universalité qui lui permet d'être utilisables dans de nombreux domaines (transport, télécom, banques,...) et pour différents types de systèmes (Systèmes d'Informations, réseaux Intranets, systèmes embarqués,...).

Enfin, UML est un langage libre ce qui signifie qu'il est du domaine public et que de ce fait tout le monde peut l'utiliser.

2.3.4 Inconvénients d'UML

Certain avantages qu'offre UML sont paradoxalement des défauts. En effet, UML est langage et non une méthode, cela signifie qu'aucune méthodologie ne lui est associée. Il est par conséquent difficile de savoir comment l'utiliser.

Par ailleurs, dû au fait qu'il soit graphique, UML peut manquer parfois de précision et de rigueur qui peuvent rendre difficilement exploitables les diagrammes réalisés tout au long du cycle de vie d'un développement.

2.4 Description du langage UML

2.4.1 Modèle, vues, diagrammes

Lorsqu'il s'agit de décrire UML et ses diagrammes, il important de faire la différence entre les notions de modèles, de vues et de diagrammes.

En effet, comme décrit plus haut dans ce chapitre (§ section 2.1.1), un modèle est une abstraction d'un système composé d'un ensemble d'éléments. Une vue s'apparente à la projection de ce modèle selon une certaine perspective. Une vue omet donc les aspects non pertinents par

rapport à cette perspective. Les vues statiques et dynamiques se manifestent par exemple en UML.

Un diagramme en UML, est une représentation visuelle des différents aspects concrets du modèle (diagrammes de classes, de cas d'utilisation,...)

2.4.2 Conception et UML

UML propose plusieurs diagrammes à l'aide desquels il permet de définir et de visualiser un modèle.

Un diagramme UML est défini comme une représentation graphique qui s'intéresse à un aspect précis du modèle.

Différents diagrammes UML ont été définis. Chaque type de diagrammes possède une structure et véhicule une sémantique précise.

Selon leur utilisation, les diagrammes peuvent être classés selon différents critères comme suit :

1. Descriptifs versus prescriptifs :

Une démarche descriptive vise à utiliser les diagrammes UML pour décrire l'existant.

Une démarche prescriptive permet de décrire le futur système à réaliser à l'aide des diagrammes UML.

2. Diagrammes destinés à différents acteurs :

Généralement, on distingue des digrammes UML destinés aux utilisateurs et d'autres destinés aux concepteurs/développeurs

3. Diagrammes statiques (ou structurels) versus diagrammes dynamiques (ou comportementaux) :

Les diagrammes statiques permettent de décrire les aspects structurels du système.

Les diagrammes dynamiques permettent de décrire les comportements et les interactions entre les différentes briques du système.

Combinés entre eux les différents diagrammes UML offrent une vue assez complète des aspects à la fois statiques et dynamiques d'un système.

2.4.3 Les différents diagrammes UML 2

Les diagrammes UML sont élaborés tout au long du cycle de vie du développement d'un logiciel, depuis le recueil des besoins jusqu'à la phase de déploiement.

Les diagrammes UML permettent de modéliser une application selon plusieurs points de vue.

En UML 2 , on distingue deux principaux points de vue : Le point de vue structurel et le point de vue comportemental (voir figure 2.9).

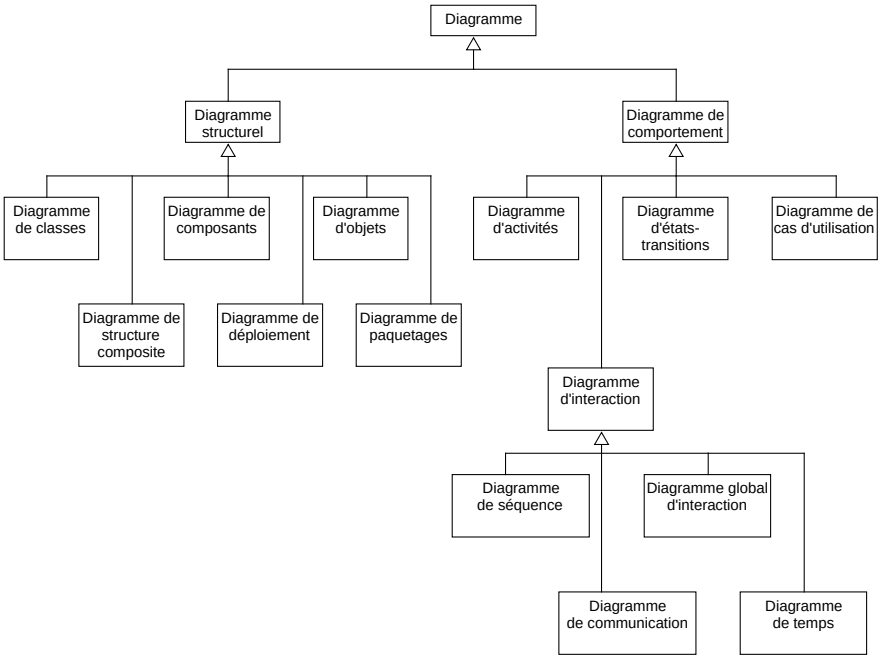


FIGURE 2.9 – Diagrammes UML 2

Pour la partie structurelle, on distingue cinq diagrammes : diagrammes de classes, diagrammes d'objets, diagrammes de composants, diagrammes de structure composite, diagrammes de déploiement et diagrammes de paquetage.

Pour la partie comportement d'un système, cinq types de diagrammes peuvent être utilisés : diagrammes de cas d'utilisation, diagrammes de séquence, diagrammes de communication (appelés diagrammes de collaboration en UML 1), diagrammes d'activités et diagrammes d'états-transitions. A ces cinq classes de diagrammes connus dans les versions UML 1, il faut ajouter les diagrammes de temps et les diagrammes d'interaction globale introduits en UML 2.

En plus de ces diagrammes, qui seuls ne permettent pas de définir toutes les contraintes de spécification requises, UML inclut un langage textuel de contraintes appelé OCL [2]. Il est donc possible d'utiliser le langage OCL en complément et qui s'applique sur les éléments de la plupart des diagrammes.

2.4.4 Notations communes aux diagrammes UML 2

Les diagrammes UML partagent plusieurs notations communes. Nous allons donc introduire quelques éléments de langage d'UML communs à tous les diagrammes. Ces éléments sont :

— les stéréotypes
— les commentaires (ou notes)
— les contraintes
— la relation de dépendance

2. OCL : Object Constraint Language

— les paquetages
— les espaces de nom et les noms qualifiés

Ces éléments permettent de garantir l'intégrité conceptuelle de la notation.

2.4.4.1 Stéréotype :

Un stéréotype est une annotation s'appliquant sur un élément du modèle qui permet de définir une utilisation particulière des éléments de modélisation. Il est représenté par une chaîne de caractères entre guillemets (<<>>) dans l'élément du modèle. Il existe de nombreux stéréotypes prédéfinis parmi lesquels : <<create>>, <<actor>>, <<invariant>>,...

2.4.4.2 Note :

Une note contient une information textuelle comme un commentaire, une valeur marquée ou une contrainte. Graphiquement, elle est représentée par un rectangle dont l'angle supérieur droit est plié (voir figure 2.10).

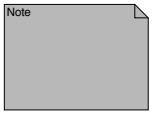


FIGURE 2.10 – Représentation graphique d'une note en UML

Une note est reliée à l'élément qu'elle décrit grâce à une ligne en pointillés.

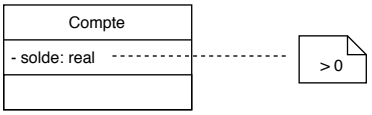


FIGURE 2.11 – Exemple d'utilisation d'une note

2.4.4.3 Contrainte :

Une contrainte est une relation entre éléments de modélisation afin de permettre à des propriétés d'être vérifiées. Les contraintes sont notées entre deux accolades : contrainte

En UML, il existe plusieurs façons d'exprimer les contraintes :

— en langage naturelle. Par exemple : délai de latence inférieur à 100s
— en utilisant des contraintes prédéfinies comme par exemple : ordered, xor, disjoint,...
— à l'aide du langage de contrainte OCL
— ou encore à l'aide d'un langage de programmation

2.4.4.4 Relation de dépendance :

La relation de dépendance définit une relation d'utilisation unidirectionnelle entre deux éléments de modélisation appelés respectivement source et cible de la relation. La relation de dépendance est notée comme suit : [source]− − − − − − − − − − − − − − >[cible]

La relation de dépendance est souvent stéréotypée à l'aide de nombreux stéréotypes, comme <<bind>>, <<realize>>, <<use>>, <<create>>, <<call>>, ... qui sont des stéréotypes prédéfinis.

2.4.4.5 Paquetage :

La notion de paquetage (package en anglais) est un élément d'organisation important dans UML. Graphiquement, un paquetage se représente comme un dossier avec son nom inscrit dedans (voir figure 2.12).

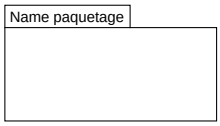


FIGURE 2.12 – Représentation d'un paquetage en UML 2

Un paquetage peut être défini comme un regroupement d'éléments de modèle et de diagrammes. Il permet ainsi d'organiser des éléments de modélisation en groupes fortement cohérents. Chaque paquetage peut contenir un ensemble de diagrammes et/ou de paquetages (décomposition hiérarchique).

Chaque élément d'un paquetage possède un nom unique dans ce paquetage. Il est également possible de définir des relations entre paquetages afin de construire des diagrammes de paquetage définis en UML 2.

2.12).

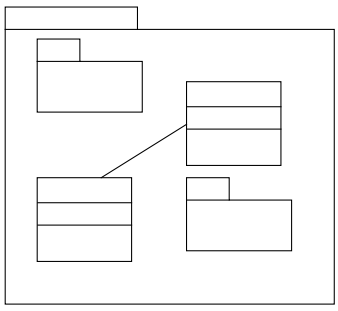


FIGURE 2.13 – Exemple d'utilisation des paquetages

La figure 2.13 illustre un exemple d'utilisation d'un paquetage qui contient des éléments de modélisation et d'autres paquetages.

Espace de nom et nom qualifié :

Dans un diagramme de paquetage, on peut identifier un élément nommé de façon unique par son nom qualifié. Ce dernier est constitué de la série des noms des paquetages et/ou de classes (appelés également des espaces de noms) depuis la racine jusqu'à l'élément en question. Dans un nom qualifié, chaque espace de nom est séparé par deux doubles points (::).

Par exemple, si un paquetage B est inclus dans un paquetage A et contient une classe X, il faut écrire $A :: B :: X$ pour pouvoir utiliser la classe X en dehors du contexte du paquetage B.

2.4.5 Les différentes utilisations d'UML

Il existe trois façons d'utiliser UML [3] :

1. En mode esquisse (méthodes Agile) où les diagrammes sont tracés à la main de façon informelle et incomplète. Dans ce cas, les diagrammes servent davantage de support de communication pour concevoir les parties critiques.
2. en mode plan où les diagrammes sont plus formels et relativement détaillés même si les annotations sont écrites en langue naturelle.
3. enfin, en mode programmation avec les architecture MDA (model driven architecture). Dans ce cas, il s'agit d'une spécification complète et formelle en UML dans le but de générer automatiquement un exécutable à partir des diagrammes.

2.5 Conclusion

La phase de modélisation intervient très tôt dans le processus de mise en œuvre d'un logiciel et constitue un réel défi puisque d'elle dépende le reste des phases en aval du cycle de vie d'un logiciel.

Dans ce chapitre, nous avons abordé la notion de modélisation et plus particulièrement sur la modélisation objet inspirée directement du paradigme de l'orienté objet. Nous avons ainsi introduit les concepts de base de ce paradigme ainsi que le langage unifié qui en découle à savoir UML.

La langage UML est né de la volonté d'unifier les différentes méthodes de modélisation orientée objet même si ce dernier ne donne aucune indication quant à la démarche à suivre. Il n'est qu'un outil, certes puissant, de communication. L'objectif d'UML est de faciliter les interactions entre les différents intervenants lors de la mise en œuvre d'un logiciel tout en facilitant la mise en œuvre des phase de spécification, de conception, d'implémentation et de déploiement.

La bonne modélisation d'un système à l'aide d'UML dépend de la maîtrise des concepts UML et de leur utilisation judicieuse. Il est évident que cette maîtrise est un processus long qui vient avec l'expérience, la répétition des modèles, l'amélioration des modèles déjà réalisés et la remise en cause de soi.

UML 2 ratifié en 2005 est la version qui sera considérée dans la suite de ce cours.

3. selon M. Fowler dans son livre UML 2.0 paru en 2004, édition Campus Press

Dans les prochains chapitres, nous allons nous intéresser à différents diagrammes UML 2 qu'ils soient structurels ou comportementaux.

2.6 Exercices

Exercice 1

1. Pourquoi est né UML ?
2. Que signifie l'acronyme UML ?
3. Quels sont les concepts de base sur lesquels repose la conception orientée objet ?
4. Citer 3 diagrammes structurels et 3 diagrammes de comportement en UML
5. En modélisation orientée objet, quel concept doit-on utiliser pour modéliser la phrase suivante : Un animal omnivore est à la fois un animal herbivore et carnivore.

Exercice 2 : Questions à choix multiple

1. La modélisation orientée objet permet de :
   (a) considérer un logiciel comme une collection de fonctionnalités
   (b) dissocier le problème de la représentation des données et celui de leurs traitements
   (c) faire émerger les fonctionnalités du système à travers les différents objets qui le constituent
   (d) décrire les tâches du projet et leurs interactions
2. Dans la terminologie UML, l'héritage est appelé :
   (a) Une composition
   (b) Une agrégation
   (c) Une généralisation
   (d) Une extension
3. Parmi ces langages, lequel n'est pas orienté objet :
   (a) C++
   (b) Java
   (c) Small talk
   (d) C
4. L'objectif d'UML est de fournir :
   (a) un langage de modélisation selon une approche de Conception Orientée Objet
   (b) une méthodologie de description d'une modélisation orienté objet
   (c) une technique d'analyse pour l'implémentation de gros projets
   (d) aucun de ces réponses

Chapitre 3

Diagramme UML de cas d'utilisation : vue fonctionnelle

Avant de développer un système, il faut savoir précisément à QUOI il devra servir et à quels besoins il devra répondre. Modéliser les besoins permet d'une part de faire l'inventaire des fonctionnalités attendues du point de vue des différents utilisateurs du système. Cela permet également d'organiser les besoins entre eux, de manière à faire apparaître des relations (réutilisations possibles, ...). C'est donc la base de toute démarche de mise en œuvre du cycle de développement d'un système.

UML offre la possibilité de recourir à une modélisation sous une forme graphique, suffisamment simple pour être compréhensible par toutes les personnes impliquées dans le projet (surtout les non informaticiens).

Avec UML, on modélise les besoins au moyen de diagrammes de cas d'utilisation (Use case diagrams. Le diagramme use case permet de recueillir, d'analyser et d'organiser les besoins. Le diagramme use case est donc le premier à construire, avec lui débute l'étape d'analyse du système à mettre en œuvre.

3.1 Qu'est-ce qu'un diagramme de cas d'utilisation ?

Un cas d'utilisation décrit sous la forme d'actions et de réactions, le comportement d'un système du point de vue de ses utilisateurs et plus généralement de l'environnement extérieur.

Les diagrammes de cas d'utilisation permettent de définir les limites d'un système et les relations entre ce système et son environnement.

Ainsi, si on se place du point de vue des différents protagonistes ; pour un client, les diagrammes de cas d'utilisation permettent de décrire ses besoins. Il pourra parvenir à un accord (contrat) avec les développeurs plus rapidement, en évitant certains écueils dus à des incohérences ou des ambigüités (véhiculées lors d'échanges verbaux ou dans des rapports).

Pour les informaticiens impliqués dans la mise en œuvre d'un projet, les diagrammes de cas d'utilisation sont un point d'entrée pour les étapes suivantes du développement d'un logiciel.

En UML, la mise œuvre d'un diagramme de cas d'utilisation passe par les étapes suivantes :

— L'identification du cadre du système à modéliser (environnement extérieur)
— L'identification des acteurs
— L'identification des cas d'utilisation
— L'identification des relations entre ces différentes entités

Une fois ces étapes suivies et terminées on obtient un diagramme de cas d'utilisation de la forme générale illustrée par la figure 3.1.

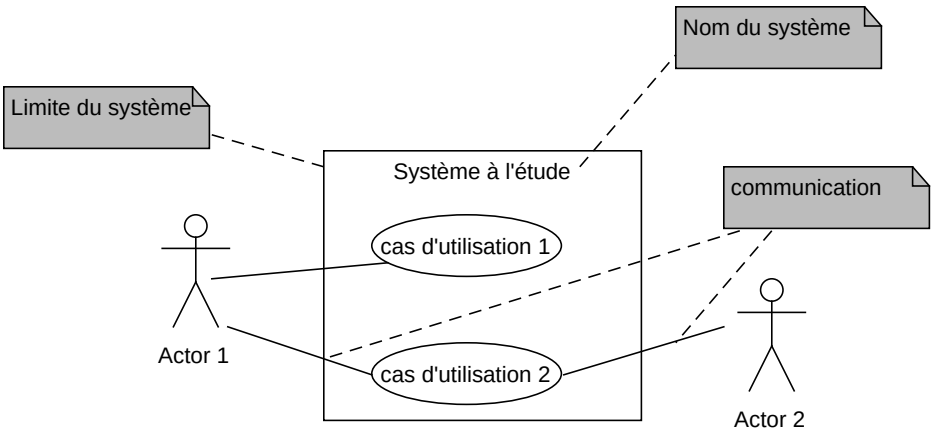


FIGURE 3.1 – Structure générale d'un diagramme UML de cas d'utilisation

La figure 3.1 montre que :

— Le système à modéliser apparaît dans un cadre : cela permet de séparer le système à modéliser de son environnement (monde extérieur).
— Il y a des entités extérieures au système et qui interagissent avec lui, appelées «acteur»
— Les grandes fonctionnalités du système (les cas d'utilisation) sont représentées par des ellipses.

Dans la suite, nous allons détailler les éléments de base constituant le diagramme de cas d'utilisation à savoir les acteurs, les cas d'utilisation et les différentes relations présentes dans ce diagramme.

3.2 Les acteurs

Dans le diagramme UML de cas d'utilisation, on représente toute entité qui interagit avec le système et qui à l'extérieur du système comme acteur.

ATTENTION : 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.

Les principaux acteurs sont les utilisateurs du système.

En plus des utilisateurs, les acteurs peuvent être :

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

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

3.2.1 Représentation d'un acteur

UML donne la possibilité de représenter les acteurs de deux manières. Un acteur se représente donc soit par un petit bonhomme avec son nom (i.e. son rôle) inscrit dessous. Il est également possible de représenter un acteur sous la forme d'un classeur stéréotypé << actor >> (figure 3.2).

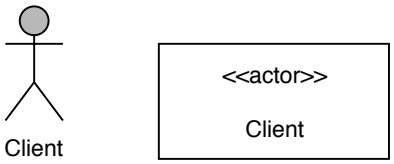


FIGURE 3.2 – Les différentes représentations d'un acteur

3.2.2 Types d'acteurs

On distingue deux types d'acteurs : les acteurs principaux et les acteurs secondaires.

Les acteurs principaux sont ceux qui sollicitent le système et qui sont à l'initiative des échanges avec le système. A l'inverse, lorsqu'ils sont sollicités par le système, ils sont dits acteurs secondaires

Le stéréotype << primary >> vient orner l'association reliant un cas d'utilisation à son acteur principal, le stéréotype << secondary >> est utilisé pour les acteurs secondaires.

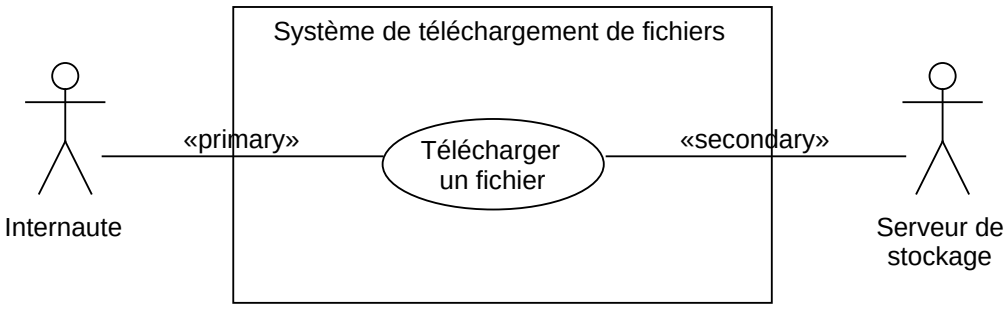


FIGURE 3.3 – Exemple d'acteurs principal et secondaire

3.2.3 Identification des acteurs

La tâche d'identifier les acteurs dans un diagramme de cas d'utilisation est étroitement liée à celle qui consiste à définir les limites du système. En effet, savoir ce qui est inclus ou pas dans le système est la première chose à faire avant de rechercher les acteurs et leurs besoins.

En procédant ainsi, tout ce qui est à l'extérieur du système et qui interagit avec lui est un acteur et tout ce qui est à l'intérieur est une fonctionnalité à réaliser.

Par ailleurs, lorsqu'on veut identifier les acteurs dans un diagramme de cas d'utilisation, il est très important de vérifiez que les acteurs communiquent bien directement avec le système par émission ou réception de messages. Par exemple, l'hôtesse de caisse d'un magasin de grande distribution est un acteur principal pour la caisse enregistreuse, par contre, les clients du magasin n'interagissent pas directement avec la caisse, ils seront tout au plus un acteur secondaire.

3.2.4 Relation entre acteurs

La seule relation possible entre deux acteurs est celle de la généralisation. Cette relation possède exactement le même comportement et la même représentation graphique que dans différents diagrammes UML.

Cette relation doit être identifiée et son utilisation encouragée car elle permet de factoriser certaines relations, ce qui permet de clarifier et de rendre plus lisible un diagramme de cas d'utilisation.

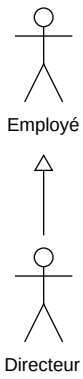


FIGURE 3.4 – Exemple de relation d'héritage entre acteurs

3.3 Cas d'utilisation

Un cas d'utilisation est une unité cohérente représentant une fonctionnalité visible de l'extérieur. Il réalise un service de bout-en-bout pour l'acteur qui l'initie avec :

— un déclenchement,
— un déroulement

— une fin

Un cas d'utilisation est qualifié d'interne quand il n'est pas directement relié à un acteur.

3.3.1 Représentation d'un cas d'utilisation

Un cas d'utilisation se représente par une ellipse contenant le nom du cas et éventuellement un stéréotype (figure 3.5).

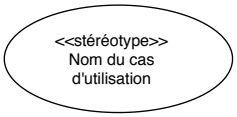


FIGURE 3.5 – Représentation générale d'un cas d'utilisation

Les cas d'utilisation sont nommés avec un verbe à l'infinitif suivi d'un complément. Le choix du verbe doit être effectué on se plaçant du point de vue de l'acteur et non de celui du système. Par exemple, un distributeur de billets aura probablement un cas d'utilisation «Retirer de l'argent» et non pas «Distribuer de l'argent».

3.3.2 Identification des cas d'utilisation

Pour identifier les cas d'utilisation, il faut se placer du point de vue de chaque acteur et déterminer comment et surtout pourquoi il se sert du système. Chaque cas d'utilisation correspond donc à une fonction métier du système, selon le point de vue d'un de ses acteurs.

L'ensemble des cas d'utilisation doit décrire plus ou moins exhaustivement les exigences fonctionnelles du système.

Néanmoins, trouver le bon niveau de détail pour les cas d'utilisation est un problème difficile qui nécessite de l'expérience.

3.3.3 Relation entre les cas d'utilisation

Les relations entre cas d'utilisation permettent de compléter le diagramme par d'autres cas d'utilisation non liés à des acteurs principaux mais à d'autres cas d'utilisation qui apporteront plus de précision au diagramme.

Il existe principalement deux types de relations :

— Les relations de dépendances stéréotypées, qui sont explicitées par un stéréotype (les plus utilisés sont l'inclusion et l'extension) ;
— La généralisation/spécialisation.

3.3.3.1 Relation de dépendance stéréotypée <<include >>

La relation d'inclusion, notée <<include>>, est une relation de dépendance qui sert à enrichir un cas d'utilisation par un autre cas d'utilisation.

Elle peut être définie comme suit : Un cas A inclut un cas B si le comportement décrit par le cas A inclut le comportement du cas B : le cas A dépend de B. Lorsque A est sollicité, B l'est obligatoirement, comme une partie de A.

Cette dépendance est symbolisée par le stéréotype <<include >>.

Dans un diagramme de cas d'utilisation, cette relation est représentée par une flèche en pointillée reliant deux cas d'utilisation et munie du stéréotype <<include >>. La relation de dépendance permet :

— La factorisation : Les inclusions permettent de factoriser une partie de la description d'un cas d'utilisation qui serait commune à d'autres cas d'utilisation (voir l'exemple de la figure 3.6).
— La décomposition : Les inclusions permettent également de décomposer un cas complexe en sous-cas plus simples (voir l'exemple de la figure 3.7).

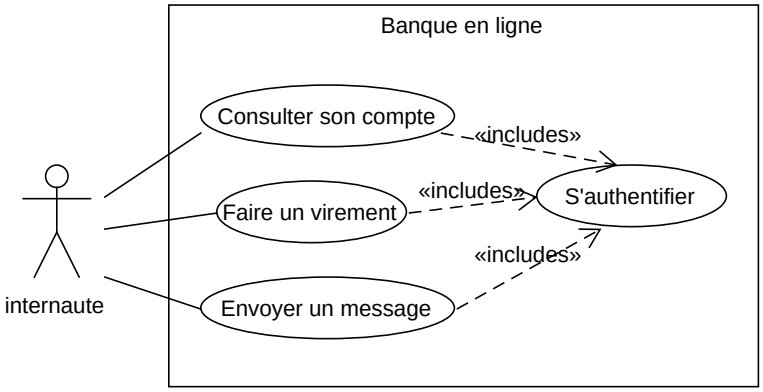


FIGURE 3.6 – Exemple de factorisation à l'aide de la relation <<include>>

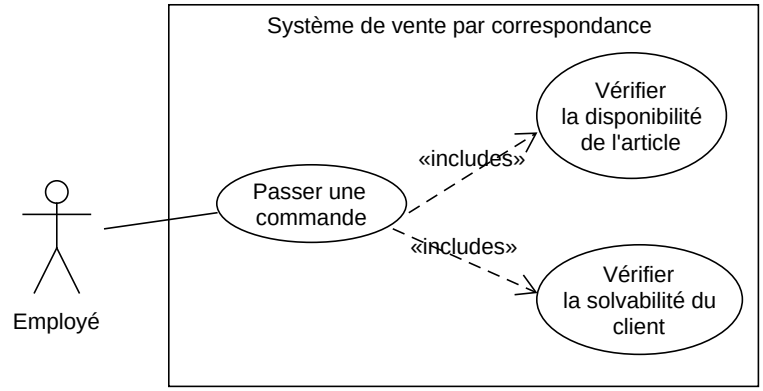


FIGURE 3.7 – Exemple de décomposition à l'aide de la relation <<include>>

3.3.3.2 Relation de dépendance stéréotypée <<extend >>

La relation d'extension, symbolisée par le stéréotype <<extend>>, peut être définie comme suit :

On dit qu'un cas d'utilisation A étend un cas d'utilisation B lorsque le cas d'utilisation A peut être appelé au cours de l'exécution du cas d'utilisation B. Exécuter B peut éventuellement entraîner l'exécution de A. Contrairement à l'inclusion, l'extension est optionnelle.

Tout comme l'inclusion, la relation d'extension est représentée par une flèche en pointillée reliant les 2 cas d'utilisation et munie du stéréotype <<extend>> (figure 3.8).

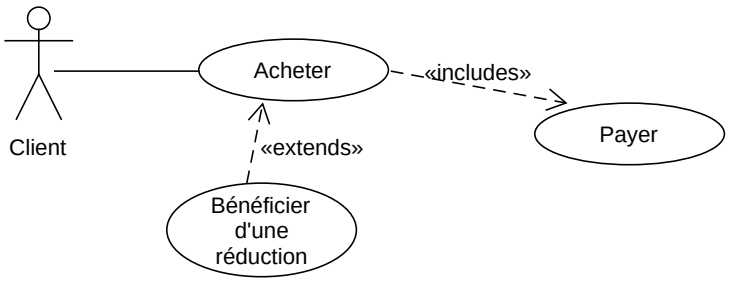


FIGURE 3.8 – Exemple d'extension entre cas d'utilisation

Dans l'exemple illustrée par la figure 3.8, un client qui achète est obligé de payer cette obligation est représentée par la relation de dépendance stéréotypée <<include>>. Ce client peut , éventuellement, bénéficier d'une réduction (c'est optionnel) d'où l'utilisation de la relation de dépendance <<extend>>.

Le point d'extension :

L'extension peut intervenir à un point précis du cas étendu c'est le point d'extension. Il porte un nom, qui figure dans un compartiment du cas étendu sous la rubrique point d'extension, et est éventuellement associé à une contrainte indiquant le moment où l'extension intervient. Une extension est souvent soumise à condition. Graphiquement, la condition est exprimée sous la forme d'une note.

La figure 3.9 illustre l'exemple d'une banque où la vérification du solde du compte lors d'un virement n'intervient que si la demande de virement dépasse 20.

3.3.4 Relation de généralisation

La relation de généralisation/spécialisation est également présente entre deux cas d'utilisation et se traduit par le concept d'héritage dans les langages orientés objet. Un cas A est une généralisation d'un cas B si B est un cas particulier de A. Cela signifie, comme pour les classes, que B va hériter du comportement de A et le spécialiser.

La relation de généralisation est représentée par une flèche avec une extrémité triangulaire.

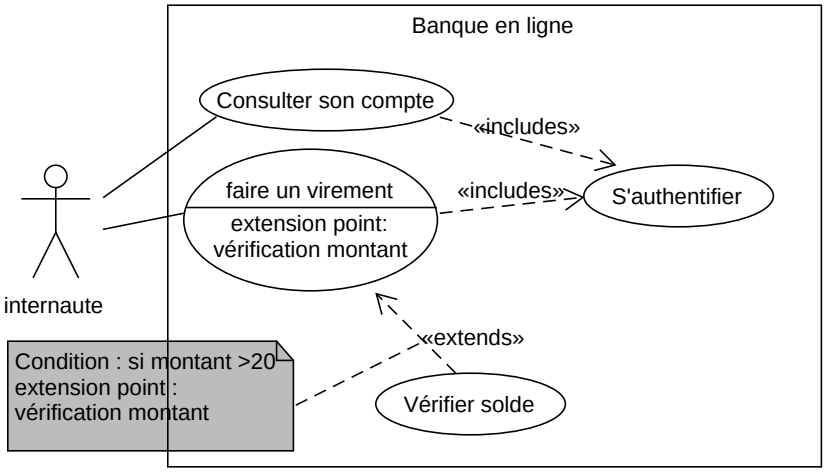


FIGURE 3.9 – Exemple de l'utilisation d'un point d'extension

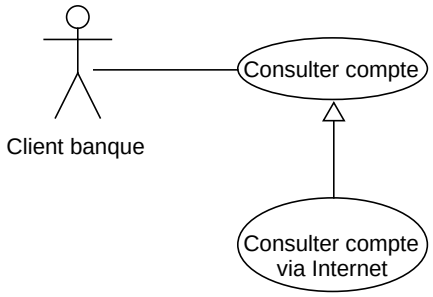


FIGURE 3.10 – Exemple d'héritage entre cas d'utilisation

3.4 Relation entre cas d'utilisation et acteurs

3.4.1 Association :

Une relation d'association est le chemin de communication entre un acteur et un cas d'utilisation et est représentée par un trait continu entre ces deux entités.

3.4.2 Multiplicité :

Lorsqu'un acteur peut interagir plusieurs fois avec un cas d'utilisation, il est possible d'ajouter une multiplicité sur l'association du côté du cas d'utilisation. Le symbole * signifie plusieurs, exactement n s'écrit tout simplement n, n..m signifie entre n et m, etc.

La notion de multiplicité n'est pas propre au diagramme de cas d'utilisation. Nous en reparlerons dans le chapitre 4 consacré au diagramme de classes.

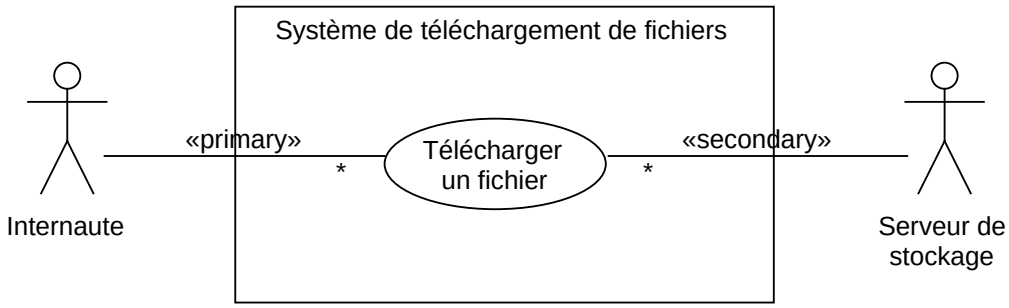


FIGURE 3.11 – Exemple d'utilisation des multiplicités dans un diagramme de cas d'utilisation

3.5 Cas d'étude (inspiré du livre UML 2 par la pratique de P. Roques)

Nous allons à présent consolider les concepts appris sur le diagramme de cas d'utilisation à travers l'étude d'un cas pratique qui s'énonce comme suite :

On considère le système suivant de gestion d'un Guichet Automatique Bancaire (GAB) :
Le GAB délivre de l'argent à tout porteur de carte (carte Visa ou carte de la banque)
Pour les clients de la banque, il permet en plus :
- la consultation du solde du compte
- le dépôt d'argent (chèque ou numéraire)

Toute transaction est sécurisée et nécessite par conséquent une authentification.
Pour les clients de la banque, une interaction avec le système d'information de cette dernière est nécessaire pour effectuer les opérations de consultation du solde du compte et de dépôt d'argent.
Dans le cas où une carte est avalée par le GAB, un opérateur de maintenance se charge de la récupérer.
C'est la même personne qui collecte également les dépôts d'argent et qui recharge le GAB.

Pour réaliser le diagramme de cas d'utilisation de ce système, il est important de :
- Bien identifier le système
- Identifier les acteurs (principaux et secondaires)
- Identifier les cas d'utilisation
- Enfin, réaliser le diagramme de cas d'utilisation

Identification du système à modéliser :

Il s'agit dans cette étape de bien cerner le système qu'on cherche à modéliser. Ici, il s'agit du guichet automatique bancaire.

Identification des acteurs :

Pour identifier les acteurs, il faut répondre à la question suivante : Quelles sont les entités externes qui interagissent directement avec le GAB ?

Phrase 1 : le GAB délivre de l'argent à tout porteur de carte (carte Visa ou carte de la banque)

Phrase 2 : pour les clients de la banque , il permet en plus :

- la consultation du solde du compte

- le dépôt d'argent (chèque ou numéraire)

Phrase 5 : Il est nécessaire d'interagir avec la banque (à travers son système d'information) pour effectuer ces deux opérations.

Phrase 3 : Toute transaction est sécurisée et nécessite par conséquent une authentification par un système d'autorisation

Phrase 4 : Dans le cas où une carte est avalée par le GAB, un opérateur de maintenance se charge de la récupérer.

Phrase 6 : C'est la même personne qui collecte également les dépôts d'argent et qui recharge le GAB.

Les acteurs du système sont donc :
- le porteur de carte
- le client de la banque
- le système d'information
- le système d'autorisation
- l'opérateur de maintenance

Identification des cas d'utilisation :

Afin d'identifier les cas d'utilisation du diagramme, il est nécessaire d'analyser un à un les 5 acteurs et lister les différentes interactions avec le GAB.

1. Porteur de carte :
   - Retirer de l'argent
2. Client banque :
   - Retirer argent
   - Consulter compte
   - Déposer argent
3. Système d'autorisation CB
   - Le GAB fait appel à lui pour le retrait d'argent
4. Système d'information banque
   - Le GAB fait appel à lui pour les opérations de consultation de compte et de dépôt d'argent
5. Opérateur de maintenance
   - Récupérer les cartes avalées
   - Récupérer les dépôts d'argent
   - Recharger le distributeur

On remarque qu'il y a des acteurs qui sont à l'origine des interactions avec le GAB à savoir *leporteurdecarte*, *leclientdelabanque* et *l'opérateur de maintenance*.

Ce sont les acteurs principaux.

Les acteurs *système d'information de la banque* et *système d'autorisation des cartes bancaires* sont, quant à eux, sollicités par le GAB. Ce sont donc des acteurs secondaires car ils n'utilisent pas le GAB, c'est le GAB qui fait appel à eux pour réaliser certains cas d'utilisation.

Réalisation du diagramme de cas d'utilisation :

Identifier les acteurs du système et les principaux cas d'utilisation, permet de commencer à tracer les grandes lignes du diagramme de cas d'utilisation inhérent au guichet automatique bancaire.

Au fur et à mesure, des détails sont ajoutés afin d'apporter plus d'éclaircissement et des simplifications opérées afin de garder lisible le diagramme. Ce processus est donc itératif. Le diagramme de cas d'utilisation obtenu est illustré par la figure 3.12.

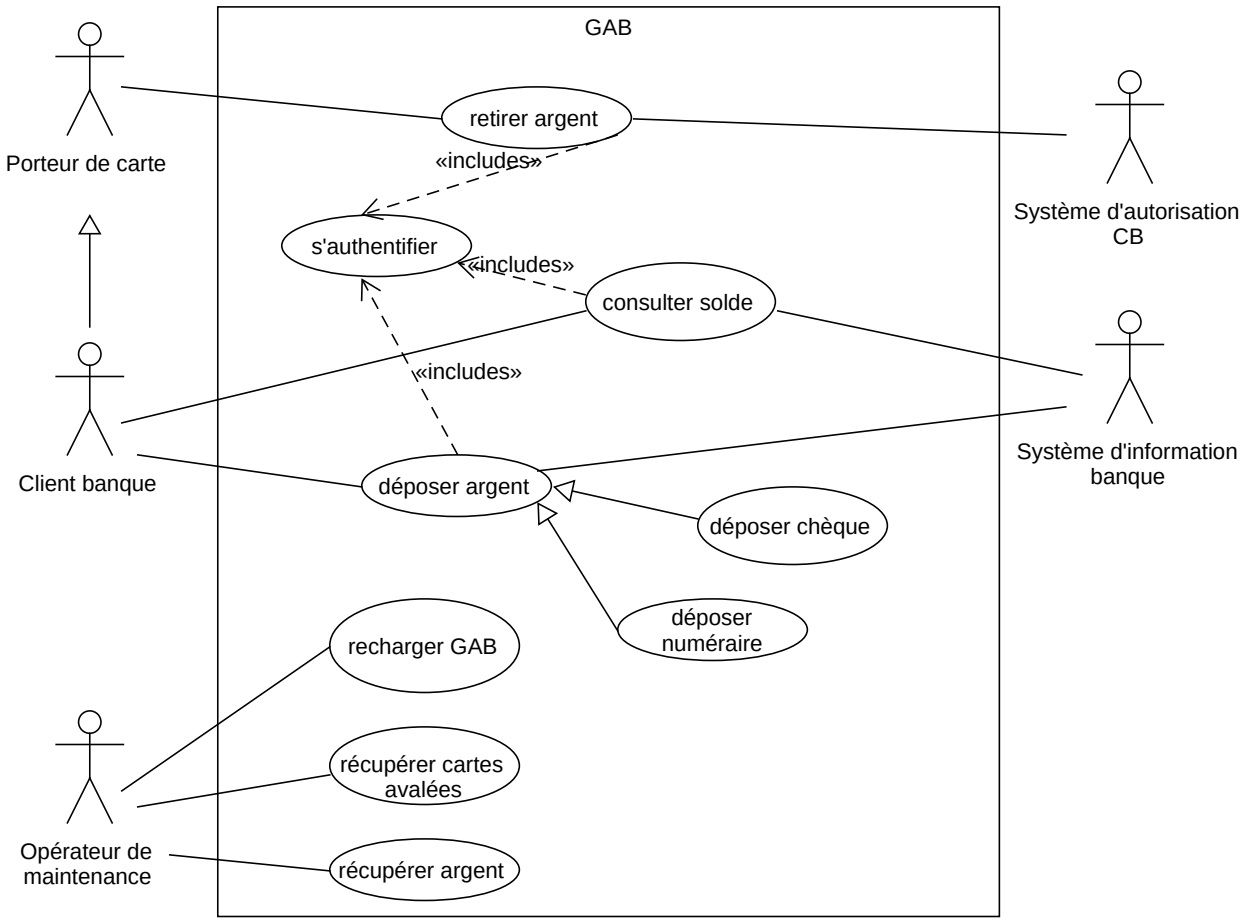


FIGURE 3.12 – Diagramme de cas d'utilisation

3.6 Éléments de méthodologie

La première étape de l'analyse consiste à bien comprendre le système à étudier. Il s'agit de lire attentivement le cahier des charges s'il existe ou à défaut à beaucoup s'entretenir avec le client qui souhaite réaliser le logiciel. Cette démarche doit permettre de délimiter le système à étudier et ainsi de retrouver les acteurs qui interagissent avec lui. Il est très important de fixer des frontières au problème. Ensuite, il faut rechercher les fonctionnalités du système par la définition de ses «cas d'utilisation».

Ces différentes étapes peuvent être résumées comme suit :
- Délimiter le cadre du système à modéliser
- Identifier les acteurs qui utilisent, gèrent et exécutent des fonctionnalités spécifiques
- Organiser les acteurs par relation de généralisation/spécialisation si c'est pertinent
- Pour chaque acteur, rechercher les cas d'utilisation du système
- Apporter des éléments des détails afin de mieux spécifier les cas d'utilisation en utilisant des cas d'utilisation internes reliés par les différentes relations étudiées au cours de ce chapitre (relation d'inclusion, relation d'extension, de généralisation/spécialisation).

Description textuelle d'un cas d'utilisation :

Il est possible d'utiliser une description textuelle d'un cas d'utilisation. UML ne définit cependant pas de représentation type.

Les travaux de Alistair Cockburn[1] peuvent être utilisés comme une référence sur ce sujet et décrivent un cas d'utilisation à travers les points suivants :
- objectif
- acteur(s) concerné(s)
- pré condition(s) (si certaines conditions particulières sont requises avant l'exécution du cas)
- post condition(s) (si certaines conditions particulières sont requises après l'exécution du cas)
- scénario nominal (le scénario principal sans incident et qui aboutit au résultat attendu)
- scénario alternatifs (les autres scénarios secondaires ou qui correspondent à des anomalies)

3.7 Conclusion

Dans ce chapitre, nous avons décrit le diagramme de cas d'utilisation. Nous avons choisi parmi tous les diagrammes proposés en UML 2 de commencer par ce diagramme car il représente la première étape lors d'une démarche de mise en œuvre d'un logiciel. En effet, il est très important, avant de se lancer dans la réalisation d'un logiciel de comprendre les besoins et les attentes du client.

Le diagramme de cas d'utilisation permet de modéliser les besoins du client et apporte une vision «utilisateur» du système. Les fonctionnalités qu'offre le système sont identifiées ainsi que ses limites.

1. COCKBURN A., Rédiger des cas d'utilisation efficaces, Eyrolles, 2001.

Le diagramme de cas d'utilisation permet également de modéliser les différentes interactions qui existent entre le système et ses utilisateurs mais également entre les fonctionnalités elles-mêmes.

Dans le chapitre suivant, nous aborderons la notion de vue structurelle du système à travers les diagrammes de classes et d'objets.

3.8 Exercice

Exercice 1 :

Soit le diagramme de cas d'utilisation illustré dans la figure 3.13, modélisant le fonctionnement d'un logiciel de questions-réponses pour que des étudiants puissent tester leurs connaissances en informatique.

Analysez ce diagramme en vérifiant qu'il traduit bien le texte ci-dessous et indiquez s'il y a des problèmes et comment les corriger :

« Un étudiant peut travailler sur une thématique seulement s'il s'est authentifié avant. Travailler sur une thématique se traduit par répondre à des questions ou consulter des cours. Après son travail, un étudiant peut, s'il le veut, imprimer un compte-rendu.

Un enseignant peut créer une thématique. Il y a deux manières de créer une thématique, soit en téléchargeant un texte déjà rédigé, soit de manière interactive.»

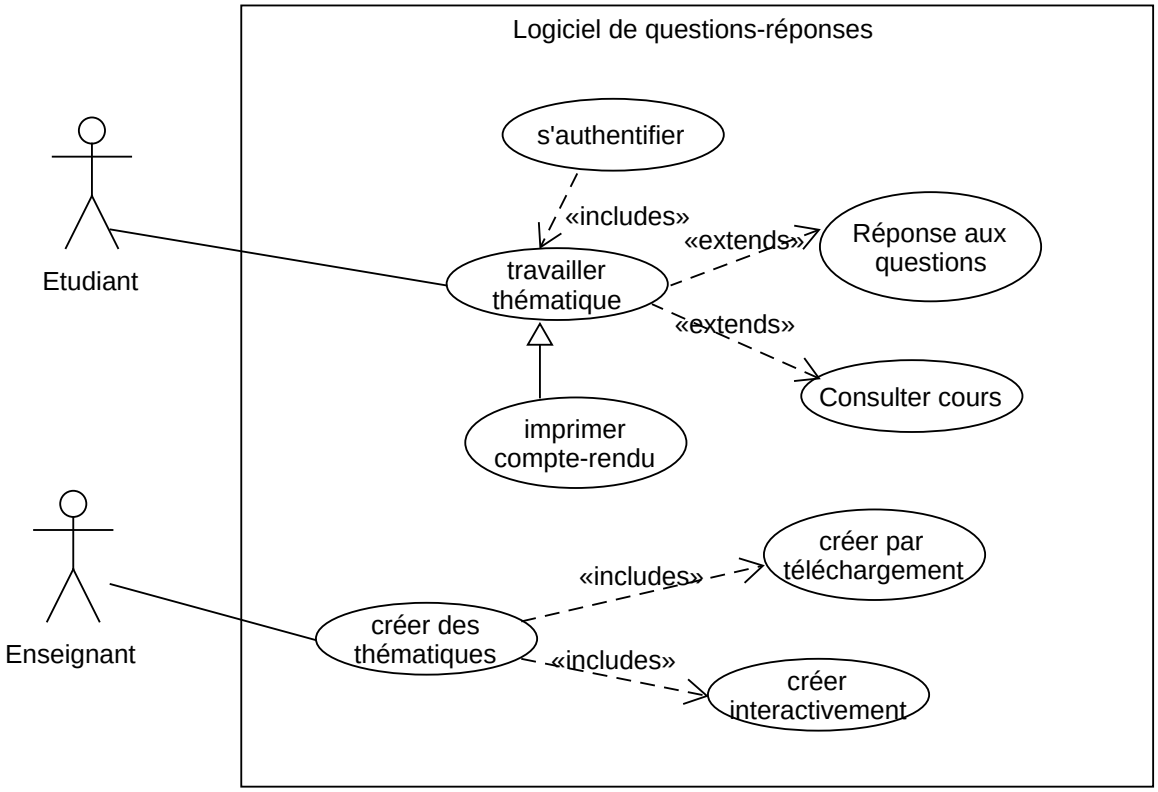


FIGURE 3.13 – Diagramme de cas d'utilisation

Exercice 2 :

Soit un système simplifié de caisse enregistreuse de supermarché. Le déroulement normal d'utilisation de la caisse est le suivant :

Un client arrive à la caisse avec des articles à payer. Le caissier enregistre le numéro d'identification de chaque article (par exemple le code bar), ainsi que la quantité si celle-ci est supérieure à 1. La caisse affiche le prix de chaque article et son libellé. Lorsque tous les achats sont enregistrés, le caissier signale la fin de la vente au niveau de la caisse. La caisse affiche le total des achats et le client est invité par le caissier à payer. Le client choisit alors parmi les trois modes de paiement pris en charge par la caisse :

- S'il paye en numéraire (argent liquide) : le caissier encaisse l'argent reçu, la caisse indique la monnaie à rendre au client
- S'il paye par chèque : le caissier vérifie la solvabilité du client en transmettant une requête à un centre d'autorisation via la caisse
- S'il paye par carte bancaire : un terminal bancaire fait partie de la caisse. Il transmet une demande d'autorisation à un centre d'autorisation en fonction du type de la carte

Lorsque le paiement est terminé, la caisse enregistre la vente et le caissier donne un ticket de caisse au client.

Après la saisie des articles, le client peut présenter au caissier des coupons de réduction pour certains articles. Ceux-ci seront alors pris en compte une fois que le caissier aura signalé la fin de la vente. Lorsque le paiement est terminé et que la vente a été enregistrée, la caisse transmet les informations sur le nombre d'articles vendus au système de gestion des stocks.

Tous les matins, le responsable du magasin initialise les caisses pour la journée.

1. Modéliser le fonctionnement de la caisse par un diagramme de cas d'utilisation

Chapitre 4

Diagrammes de classes et diagrammes d'objets : vue structurelle

Le diagramme de classe est le point central dans une modélisation orientée objet. Alors que le diagramme de cas d'utilisation montre un système du point de vue des acteurs, le diagramme de classes en montre la structure interne. Le diagramme de classe a pour objectif de décrire la structure des entités manipulées par les utilisateurs.

Dans ce chapitre, nous allons examiner de plus près les principaux concepts sur lesquels reposent les diagrammes de classes et d'objets à savoir :

— le concept de classe et d'objet
— les attributs et les méthodes des classes
— les différentes relations entre classes

4.1 Importance du diagramme de classe dans le processus de développement d'un logiciel

Le diagramme de classe est un diagramme statique qui offre une vision structurelle du système car on ne tient pas compte du facteur temporel pour décrire le comportement du système. Il est de ce fait à la base de toute démarche de modélisation orientée objet. Le diagramme de classe est d'ailleurs le seul qui soit obligatoire lors de la modélisation orientée objet d'un système.

En effet, dans une démarche de modélisation objet, on commence à construire le système autour de la notion d'objet plutôt qu'autour de la notion de fonctionnalité. L'approche orientée objet, comme nous l'avons vu dans le chapitre 2, ne distingue pas les fonctionnalités du système de ces données. Ainsi chaque classe, qui représente un aspect structurel du système sera pourvue de données et de traitements. En outre, le diagramme de classe mais également une évidence les relations qui existent entre les différentes classes.

Lors des étapes d'analyse de besoins et de conception, le diagramme de classe permet de définir d'une façon précise les objets qui composent le système et qui apparaissent dans les diagrammes de cas d'utilisation et d'interaction (cf. chapitres 3 et 5). Ces objets sont connus

également sous le nom d'objets du domaine. Lors de la phase d'implémentation les classes définies se transforment en objets logiciels.

Ainsi, les diagrammes de classes s'étoffent, s'enrichissent et parfois se transforment au fur et à mesure que l'on va de hauts niveaux à de bas niveaux d'abstraction i.e. de la spécification à l'implémentation.

4.2 Concepts de classe et d'objet

Une classe représente la description abstraite (principe d'abstraction) d'un ensemble d'objets ayant une sémantique et des caractéristiques communes (c'est comme un type).

Le nom de la classe doit évoquer le concept décrit par la classe. En UML, il commence par une majuscule.

Un objet est une instance (une occurrence) de classe c'est-à-dire une concrétisation du concept abstrait. Par exemple l'objet leila est une instance de la classe Personne.

4.2.1 Représentation d'une classe en UML :

Une classe est représentée par un rectangle (appelé également classeur) divisé en plusieurs compartiments :

Le premier compartiment contient le nom de la classe.

Le deuxième compartiment contient les attributs.

Le troisième compartiment contient les méthodes.

D'autres compartiments peuvent être ajoutés : responsabilités, exceptions, etc.

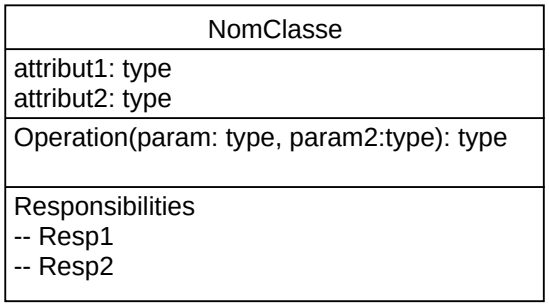


FIGURE 4.1 – Exemple de classe avec quatre compartiments

Dans certains cas, lorsque la modélisation s'intéresse davantage aux relations entre les classes qu'à leurs contenus eux-mêmes ou que ces informations ne sont pas forcément toutes connues (selon l'avancement de la modélisation), les compartiments 2 et 3 pourront restés vides car seul le nom de la classe est obligatoire (figure 4.2).

Certaines règles sont à respecter lors de la représentation d'une classe parmi lesquelles :

- Le nom de la classe commence par une majuscule

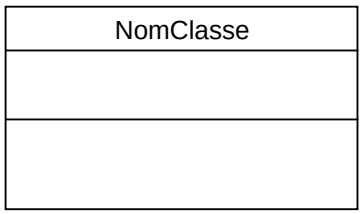


FIGURE 4.2 – Représentation simplifiée d'une classe

- Le nom d'une classe est unique au sein d'un package.
- Le nom d'une propriété commence par une minuscule
- Les types de base (integer, real, string, boolean,...) sont en minuscules
- Il n'y a pas d'espace dans les noms de classes ou de propriétés

4.3 Caractéristiques d'une classe

Les caractéristiques d'une classe permettent de spécifier l'état et le comportement des objets. L'état d'un objet est décrit à travers ses attributs. Le comportement d'un objet est décrit à travers ses méthodes (ou opérations).

4.3.1 Attribut

Un attribut est une propriété qui représente un type d'information contenu dans une classe. Par exemple, le nombre de page, l'auteur, la date de parution... peuvent constituer des attributs d'une classe Livre. Chacune de ces informations est définie par un nom, un type, une visibilité (décrite en section 4.3.3) et peut être initialisée selon la syntaxe suivante :

visibilite nom_attribut : *type_attribut*= *valeur_initiale*.

Il est à noter que le nom de l'attribut doit être unique dans la classe.

En plus de ces informations, un attribut peut avoir des caractéristiques particulières qui le différencieront des autres attributs. On distingue alors les attributs de classe et les attributs dérivés

Attribut de classe :

Par défaut, chaque instance d'une classe possède sa propre copie des attributs de la classe. Les valeurs des attributs peuvent donc différer d'un objet à un autre. Cependant, il est parfois nécessaire de définir un attribut de classe qui garde une valeur unique et partagée par toutes les instances de la classe (*static* en Java ou en C++). Les instances ont accès à cet attribut, mais n'en possèdent pas une copie. Un attribut de classe n'est donc pas une propriété d'une instance, mais une propriété de la classe et l'accès à cet attribut ne nécessite pas l'existence d'une instance (il n'est donc pas instancié avec l'objet). Graphiquement, un attribut de classe est souligné (voir figure 4.3).

Attribut dérivé :

Les attributs dérivés sont des attributs qui peuvent être calculés au travers d'une fonction utilisant d'autres attributs de la classe. Les noms des attributs dérivés sont précédés d'un « / ».

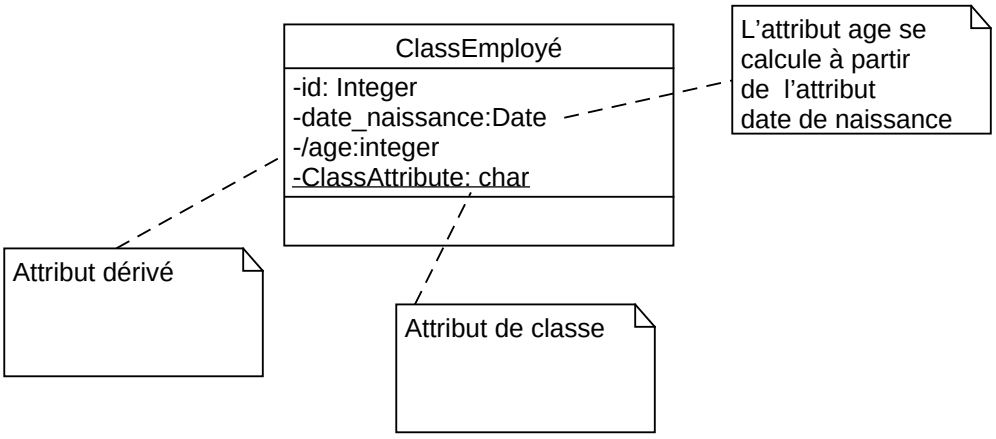


FIGURE 4.3 – Attributs d'une classe

4.3.2 Méthode

Une méthode permet de spécifier le comportement de la classe et/ou de ses instances.

Les méthodes sont des opérations ou des fonctions qui peuvent prendre des valeurs en entrée et modifier les attributs ou produire des résultats.

Quand le nom d'une opération apparaît plusieurs fois avec des paramètres différents, on dit que l'opération est surchargée. En revanche, il est impossible que deux opérations ne se distinguent que par leur valeur retournée.

La déclaration d'une opération contient la visibilité (décrite en section 4.3.3), le nom de l'opération, les types des paramètres et le type de la valeur de retour selon la syntaxe suivante :

$visibilite\ nom\_operation(param1 : type, param2 : type, ...) : type\_resultat$

Méthode de classe :

Comme pour les attributs de classe, il est possible de déclarer des méthodes de classe. Une méthode de classe ne peut manipuler que des attributs de classe et ses propres paramètres.

Cette méthode n'a pas accès aux autres attributs de la classe.

L'accès à une méthode de classe ne nécessite pas l'existence d'une instance de cette classe.

Graphiquement, une méthode de classe est soulignée (voir figure 4.4).

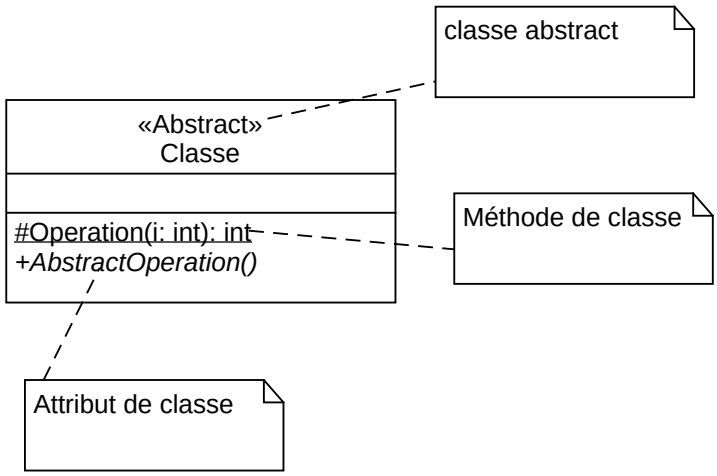


FIGURE 4.4 – Attributs d'une classe

Méthode et classe abstraite :

Une méthode est dite abstraite lorsqu'on connaît son en-tête, mais pas la manière dont elle peut être réalisée (i.e. on connaît sa déclaration, mais pas sa définition).

Une classe est dite abstraite lorsqu'elle définit au moins une méthode abstraite ou lorsqu'une classe parent contient une méthode abstraite non encore définie.

On ne peut instancier une classe abstraite : elle doit se spécialiser d'abord.

Une classe abstraite pure ne comporte que des méthodes abstraites. En Java, une telle classe est appelée une interface.

Pour indiquer qu'une classe est abstraite, il faut ajouter le mot-clef *abstract* avant son nom à l'aide du stéréotype <<*abstract*>> (voir figure 4.4).

4.3.3 Encapsulation et visibilité

Comme nous l'avons évoqué au chapitre 2, l'un des concepts phares de l'orienté objet est l'encapsulation. Dans la pratique, l'encapsulation est un mécanisme qui permet de cacher l'implémentation d'un objet en empêchant l'accès aux données par un autre moyen que les services proposés.

Ces services accessibles (offerts) aux utilisateurs de l'objet définissent ce que l'on appelle l'interface de l'objet (sa vue externe).

L'encapsulation permet donc de garantir l'intégrité des données contenues dans l'objet.

Exemple : Pour modifier la valeur de l'*attribut_1* de ClasseA (figure 4.5), on passe par les opérations *get* et *set* (c'est le cas également pour l'*attribut_2*).

L'encapsulation permet de définir des niveaux de visibilité des éléments. La visibilité déclare la possibilité pour un élément de modélisation de référencer un élément qui se trouve dans un espace de nom différent de celui de l'élément qui établit la référence. Il existe quatre visibilités prédéfinies :

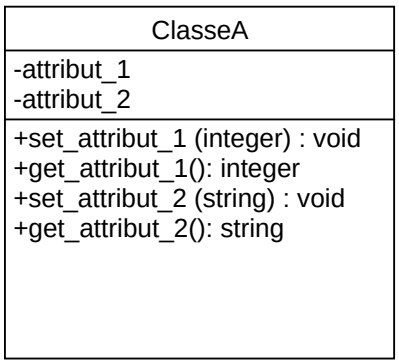


FIGURE 4.5 – Intérêt de l'encapsulation

Private ou - : limite la visibilité d'une propriété à la classe elle-même.

Public ou + : ne limite pas la visibilité d'une propriété.

Protected ou # : limite la visibilité d'une propriété à la classe elle-même et à ses sous-classes.

Package ou ~ ou rien : permet de limiter la visibilité d'une propriété au package de la classe.

La figure 4.6 illustre un exemple d'utilisation de la propriété d'encapsultaion avec différents niveaux de visibilité.

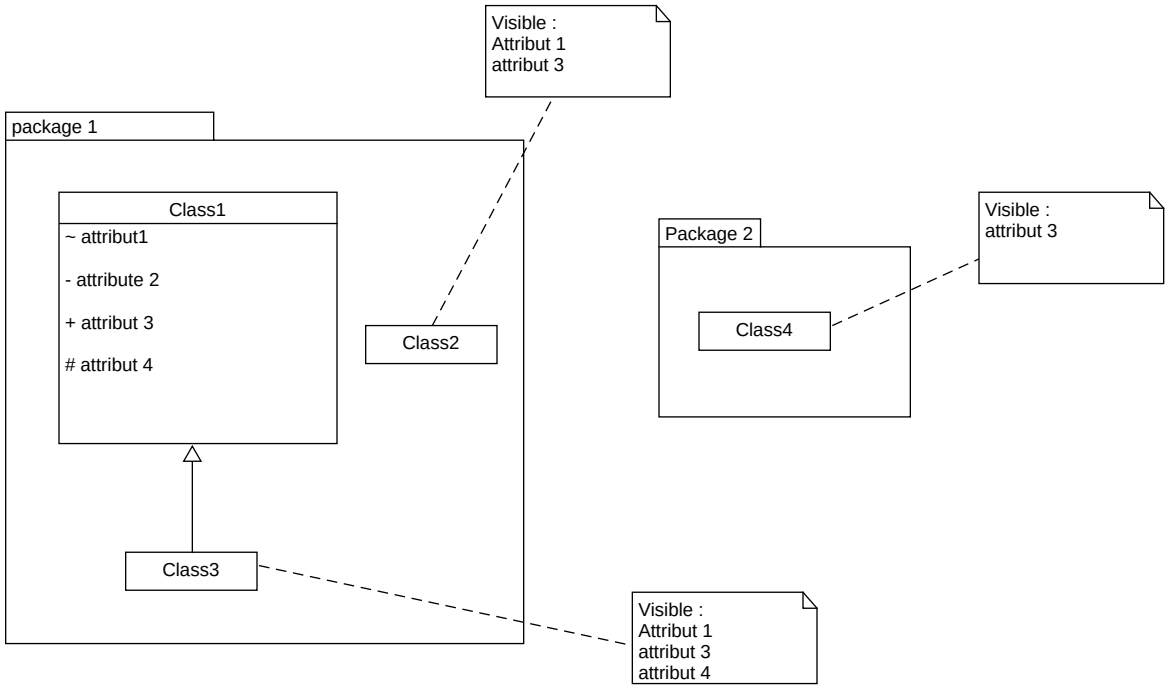


FIGURE 4.6 – Exemple d'encapsulation et de niveaux de visibilité

4.4 Relations entre classes

Il existe une multitude de relations entre les classes préalablement identifiées. Ces relations servent à mettre en avant les liens sémantiques entre les classes et peuvent faciliter grandement la manière dont ces classes seront implémentées.

4.4.1 Relation de dépendance

Une dépendance est une relation unidirectionnelle exprimant une dépendance sémantique entre des éléments du modèle.

La relation de dépendance est la forme la plus faible de relations entre classes.

Une dépendance entre deux classes signifie que l'une des deux utilise l'autre (utilisation du stéréotype <<use>>). Typiquement, il s'agit d'une relation transitoire, au sens où la première interagit brièvement avec la seconde sans conserver à terme de relation avec elle (liaison ponctuelle). Par exemple, lorsqu'une classe utilise un objet d'une autre classe comme argument dans la signature d'une méthode ou alors lorsque l'objet de l'autre classe est créé à l'intérieur de la méthode. Dans les deux cas, la durée de vie de l'objet est très courte, elle correspond à la durée d'exécution de la méthode.

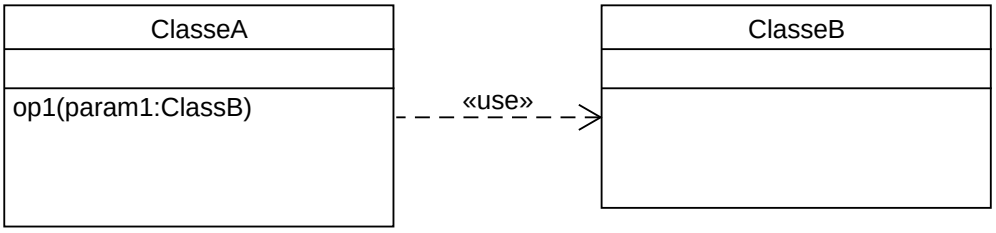


FIGURE 4.7 – Exemple de relation de dépendance

4.4.2 Relation d'association

Une association est une relation entre deux ou plusieurs classes qui décrit les connexions structurelles entre leurs instances.

Exemple : une personne peut posséder des voitures. La relation possède est une association entre les classes Personne et Voiture.

Contrairement à la dépendance qui autorise simplement une classe à utiliser des objets d'une autre classe, la relation d'association est plus forte et signifie qu'une classe contiendra un attribut de la classe associée et inversement.

Selon le nombre de classes impliquées dans la relation, on peut distinguer les associations binaires qui relient entre elles deux classes et les associations n-aire qui relient entre elles n classes.

4.4.2.1 Association binaire

Une association binaire est matérialisée par un trait plein entre les classes associées. Elle peut être ornée d'un nom, avec éventuellement une précision du sens de lecture (> ou <).

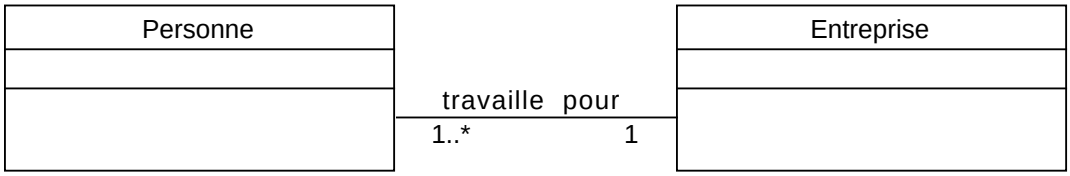


FIGURE 4.8 – Exemple d'association binaire

Quand les deux extrémités de l'association pointent vers la même classe, l'association est dite réflexive (voir figure 4.9).

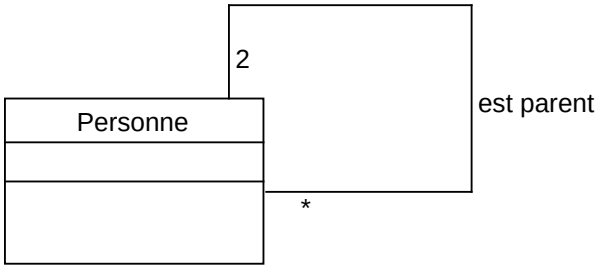


FIGURE 4.9 – Exemple d'association réflexive

4.4.2.2 Association n-aire

Une association n-aire lie plus de deux classes. On représente une association n-aire par un grand losange avec un chemin partant vers chaque classe participante. Le nom de l'association, le cas échéant, apparaît à proximité du losange.

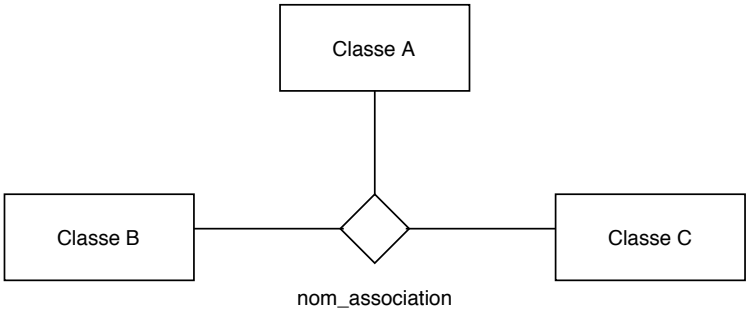


FIGURE 4.10 – Exemple d'association ternaire

Remarque : Il est souvent préférable de traduire une association n-aire en un ensemble d'associations binaires

4.4.2.3 Multiplicité :

Les multiplicités permettent de contraindre le nombre d'objets intervenant dans les instanciations des associations. On en place de chaque côté des associations. Une multiplicité d'un côté spécifie combien d'objets de la classe du côté considéré sont associés à un objet donné de la classe de l'autre côté.

Syntaxe : min..max, où min et max sont des nombres représentant respectivement les nombres minimaux et maximaux d'objets concernés par l'association.

Voici quelques exemples de multiplicité :

exactement un : 1 ou 1..1 ;

plusieurs : * ou 0..* ;

au moins un : 1..* ;

de 3 à six : 3..6.

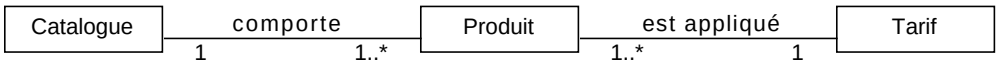


FIGURE 4.11 – Exemple d'association avec multiplicités

Exemple :

Donner pour chaque phrase le diagramme de classe correspondant en indiquant les multiplicités.

1. Un pays n'a qu'une seule ville comme capitale et une ville n'est la capitale que d'un seul pays

2. Un continent contient plusieurs pays, mais un pays ne fait partie que d'un seul continent

3. Un pays peut avoir un roi ou pas et un roi officie dans un pays

La multiplicité dans le cas des associations n-aires :

Dans une association binaire la multiplicité sur la terminaison cible contraint le nombre d'objets de la classe cible pouvant être associés à un seul objet donné de la classe source.

Dans une association n-aire la multiplicité sur le lien de chaque classe s'applique sur une instance de chacune des classes, à l'exclusion de la classe considérée.

Par exemple, si on prend une association ternaire entre les classes (A, B, C), la multiplicité de la terminaison C indique le nombre d'objets C qui peuvent apparaître dans l'association avec une paire particulière d'objets A et B.

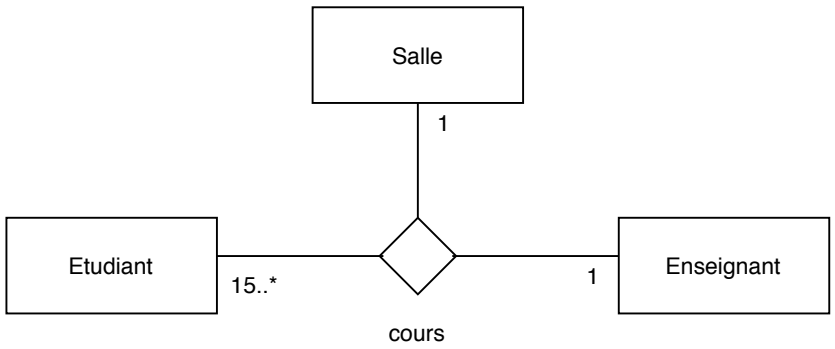


FIGURE 4.12 – Exemple de multiplicité pour une association ternaire

4.4.2.4 Les rôles

Il est possible de nommer les extrémités d'une association en précisant le rôle joué par une classe dans l'association comme l'illustre la figure 4.13.

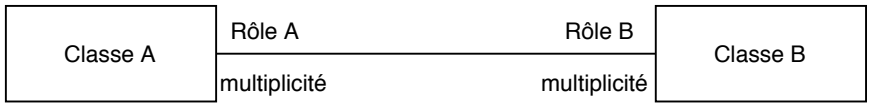


FIGURE 4.13 – Notation des rôles dans une association

Un exemple de l'utilisation des rôles dans une association est illustré par la figure 4.14.

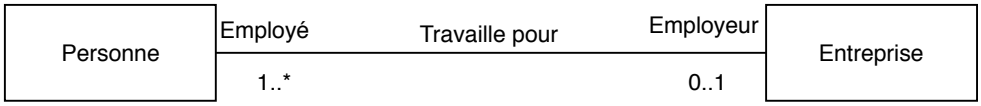


FIGURE 4.14 – Exemple d'utilisation des rôles dans une association

Intérêt des rôles :

L'intérêt des rôles est mis en évidence en particulier dans deux cas :

1. Lorsque plusieurs associations relient deux classes, chaque association exprime un concept distinct qui n'est pas forcément déductible. L'utilisation des rôles prend alors tout son intérêt.
2. La définition de rôle est également indispensable pour les associations réflexives.

Un exemple pour chaque cas est illustré par la figure 4.15.

4.4.2.5 Les contraintes sur les associations

Des contraintes peuvent être définies sur une association ou sur un groupe d'associations. La multiplicité est, par exemple, une contrainte sur le nombre de liens qui peuvent exister entre deux objets.

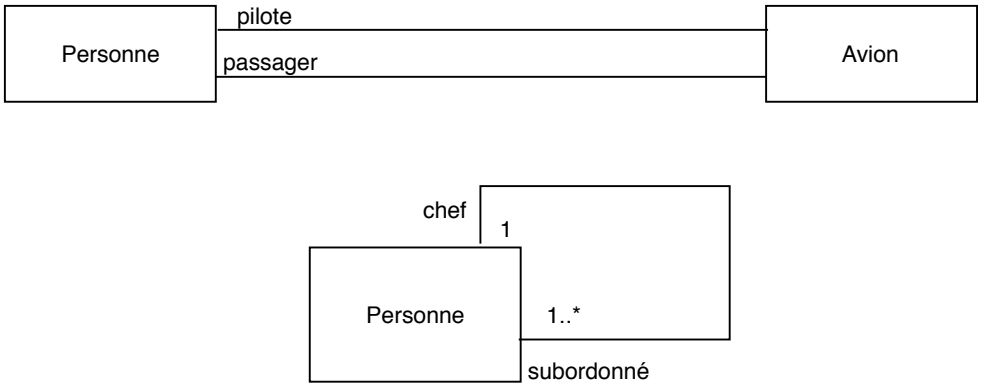


FIGURE 4.15 – Intérêts d'utilisation des rôles

Les contraintes se représentent dans les digrammes UML par des expressions entre accolades. Une contrainte peut être exprimée en langage naturel (texte) ou formel.

OCL (Object Constraint Language) est un langage spécialement conçu pour exprimer des contraintes.

On distingue généralement deux grandes catégories de contraintes :

1. les contraintes propres à une seule association
2. les contraintes entre deux associations.

Voici quelques exemple de contraintes prédéfinies (en OCL) :

1. Sur une association :

   variable : instance modifiable (par défaut).

   frozen : instance non modifiable.

   addOnly : instances ajoutables mais non retirables (si mult. > 1).

   ordered : spécifie qu'une relation d'ordre décrit les objets.

   ...

La figure 4.16 illustre des exemples de contraintes prédéfinies sur une association.

2. Entre deux associations :

   subset : une association est un sous-ensemble d'une autre association.

   xor : une association exclut une autre association (ou exclusif).

   ...

La figure 4.17 illustre des exemples d'utilisation de ces deux contraintes.

4.4.2.6 La classe association

Certaines associations nécessitent souvent l'ajout d'informations supplémentaires. Ces informations ne peuvent être stockées dans aucune des classes associées car elles sont inhérentes à l'association. Il est alors possible de les stocker dans une classe particulière appelée la classe association.

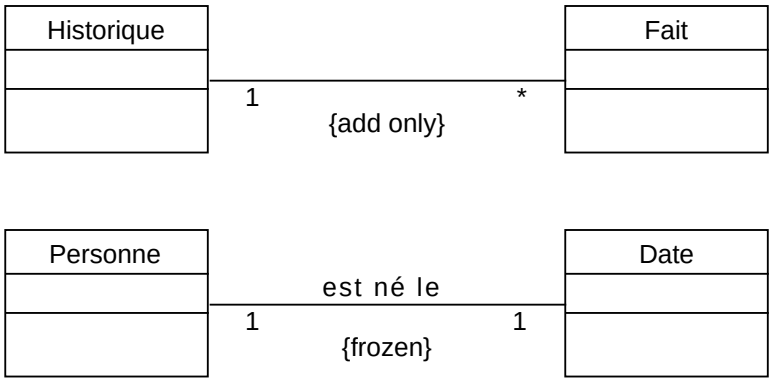


FIGURE 4.16 – Exemple de contraintes sur une seule association

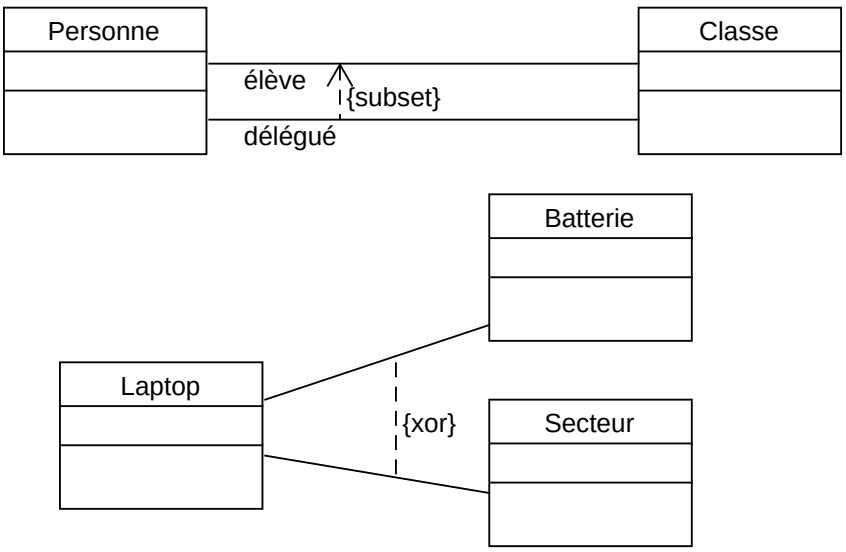


FIGURE 4.17 – Exemple de contraintes entre plusieurs associations

Graphiquement, une classe association se représente exactement de la même manière qu'une classe classique, elle est simplement reliée à une association par un trait en pointillé. La figure 4.18 illustre un exemple de classe association.

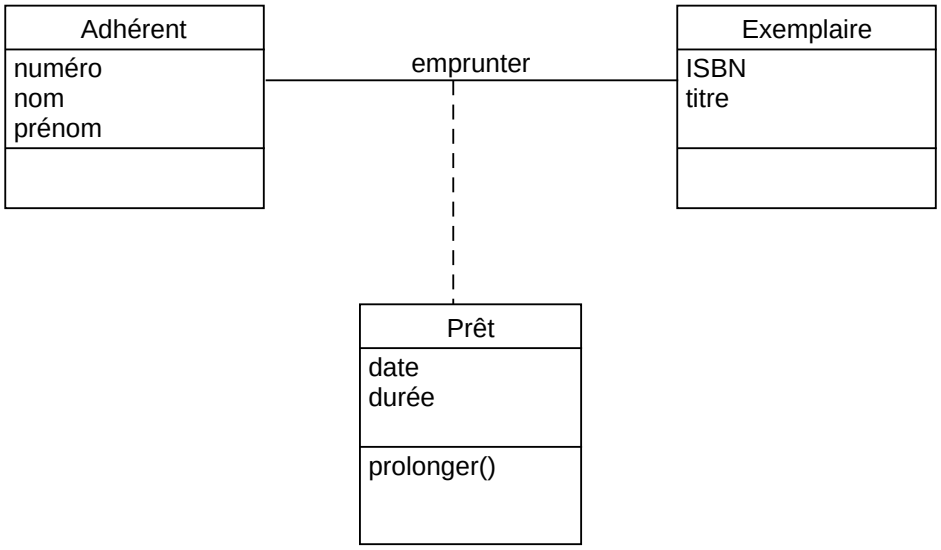


FIGURE 4.18 – Exemple de classe association

4.4.3 Association orientée : la navigabilité

On parle de navigabilité lorsqu'une association est orientée (ou unidirectionnelle). Cela signifie qu'une classe possède un attribut qui fait référence à une autre classe mais l'inverse n'est pas possible ; la deuxième classe ne connaît pas la première.

Par défaut, une association est navigable dans les deux sens (figure 4.19).

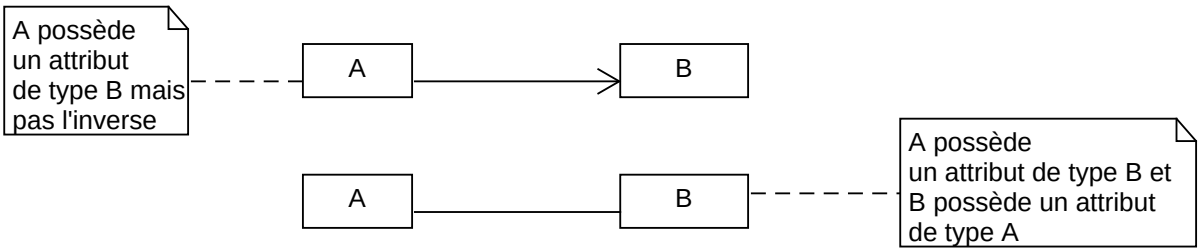


FIGURE 4.19 – Navigabilité versus Association

On représente graphiquement la navigabilité de trois façons différentes (figure 4.20) :

1. Une flèche
2. Une croix du côté de la classe non navigable
3. La combinaison des deux représentations précédentes

FIGURE 4.20 – Représentations de la relation de navigabilité

Exemple :

Si on considère les deux classes *Commande* et *Produit*, nous pouvons conclure que les objets de la classe *Commande* connaissent ceux de la classe *Produit* auxquels ils sont liés, mais pas l'inverse.

Cela implique que la terminaison du côté de la classe *Commande* n'est pas navigable. Les instances de la classe *Produit* ne stockent donc pas de liste d'objets du type *Commande*.

Inversement, la terminaison du côté de la classe *Produit* est navigable : chaque objet commande contient une liste de produits. Cet exemple est illustré par la figure 4.21.

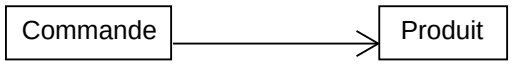


FIGURE 4.21 – Exemple d'utilisation de la relation de navigabilité

4.4.4 Relations d'association particulières : l'agrégation et la composition

Lorsque deux classes sont associées grâce à une association binaire classique rien n'est supposé sur la dépendance et la durée de vie des objets de ces deux classes. Aucune des deux n'est plus importante que l'autre.

Lorsque l'on souhaite marquer la dépendance entre les classes à travers une relation tout/partie où une classe constitue un élément plus grand (tout) composé d'éléments plus petits (partie), il faut utiliser la notion d'agrégation ou de composition (appelée également agrégation forte).

4.4.4.1 Agrégation

L'agrégation est une association qui représente une relation de « contenance » dont la sémantique est du type « composé-composant ». Une instance (agrégat) est composée de plusieurs objets (les objets agrégés).

Graphiquement, on représente l'agrégation par l'ajout d'un losange vide du côté de l'agrégat (voir figure 4.22).

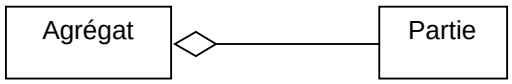


FIGURE 4.22 – Notation de la relation d'agrégation

4.4.4.2 Composition

La relation de composition est une agrégation forte qui décrit une contenance structurelle entre instances. La destruction et la copie de l'objet composite (l'ensemble) impliquent respectivement la destruction ou la copie de ses composants (les parties).

La multiplicité du côté composite ne doit pas être supérieure à 1 (i.e. 1 ou 0..1).

Concrètement, cela signifie que si un objet X est constitué d'un autre objet Y, cet objet ne peut appartenir qu'à l'objet X et ne peut être partagé avec un autre objet.

Graphiquement (figure 4.23), on utilise un losange plein pour représenter la relation de composition.

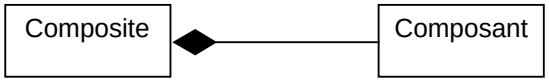


FIGURE 4.23 – Notation de la relation de composition

La figure 4.24 illustre un exemple de diagramme de classe employant l'agrégation et la composition afin de marquer la différence sémantique entre ces deux relations. Néanmoins, il est important de noter que les notions d'agrégation et de composition sont sources de confusion et de querelles entre les modélisateurs.

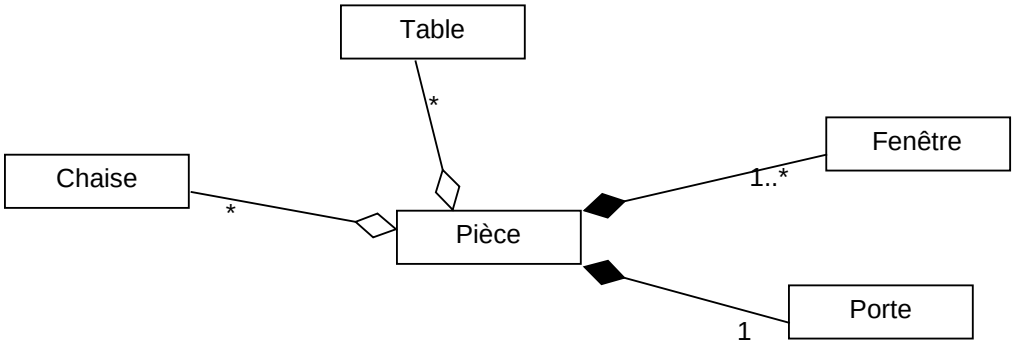


FIGURE 4.24 – Exemple de relations d'agrégation et de composition

4.4.4.3 Différence entre agrégation et composition

Il est souvent difficile de distinguer l'agrégation de la composition car cela est souvent subjectif.

Sémantiquement, l'agrégation indique qu'un objet possède un autre objet. Il y a, à la fois, une relation de possession et d'utilisation, mais contrairement à la composition, ces deux objets peuvent exister indépendamment l'un de l'autre. La suppression de l'objet conteneur n'entraîne pas la suppression de l'objet contenu. Par ailleurs, dans une composition, les objets composites ne peuvent appartenir qu'à l'objet composant.

Pour résumer, il faut vérifier les deux critères suivants pour qu'une agrégation soit considérée comme une composition :

— La multiplicité du côté du composite ne doit pas être supérieure à un.
— Le cycle de vie des parties doit dépendre de celui des composites (en particulier pour la destruction).

Si ces deux critères sont vérifiés alors on peut remplacer l'agrégation par une relation de composition.

4.4.5 Relation d'héritage (généralisation/spécialisation)

Le mécanisme d'héritage permet de mettre en relation des classes ayant des caractéristiques communes (attributs et comportements) en respectant une certaine filiation.

Dans le langage UML, ainsi que dans la plupart des langages objet, le concept d'héritage traduit une relation de généralisation ou de spécialisation (cf. chapitre 2).

dans le contexte du diagramme de classe, la classe spécialisée est intégralement cohérente avec la classe de base, mais comporte des informations supplémentaires.

En UML, la relation d'héritage n'est pas propre aux classes. Elle s'applique à d'autres éléments du langage comme les paquetages, les acteurs ou les cas d'utilisation

4.4.5.1 Propriétés de l'héritage

La relation d'héritage vérifie plusieurs propriétés parmi lesquelles :

- La classe enfant possède toutes les propriétés de la classe parent (attributs et opérations), une classe enfant ne peut toutefois pas accéder aux propriétés privées de la classe parent.

- Une classe enfant peut redéfinir (même signature) une ou plusieurs méthodes de la classe parent. Sauf indication contraire, un objet utilise les opérations les plus spécialisées dans la hiérarchie des classes.

- Une instance d'une classe peut être utilisée partout où une instance de sa classe parent est attendue : c'est le principe de substitution, par contre l'inverse n'est pas toujours vrai.

4.4.5.2 Héritage multiple

Lorsqu'une classe possède plusieurs parents, on parle alors d'héritage multiple. Le langage C++ est un des langages objet permettant son implémentation effective, le langage Java ne le permet pas.

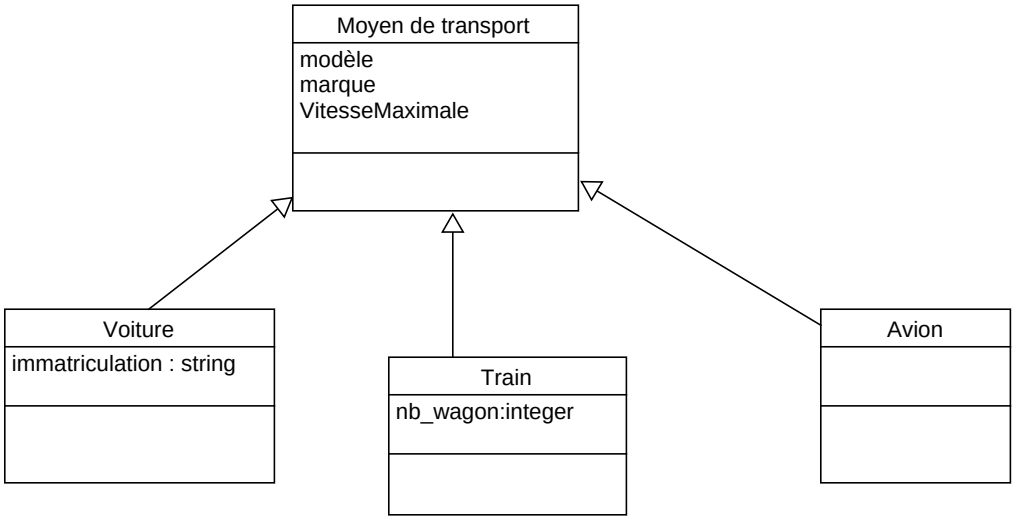


FIGURE 4.25 – Exemple de relation d'héritage

Exemple :

Modéliser la phrase suivante :

Un animal omnivore est à la fois un animal herbivore et carnivore

4.4.5.3 Différence entre héritage et instanciation

Il est important de préciser à ce stade la différence qui existe entre les notions d'héritage et d'instanciation afin de dissiper toute confusion.

L'instanciation signifie qu'un objet d'une classe est créé et qu'il possède toute les caractéristiques qui se trouvent dans la classe alors que l'héritage se passe entre deux ou plusieurs classes et implique une hiérarchie entre elles.

4.5 Eléments de méthodologie pour l'élaboration d'un diagramme de classe

Une démarche couramment utilisée pour bâtir un diagramme de classes consiste à :

1. Trouver les classes du domaine étudié : Les classes correspondent généralement à des concepts ou des substantifs du domaine ;
2. Trouver les associations entre classes : Les associations correspondent souvent à des verbes, ou des constructions verbales, mettant en relation plusieurs classes, comme « est composé de », « pilote », « travaille pour ».
3. Trouver les attributs des classes : Les attributs correspondent souvent à des substantifs, ou des groupes nominaux, tels que « la masse d'une voiture » ou « le montant d'une transaction ». Les adjectifs et les valeurs correspondent souvent à des valeurs d'attributs. Vous pouvez ajouter des attributs à toutes les étapes du cycle de vie d'un projet

(implémentation comprise). N'espérez pas trouver tous les attributs dès la construction du diagramme de classes ;

4. Organiser et simplifier le modèle en éliminant les classes redondantes et en utilisant l'héritage ;
5. Itérer et raffiner le modèle.

Un modèle est rarement correct dès sa première construction. La modélisation objet est un processus non pas linéaire, mais itératif.

4.6 Diagramme d'objets

4.6.1 Définition

Un diagramme d'objets est une instance d'un diagramme de classe permettant d'illustrer l'état d'un système de manière statique. Un diagramme d'objets représente :
- des objets (i.e. instances de classes) et
- leurs liens (i.e. instances de relations)

Il permet de donner une vue figée de l'état d'un système à un instant donné. La notation des diagrammes d'objet est dérivée de celle des diagrammes de classe.

4.6.2 Intérêts du diagramme d'objets

Un diagramme d'objets peut être utilisé pour plusieurs raisons :
- prendre une image (snapshot) d'un système à un moment donné.
- illustrer le modèle de classes en montrant un exemple qui explique le modèle ;
- préciser certains aspects du système en mettant en évidence des détails imperceptibles dans le diagramme de classes ;

Ainsi, le diagramme de classes modélise les règles et le diagramme d'objets modélise des faits.

4.6.3 Les objets

Comme cela a été mentionné préalablement, un objet n'est autre qu'une instance de classe. Il représente l'état d'une classe à un instant précis. Comme pour les classes, les objets sont représentés par des cadres compartimentés. En revanche, les noms des objets sont soulignés.

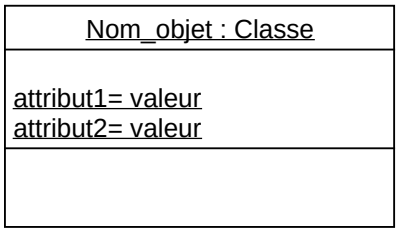


FIGURE 4.26 – Notation UML pour représenter un objet

Un objet peut également être représenté par un rectangle qui contient soit le nom de l'objet, soit le nom de la classe. Les instances peuvent alors être anonymes (a), nommées (b, e), orphelines (c) ou multiples (d) Les valeurs de certains attributs significatifs d'un objet peuvent être spécifiées (4.27).

4.6.4 Les liens

Les objets sont reliés par des instances de relations qui existent entre les classes. Ces instances de relations sont appelées : « liens ». Un lien représente une relation entre objets à un instant donné.

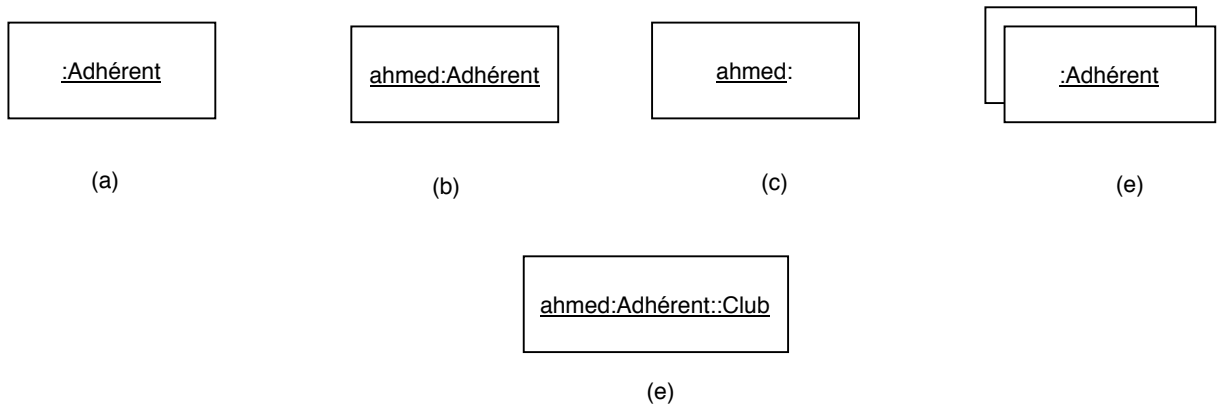


FIGURE 4.27 – Différentes représentations d'objet

Si le lien a un nom, ce dernier est souligné (voir figure 4.28). On ne représente bien sûr pas les multiplicités qui n'ont aucun sens au niveau d'un diagramme d'objets.

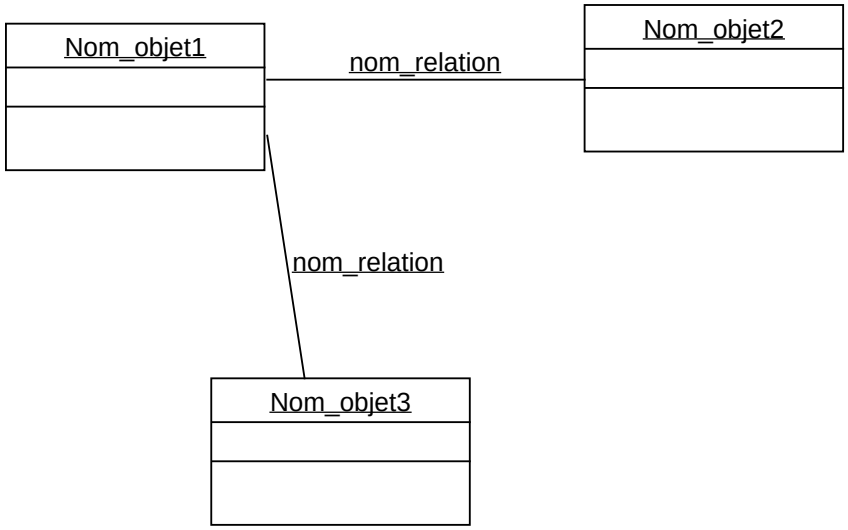


FIGURE 4.28 – Notation UML des liens dans un diagramme d'objets

4.6.5 Différence entre diagramme de classes et diagramme d'objets

La plupart des caractéristiques des relations qui existent dans un diagramme de classes peuvent être reportées dans un diagramme d'objets. Ceci permet de faciliter la compréhension de l'interaction. Ces caractéristiques peuvent être le nom, le nom des rôles,... à l'exception des multiplicités.

Le diagramme d'objets se distinguera donc d'un diagramme de classe principalement grâce aux noms des objets qui sont soulignés.

La figure 4.29 illustre l'équivalence entre un diagramme de classes où une classe Personne est

associée à elle-même (association réflexive) et un diagramme d'objets qui réutilise la notion de rôle pour distinguer entre deux instances de la classe Personne.

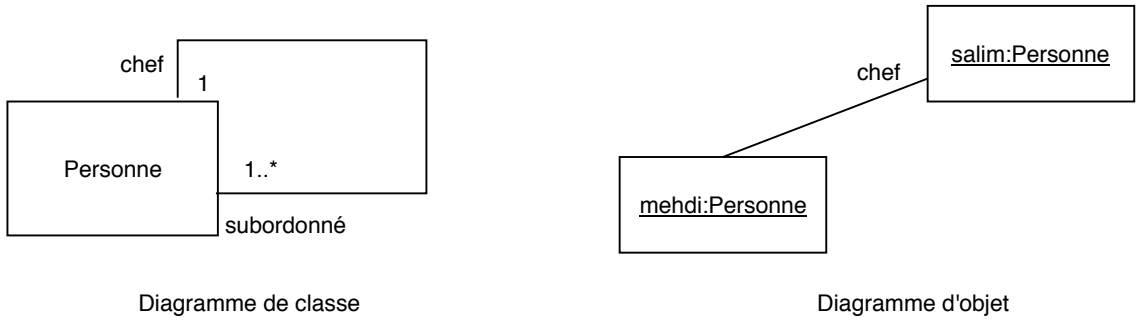


FIGURE 4.29 – Equivalence entre diagrammes de classes et d'objets

4.6.6 Cas d'étude

Modélisez les phrases suivantes à l'aide d'un diagramme de classe et d'un diagramme d'objet :

Une entreprise fait travailler au moins une personne.

Une entreprise est caractérisée par son numéro de SIRET

Soit une entreprise alpha dont le numéro de SIRET est le 4586 qui emploie une personne p1 et une seconde personne

Donnez les diagrammes de classes et d'objets correspondant.

4.6.6.1 Réalisation des diagrammes de classes et d'objets :

Identification des classes et des instances de classes :

L'analyse des phrases énoncées dans le cas d'étude, permet d'identifier deux classes : la classe Entreprise et la classe Personne.

Le numéro de SIRET est un attribut de la classe entreprise.

L'entreprise alpha est une instance (objet) de la classe Entreprise. p1 et une seconde personne (objet anonyme) sont des instances de la classe Personne.

Identification des relations :

Les classes Entreprise et Personne sont reliées par la relation « travaille pour » qui est une association. On distingue également la multiplicité de cette relation à travers la phrase : « Une entreprise fait travailler au moins une personne ».

Cette association sera instanciée dans le diagramme d'objet à travers les liens entre l'entreprise alpha et les personnes p1 et anonyme.

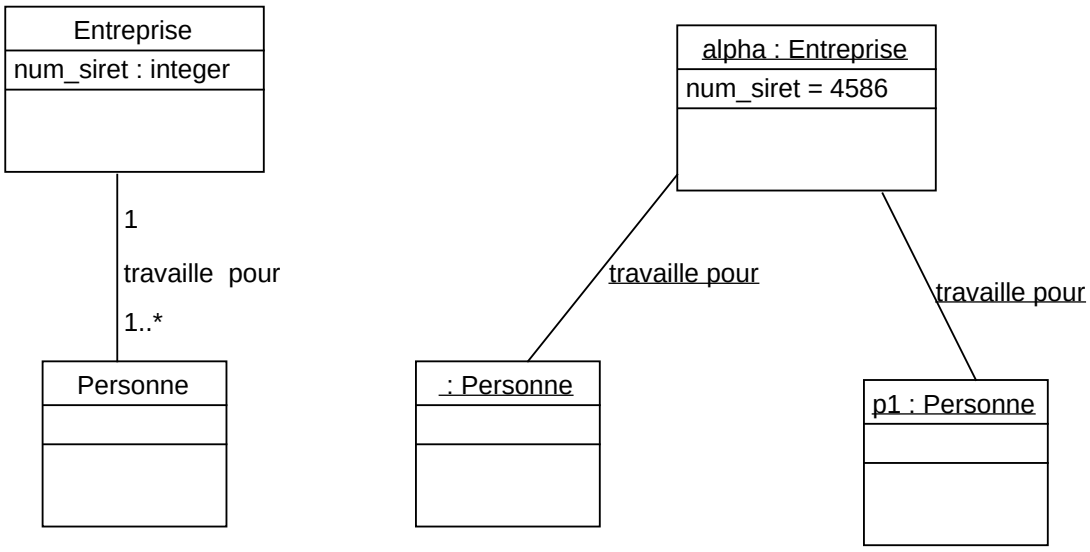


FIGURE 4.30 – Diagrammes de classes et d'objets obtenus

4.7 Conclusion

Dans ce chapitre nous avons étudié le diagramme de classes qui est le pivot d'une modélisation orientée objet ainsi que le diagramme d'objets.

Ces deux diagrammes permettent de décrire la structure interne du système et peuvent être utilisés aussi bien dans les étapes de spécification et de conception que lors de la phase d'implémentation.

Nous avons vu que la représentation statique s'appuie sur les concepts de classe et d'objet mais également sur les relations qui peuvent exister entre elles.

Les classes inclus à la fois les données qu'on appelle attributs et les opérations qui constituent les traitements sans qu'aucun détail d'implémentation ne soit représenté directement dans le diagramme.

Dans les chapitre suivants, nous allons étudier quelques diagrammes utilisés pour fournir une vue dynamique du système.

4.8 Exercices

Exercice 1 : Propriétés d'une classe

Un salarié est caractérisé par son nom, son prénom, son genre (homme/femme), sa date de naissance, son âge et sa fonction. L'âge est calculé à partir de sa date de naissance. Les attributs de la classe Salarié sont privés ; le nom, le prénom ainsi que la fonction doivent pouvoir être accessibles par des opérations publiques.

1. Donner une représentation UML de la classe Salarié en remplissant tous les compartiments de manière adéquate.

Les salariés doivent pouvoir calculer leurs revenues ainsi que le montant de leurs charges. Les revenues sont de deux types : le salaire de base et les primes de rendement. Pour calculer les charges globales, on applique un coefficient de 20

De plus, le calcul des revenues et des charges ne se fait pas de la même manière si la personne est à la retraite.

2. Enrichissez la représentation précédente pour prendre en compte ces nouveaux éléments

Exercice 2 : Relations entre les classes

Modéliser les phrases suivantes à l'aide de diagrammes de classe en choisissant le type de relation appropriée :

1) Tout écrivain a écrit au moins une œuvre

2) Une bibliothèque contient des livres

3) Un cinéma est constitué de plusieurs salles. Les films sont projetés dans des salles. La projection de films dans des salles a lieu à une heure déterminée.

4) Les étudiants et les enseignants sont deux sortes de personnes. Un doctorant est un étudiant qui assure des enseignements.

Exercice 3 : Élaboration d'un diagramme de classe complet

Un musée contient des œuvres d'art. Parmi ces œuvres, on retrouve des tableaux de peintres célèbres de tendances artistiques différentes ainsi que des sculptures.

Tout œuvre a un auteur, une date d'acquisition, un titre et un numéro de catalogue. Une œuvre est exposée dans l'une des salles du musée. Chaque salle est caractérisée par son nom et le nombre d'œuvres qui y est exposé.

Chaque œuvre doit être assurée par une compagnie d'assurance dont ont connaît le nom et l'adresse. Une œuvre peut être acquise, restaurée ou empruntée. Les tableaux peuvent subir un vernissage et les sculptures sont de différentes matières (marbre, bronze, ...).

1. Réalisez un diagramme de classe du système

Exercice 4 :

Un client passe une ou plusieurs commandes.

Un client possède un nom et un numéro client.

Une commande est passée à une date et est identifiée par son numéro.

Ahmed est un client régulier qui a passé 2 commandes au cours du mois de Mars de l'année 2015 : la commande numéro 87 le 03/03/2015 et la commande 126 le 19/03/2015. Leila est une cliente épisodique qui a passé une commande le 3 Mars 2015

1. Donnez les diagrammes de classe et d'objet correspondant

2. Est-ce que le diagramme d'objets de la figure 4.31 est correct ?

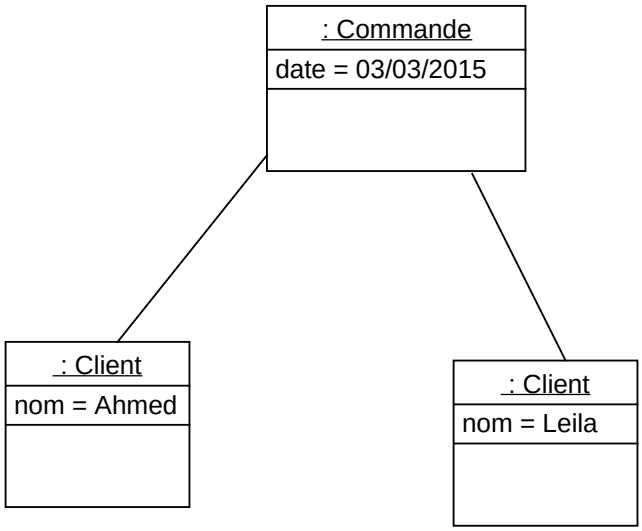


FIGURE 4.31 – Diagramme d'objets

Exercice 5 :

1. Dessinez le diagramme d'objets correspondant à la situation décrite ci-dessous :

Ahlem, Salim, Assia, Manel et Nouri sont des étudiants inscrits uniquement au département Mathématiques et informatique.

Nouri et Ahlem étudient les matières dont les intitulés sont :« algorithmique » et « probabilités ».

Salim étudie la matière « réseaux avancées » dont le code est RA et le volume horaire est de 30heures.

Assia, Nouri et Manel étudient la matière« langage C ».

Par contre, aucun étudiant n'étudie la matière « analyse de données ».

2. En déduire un diagramme de classe.

Exercice 6 :

Le directeur d'une chaîne d'hôtel vous demande de concevoir une application de gestion de ses hôtels selon les informations suivantes :

Un hôtel est géré par un responsable et dispose d'un certain nombre de chambres. Les pièces de l'hôtel qui ne sont pas des chambres (hall d'accueil, cuisine, ...) ne font pas partie de l'étude (hors contexte).

Chaque chambre dispose de lits et se loue à un prix donné. Il existe deux catégories de chambres, les chambres standards et les suites. Une suite possède une salle de bain.

Sachant qu' :

- Un hôtel a pour caractéristiques son nom, son adresse et le nombre d'étoiles dont il dispose.
- Une chambre est définie par son numéro et son prix.
- Une personne est soit un client de l'hôtel, soit un responsable de l'hôtel.

1. Réaliser un diagramme de classe

2. En vous basant sur le précédent diagramme de classe, faites un diagramme d'objets qui représente cette situation : « L'hôtel El Djazair a pour responsable M. Mouloud et dispose de 20 chambres. M. Kamel est un client de l'hôtel ».

Chapitre 5

Diagrammes UML d'interaction : vue dynamique

Dans les chapitres précédents, nous avons d'abord montré comment UML décrivait les fonctionnalités du système offertes à ses utilisateurs à l'aide du diagramme de cas d'utilisation (vue fonctionnelle). Par la suite, nous nous sommes intéressés à la structure de ce système et aux différents modules qui le composent à travers les digrammes de classes et d'objets (vue statique). Néanmoins, nous ne pouvions pas décrire l'évolution du système dans le temps, ni même ses interactions.

Dans ce chapitre, nous allons nous pencher sur la modélisation de la dynamique d'un système afin de mieux le comprendre. Nous allons voir comment UML met à notre disposition plusieurs diagrammes riches sémantiquement et capables de modéliser les différentes facettes de la dynamique d'un système.

Concepts dynamiques d'UML

Les concepts dynamiques d'UML visent à montrer le comportement du système, les interactions des objets qui le constituent et leur évolution dans le temps.

Pour cela, ces concepts mettent en avant la chronologie et décrivent le comportement du système et son évolution de différentes façons. Le comportement du système dans le temps et les interactions entre les objets qui le constituent sont ainsi décrits à l'aide des diagrammes d'interaction. L'évolution des objets et de leur état dans le temps est, quant à lui, décrit à l'aide du diagramme d'états-transitions où le comportement d'un objet est représenté sous la forme d'un automate à états.

Dans le présent chapitre, nous nous intéressons aux diagrammes d'interaction. Le chapitre 6 est consacré au diagramme d'états-transitions.

Rôle des concepts dynamiques dans le processus de développement d'un logiciel

La phase d'analyse dynamique intervient chronologiquement après l'étape d'analyse statique car elle s'appuie sur les objets identifiés et décrits dans les modèles statiques. Elle s'appuie également sur les évènements provenant de l'extérieur du système et décrits par le modèle

des use cases. L'analyse dynamique offre un support indispensable lors du passage à la phase de conception car elle permet de décrire les enchaînements des méthodes. Elle sert également lors des phases de test et de maintenance car elle permet d'effectuer des tests d'impact lors de la modification des objets.

L'étape de modélisation dynamique reste néanmoins plus difficile à mettre en œuvre car elle nécessite une compréhension fine du fonctionnement global du système.

5.1 Diagrammes d'interaction

Les diagrammes d'interaction mettent en avant les interactions des objets du système en offrant un point de vue temporel sur ces interactions.

Ces diagrammes peuvent servir à différentes phases du cycle de vie d'un logiciel.

Pendant la phase d'analyse des besoins, ils permettent de décrire des scénarios de cas d'utilisation afin d'enrichir la description du système et de ses fonctionnalités. Les diagrammes d'interaction montrent ainsi les interactions entre acteurs et système et comment les différents objets communiquent pour réaliser une certaine fonctionnalité.

Ils permettent de mettre en évidence progressivement les objets qui constituent un système au travers de différents scénarios. Le regroupement conceptuel de ces objets et leur abstraction permettra l'ébauche d'un diagramme de classes reflétant un scénario particulier de fonctionnement du système. Il devient ainsi plus aisé d'élaborer un diagramme de classes regroupant les différents diagrammes de classes partiel recueillis à l'aide des scénarios de fonctionnement du système.

Pendant la phase de conception, ces diagrammes permettent de mettre en avant les interactions entre un jeu d'objets mais également à réfléchir à l'affectation de responsabilités aux objets (qui crée les objets, qui permet d'accéder à un objet). Ils offrent ainsi un bon moyen de contrôler la cohérence entre les diagrammes en les élaborant en parallèle avec les diagrammes de classes.

Les interactions représentent le cœur du système, son essence même en terme de fonctionnement. Une utilisation régulière des diagrammes d'interaction est signe d'une véritable approche objet.

Avec l'avènement d'UML 2, les diagrammes d'interaction comprennent désormais quatre diagrammes : le digramme de séquence, le diagramme de communication (anciennement appelé diagramme de collaboration en UML 1.x), le diagramme de temps et le diagramme globale d'interaction.

Le diagramme de séquence met en évidence le déroulement séquentiel. Le diagramme de communication met en évidence les échanges de messages Le diagramme de temps spécifie précisément l'instant d'occurrence des événements dans le temps ce diagramme est utile car le diagramme de séquence s'intéresse à l'ordre des interactions mais ignore la mesure précise du passage du temps. Le diagramme globale d'interaction permet de rassembler dans le même diagramme des comportements différents sans leurs détails pour dessiner comme son nom l'indique une vue globale des interactions.

Dans ce cours, nous nous focalisons principalement sur les diagrammes de séquence et de communication.

5.2 Diagrammes de séquence

5.2.1 Définition

En UML 2, le diagramme de séquence fait partie des diagrammes comportementaux et plus précisément des diagrammes d'interactions.

Le diagramme de séquence permet de décrire la dynamique d'un système à travers les interactions des objets qui le composent en insistant sur l'ordre chronologique des échanges (messages). Il permet de modéliser les flux de contrôle et de données échangées lors des envois de messages.

Le diagramme de séquence représente donc les participants à une interaction et la séquence de messages qu'ils s'envoient. Les acteurs et les objets sont représentés par des lignes verticales appelées lignes de vie et chaque message est symbolisé par une flèche horizontale allant de l'émetteur au récepteur. Le temps y est explicitement représenté et s'écoule sur l'axe vertical, de haut en bas, sans échelle de temps. Avec la version UML 2 ce diagramme a gagné en puissance de représentation grâce à de nouvelles constructions permettant d'enrichir le modèle.

Un diagramme de séquence est généralement placé dans un cadre d'interaction représenté à l'aide d'un rectangle qui dispose d'un pentagone en haut à gauche à l'intérieur duquel se trouve une étiquette sd (pour sequence diagram) suivie du nom du diagramme (figure5.1).

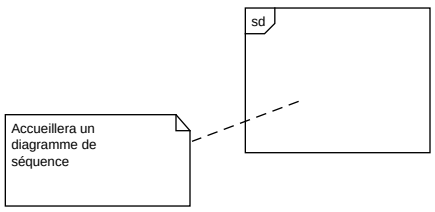


FIGURE 5.1 – Cadre d'interactions

5.2.2 Intérêt et utilisation du diagramme de séquence

A moins que le système à modéliser ne soit extrêmement simple, nous ne pouvons pas modéliser la dynamique globale du système dans un seul diagramme. Nous ferons donc appel à un ensemble de diagrammes de séquences chacun correspondant à une sous fonction du système généralement pour illustrer un cas d'utilisation.

Cela permet de montrer comment les enchaînements se succèdent ou à quel moment sont sollicités les acteurs secondaires. Dans ce cas, et par convention, il est recommandé de placer les acteurs principaux à gauche, puis un objet unique représentant le système et enfin les acteurs secondaires éventuels à droite du système.

5.2.3 Ligne de vie

Dans un diagramme de séquence, une ligne de vie représente un participant à une interaction (objet ou acteur). La syntaxe de son libellé est la suivante :

nomLigneDeVie : NomClasseOuActeur

Une ligne de vie est une instance, donc il y a nécessairement les deux points (:) dans son libellé. Par ailleurs, au moins un des deux noms doit être spécifiés dans l'étiquette les deux points sont, quant à eux, obligatoires.

Une ligne de vie est représentée à l'aide d'un rectangle contenant son nom et une ligne verticale qui permet de schématiser le temps.

Le diagramme de séquence contient plusieurs lignes de vie car il traite des interactions entre plusieurs objets.

Une ligne de vie permet également de montrer les périodes pendant lesquelles elle est active, c'est-à-dire où elle exécute ses méthodes. Les périodes d'activité sont représentées graphiquement à l'aide d'un rectangle dessiné à la place de la ligne de vie verticale.

La représentation graphique en UML de lignes de vie est illustrée par la figure 5.2.

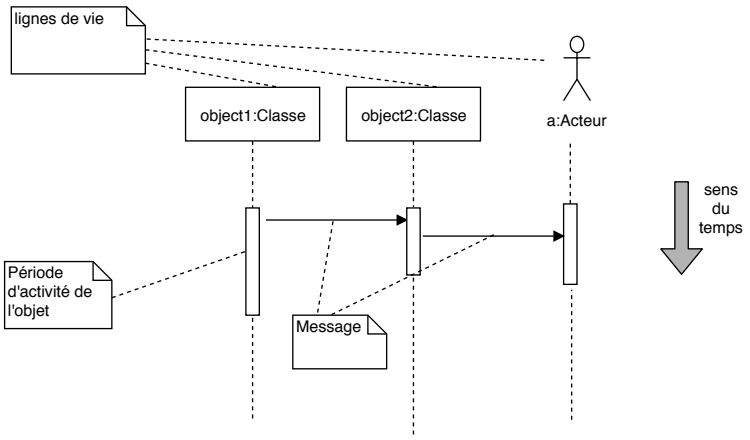


FIGURE 5.2 – Représentation graphique de lignes de vie

5.2.4 Les messages

Dans un diagramme de séquence, les lignes de vie communiquent en s'échangeant des messages. Ces messages sont représentés par des flèches du haut vers le bas, le long des lignes de vie, dans un ordre chronologique.

La notation des messages est la même que celle pratiquée dans un diagramme de communication. Un message est un mécanisme qui provoque l'exécution d'une opération au niveau de l'objet destinataire.

Un même message peut déclencher des opérations différentes selon l'objet qui le reçoit selon le principe de polymorphisme inhérent au paradigme orienté objet.

Plusieurs types de messages existent, les plus utilisés sont :

1. les messages synchrones particulièrement utilisés pour l'invocation d'une opération au niveau de l'objet cible.
2. les messages asynchrones particulièrement utilisé pour l'envoi d'un signal
3. les messages de création et destruction d'une instance

5.2.4.1 Messages synchrones

L'invocation d'une opération est le type de message le plus utilisé en programmation objet. L'invocation peut être asynchrone ou synchrone mais dans la pratique, la plupart des invocations sont synchrones. Lorsqu'il envoie un message synchrone, l'émetteur reste bloqué le temps que dure l'invocation de l'opération. En effet, si une ligne de vie A envoie un message synchrone à une ligne de vie B, A reste bloquée tant que B n'a pas terminé.

Un message synchrone est représenté par une flèche dont l'extrémité est pleine comme l'indique le schéma illustré par la figure 5.3.

On peut associer aux messages d'appel de méthode un message de retour (en pointillés) marquant la reprise du contrôle par l'objet émetteur du message synchrone. Le récepteur d'un message synchrone rend alors la main à l'émetteur du message en lui envoyant un message de retour.

Les messages de retour sont optionnels ; la fin de la période d'activité marque également la fin de l'exécution d'une méthode.

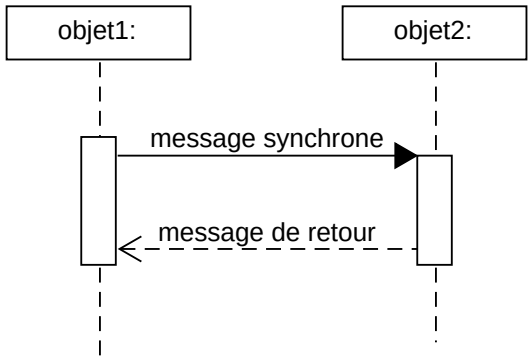


FIGURE 5.3 – Représentation graphique de messages synchrone et de retour

5.2.4.2 Messages asynchrones

Un signal est, par définition, un message asynchrone. Une interruption ou un événement sont quelques exemples de signaux.

Les messages asynchrones n'attendent pas de réponse et ne bloquent pas l'émetteur qui ne sait pas si le message arrivera à destination ni même s'il sera traité par le destinataire.

Graphiquement, un message asynchrone se représente par une flèche en traits pleins et à l'extrémité ouverte partant de la ligne de vie d'un objet expéditeur et allant vers celle de l'objet cible (voir figure 5.4).

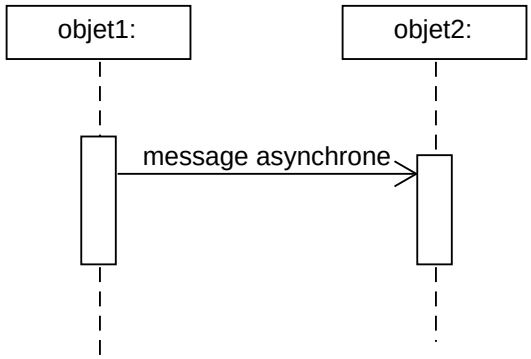


FIGURE 5.4 – Représentation graphique d'un message asynchrone

5.2.4.3 Message de création et de destruction

Message de création :

La création d'une ligne de vie se fait à l'aide d'un message stéréotypé <<create>> en pointant vers le rectangle en tête de la ligne de vie.

Message de destruction :

La destruction d'un objet est matérialisée par une croix qui marque la fin de la ligne de vie de l'objet. La destruction d'une ligne de vie peut être due à un message stéréotypé <<destroy>> mais pas forcément. En effet, la destruction d'un objet n'est pas nécessairement consécutive à la réception d'un message.

Les représentations graphiques des messages de création et de destruction de lignes de vie sont illustrées par la figure 5.5

5.3 Exemple d'utilisation d'un digramme de séquence

Dans cet exemple, nous allons utiliser un diagramme de séquence pour décrire les interactions des objets intervenant dans le scénario du cas d'utilisation « Retirer de l'argent » au niveau d'un GAB. Ce scénario illustre un des cas d'utilisation identifié dans l'étude de cas réalisé dans le chapitre 3 qui modélisait le fonctionnement d'un guichet automatique bancaire.

Description du scénario :

1. Le porteur de carte introduit sa carte dans le lecteur de carte du GAB

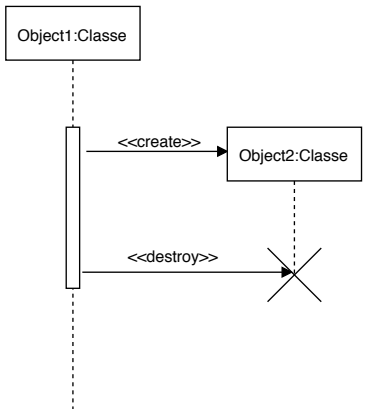


FIGURE 5.5 – Représentation graphique des messages de création et de destruction de lignes de vie

2. Le GAB vérifie que la carte introduite est bien une carte bancaire
3. Le GAB demande au porteur de saisir le code secret
4. Le porteur de carte saisit le code
5. Le GAB compare le code saisi avec celui qui est codé sur la puce de la carte
6. Le code est bon
7. Le GAB demande une autorisation au système d'autorisation
8. Le système d'autorisation donne son accord et indique le crédit hebdomadaire
9. Le GAB demande au porteur de carte de saisir le montant du retrait
10. Le porteur de carte saisit le montant
11. Le GAB contrôle le montant saisi par le porteur de carte
12. Le GAB éjecte la carte afin de la remettre au porteur de carte.
13. Le GAB donne les billets au porteur de carte

    ...

Diagramme de séquence obtenu :

Les lignes de vie sont : l'acteur porteur de carte, le GAB (le système), ainsi que l'acteur secondaire système d'autorisation.

Le diagramme de séquence obtenu suite à l'analyse de la description du scénario est illustré par la figure 5.6

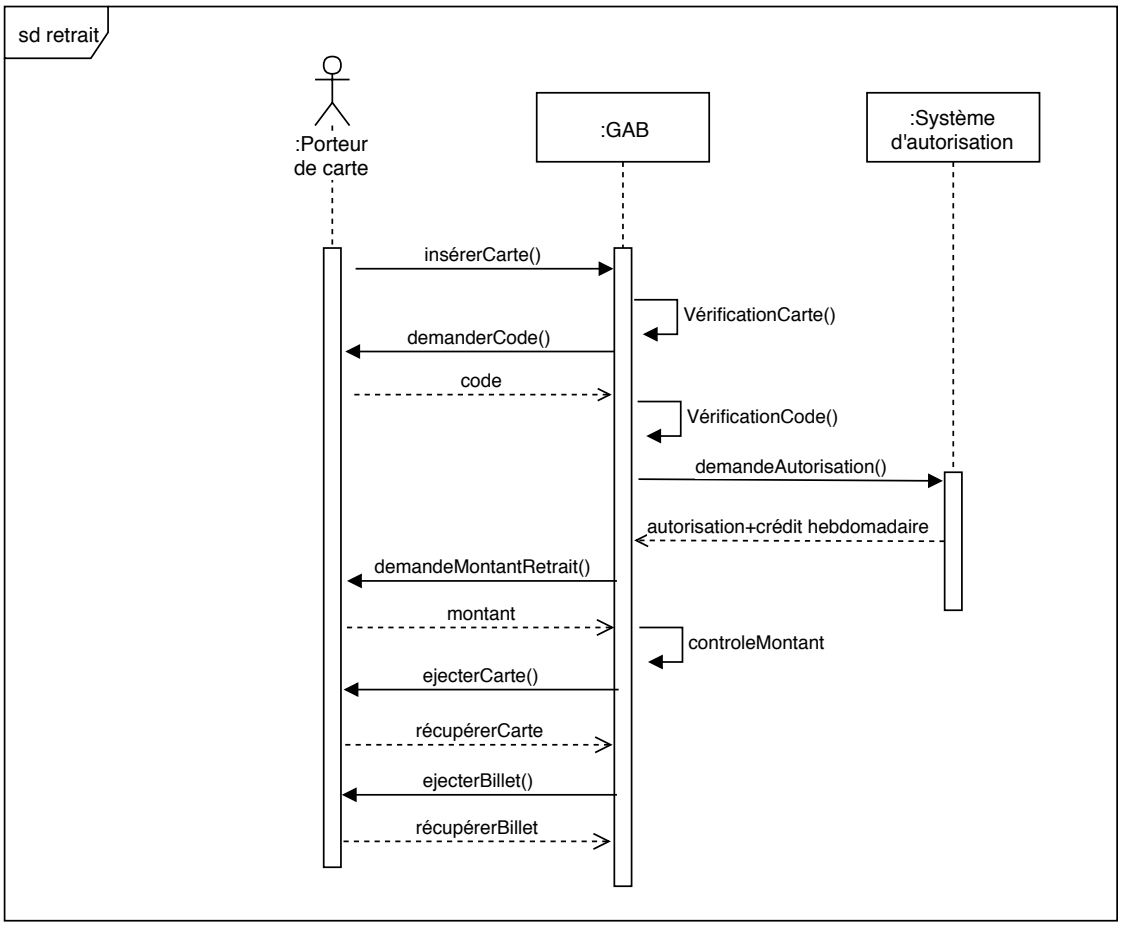


FIGURE 5.6 – Diagramme de séquence d'un scénario de retrait d'argent

5.3.1 Les fragments combinés

Les fragments combinés permettent de décomposer une interaction complexe en fragments suffisamment simples pour être compris.

Un fragment combiné se représente de la même façon qu'une interaction. Il est schématisé par un rectangle dont le coin supérieur gauche contient un pentagone. Ce pentagone contient un opérateur (l'opérateur d'interaction) qui détermine la modalité d'exécution. Les principales modalités sont le branchement conditionnel et la boucle.

5.3.1.1 Les opérateurs

Un fragment combiné est défini par un opérateur et des opérandes. L'opérateur conditionne la signification du fragment combiné. Les opérandes d'un opérateur d'interaction sont séparés par une ligne pointillée. Les conditions de choix des opérandes (éventuels) sont données par des expressions booléennes entre crochets ([ ]).

En UML 2, douze opérateurs ont été définis. Ces opérateurs peuvent être regroupés par fonction comme suit :

- Opérateurs de branchement (choix et boucle) : Alternative, option, loop, break
- Opérateurs contrôlant l'envoi en parallèle de messages : parallel, criticalregion
- Opérateurs contrôlant l'envoi de messages : ignore, consider, assertion et negative
- Opérateur fixant l'ordre d'envoi des messages :strict sequencing et weak sequencing

En plus de ces opérateurs, il existe un opérateur « ref » qui permet de réutiliser une interaction.Il suffit pour cela de placer un fragment portant la référence « ref » là où l'interaction est utile.

Par analogie à la programmation, si un fragment combiné peut être vu comme une fonction, l'utilisation de l'opérateur « ref » peut alors être interprété comme l'appel à cette fonction. Un exemple d'utilisation de ce fragment combiné est illustré par la figure 5.7.

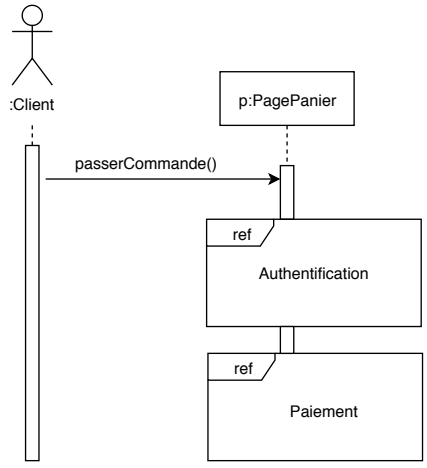


FIGURE 5.7 – Exemple d'utilisation du fragment combiné «ref»

Nous allons détailler par la suite le fonctionnement de quelques opérateurs de fragment combiné.

L'opérateur alternative (alt)

alt est un opérateur conditionnel qui possède plusieurs alternatives (ou opérandes). C'est un peu l'équivalent d'un switch dans les langages de programmation C et C++.

Les différentes alternatives sont spécifiées dans des zones délimitées par des pointillés. Chaque alternative contient une condition de garde. Les conditions sont spécifiées entre crochets dans chaque zone (voir figure 5.8). L'absence de condition de garde implique une condition vraie

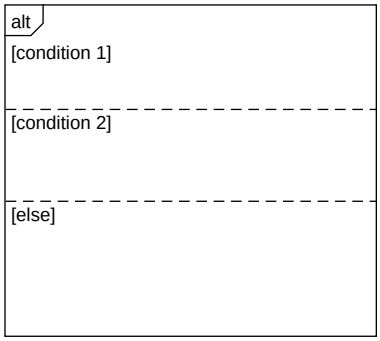


FIGURE 5.8 – Représentation graphique du fragment combiné alt

(true). La condition else est vraie si aucune autre condition n'est vraie.

Attention, si plusieurs alternatives prennent la valeur vraie, le choix est non déterministe.

Exemple :

On reprend l'exemple du guichet automatique bancaire, mais en considérant cette fois la possibilité que le porteur de carte saisisse un code erroné et que dans ce cas la carte lui est retournée par le GAB. Les interactions obtenues sont illustrées par la figure 5.9.

L'opérateur option (ou opt)

L'opérateur option (ou opt) représente une interaction qui peut se produire ou pas. Cet opérateur comporte un seul opérande et une condition de garde associée. Le sous-fragment s'exécute uniquement si la condition de garde est vraie (ne s'exécute pas dans le cas contraire). C'est un peu l'équivalent d'un alt avec une seule branche et sans else. La représentation graphique de ce fragment est illustrée par la figure 5.10.

Exemple :

Nous souhaitons illustrer la possibilité pour un utilisateur de solliciter de l'aide lorsqu'il lance une application. Etant donné que ce dernier peut ou pas avoir besoin d'aide, nous utilisons naturellement l'opérateur « opt » pour modéliser cette possibilité (voir figure 5.11).

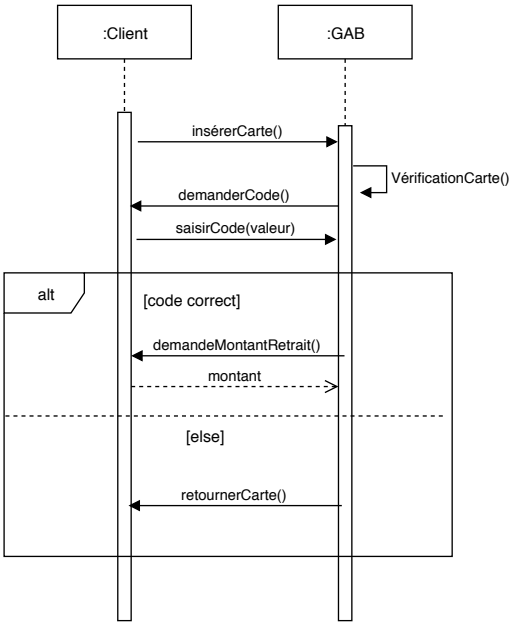


FIGURE 5.9 – Exemple d'utilisation du fragment combiné alt

FIGURE 5.10 – Représentation graphique du fragment combiné opt

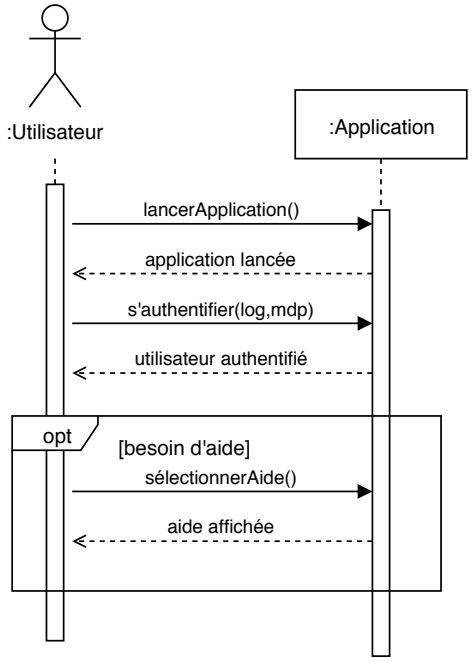


FIGURE 5.11 – Exemple d'utilisation du fragment combiné opt

L'opérateur loop

Cet opérateur est utilisé pour modéliser un ensemble d'interactions qui s'exécutent en boucle. loop est suivi des paramètres *min*, *max* et d'une condition de garde.

Le contenu du fragment est exécuté *min* fois avant que la condition de garde ne soit vérifiée puis tant que la condition est vraie et que le nombre d'itération ne dépasse pas *max*.

Chaque paramètre est optionnel. Il existe d'ailleurs, plusieurs notations possibles :

- loop(valeur) est équivalent à loop(valeur,valeur) et signifie que la séquence s'exécute valeur fois.

- loop est équivalent à loop(0,*), où « * » signifie illimité.

Exemple :

On reprend le même exemple du guichet automatique bancaire avec un client qui se trompe en saisissant son code mais en lui donnant cette fois-ci la possibilité de saisir son code jusqu'à 3 fois si jamais il se trompe. Cela donne (voir figure 5.12) :

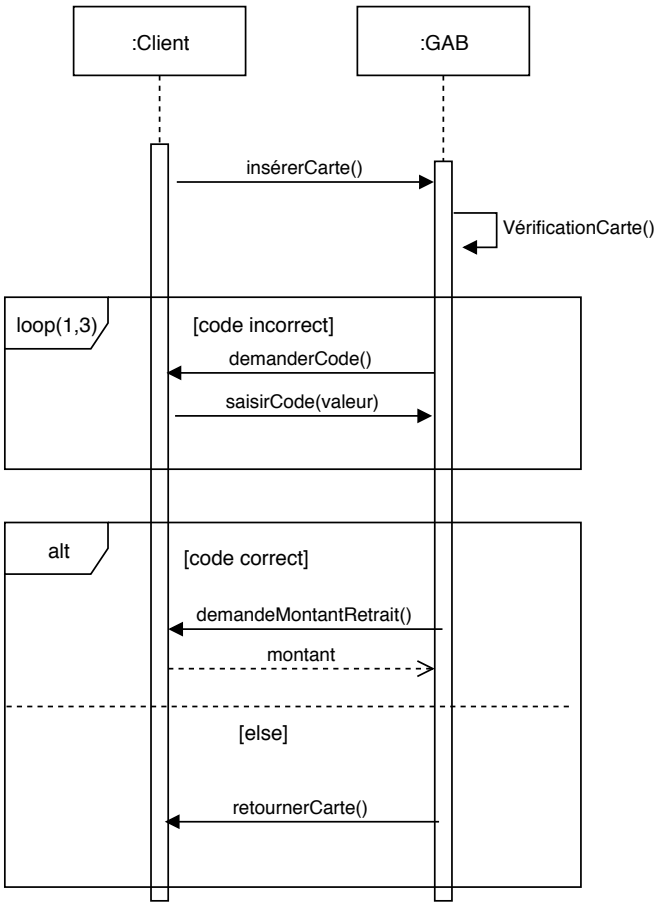


FIGURE 5.12 – Exemple d'utilisation du fragment combiné loop

L'opérateur parallel (ou par)

L'opérateur « par » permet d'envoyer des messages en parallèle et permet donc de simuler une exécution parallèle.

Cet opérateur possède au moins deux sous-fragments exécutés simultanément.

Un exemple d'utilisation de ce fragment combiné est illustré par la figure 5.13.

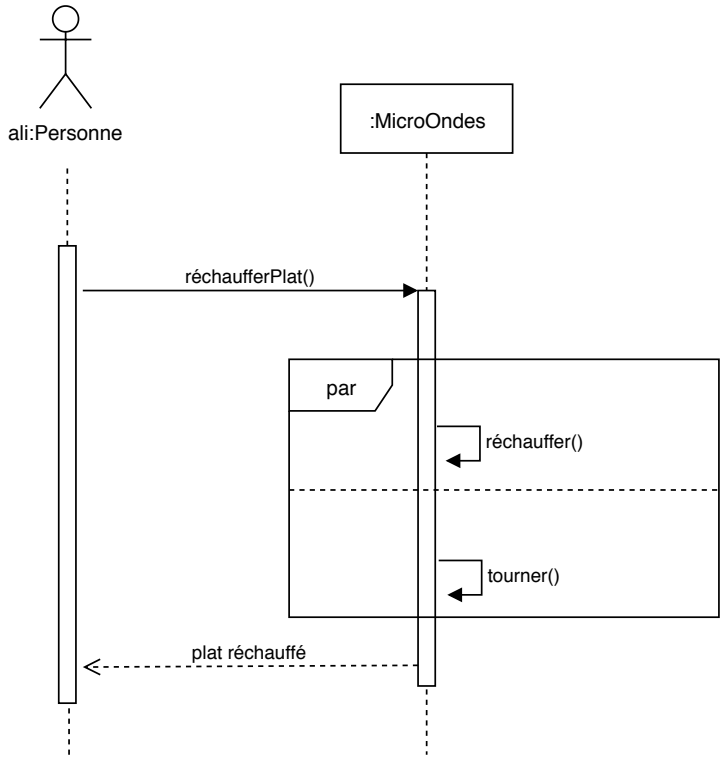


FIGURE 5.13 – Exemple d'utilisation du fragment combiné par

5.4 Diagramme de communication

Les diagrammes de communication (anciennement appelés diagrammes de collaboration en UML 1) et les diagrammes de séquence sont équivalents.

Néanmoins, contrairement à un diagramme de séquence, un diagramme de communication ne dispose pas d'une ligne de temps explicite et met davantage l'accent sur l'organisation spatiale des participants à l'interaction. Il est souvent utilisé pour illustrer un cas d'utilisation ou pour décrire une opération.

Le diagramme de communication comme le diagramme de séquence aide à valider les associations du diagramme de classe en les utilisant comme support de transmission des messages. Un exemple est illustré par la figure 5.14.

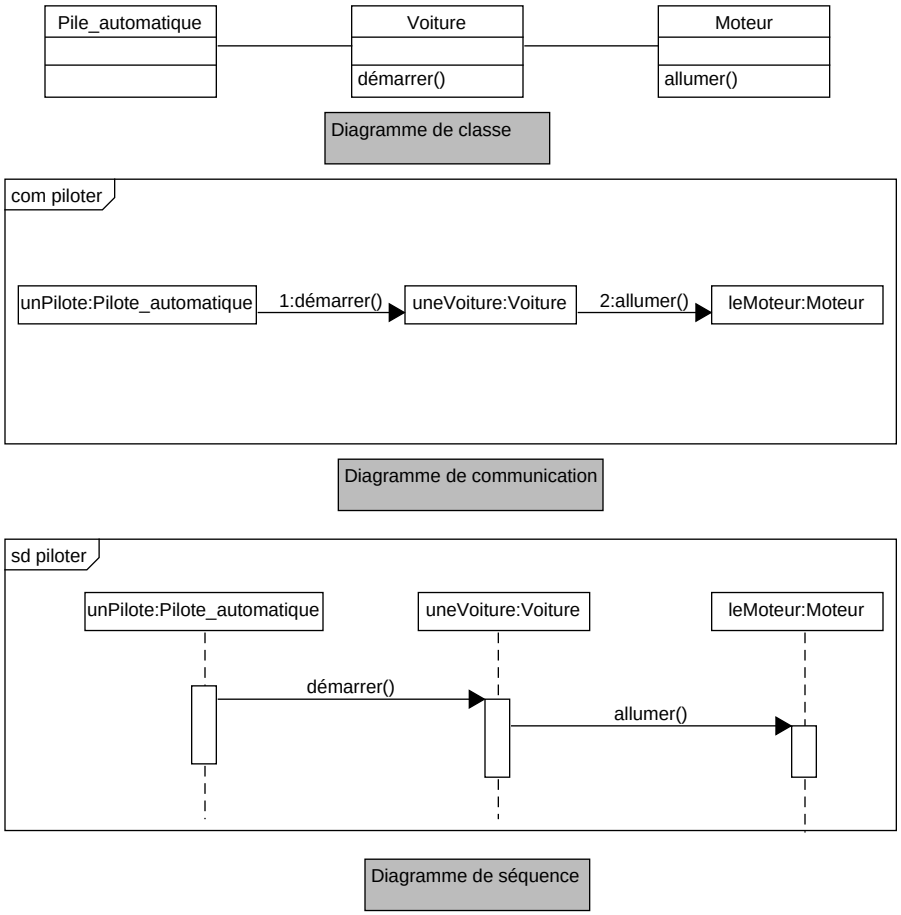


FIGURE 5.14 – Relation entre diagramme de classe et diagrammes d'interaction

5.4.1 Eléments constitutifs d'un diagramme de communication

Les diagrammes de communication s'appuient sur les éléments constituant suivants (figure 5.15) :

— les lignes de vie
— les connecteur
— les messages

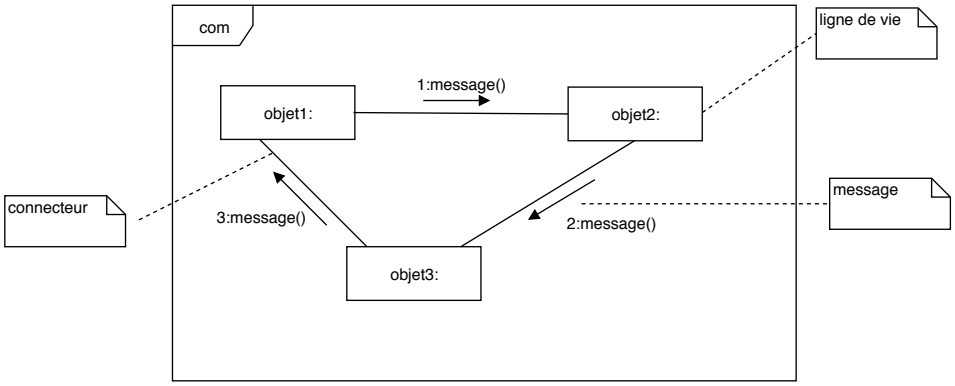


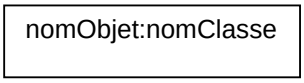
FIGURE 5.15 – Les éléments constitutifs d'un diagramme de communication

UML 2 permet de disposer un diagramme de communication à l'intérieur d'un cadre d'interaction « com » comme c'est le cas avec le diagramme de séquence.

5.4.1.1 Les lignes de vie

Les lignes de vie sont représentées par des rectangles contenant une étiquette dont la syntaxe est : *[<nom_du_role>] : [<Nom_du_type>]*

Au moins un des deux noms doit être spécifié dans l'étiquette, les deux points (:) sont, quant à eux, obligatoires.



La représentation graphique d'une ligne de vie dans un diagramme de communication est proche de cette d'un diagramme de séquence à la différence qu'il n'y a pas de ligne vertical et que les lignes de vie peuvent être disposés librement (voir figure 5.16).

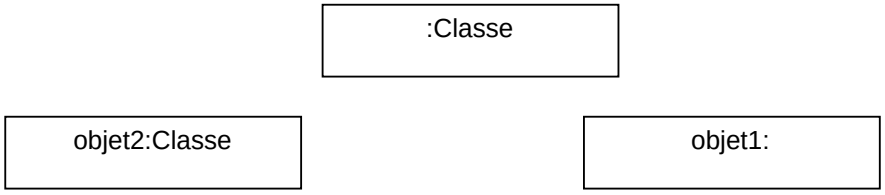


FIGURE 5.16 – Différentes représentations de lignes de vie dans un diagramme de communication

5.4.1.2 Les connecteurs

Les relations entre les lignes de vie sont appelées connecteurs et se représentent par un trait plein reliant deux lignes de vie.

Plusieurs messages, et ce dans les deux sens, peuvent circuler sur le même connecteur. Il n'y a pas un connecteur par message : tous les messages empruntent le même connecteur.

5.4.1.3 Les messages

Dans un diagramme de communication, un message est représenté par une expression séquencée et une flèche en dessous. Comme pour le diagramme de séquence, on distingue des messages synchrones et des messages asynchrones. Néanmoins, les messages de retour d'appel ne sont généralement pas représentés. Par ailleurs, les messages sont ordonnés selon un numéro de séquence croissant.

Syntaxe des messages

Chaque message entre objets est représenté par une expression, une flèche indiquant sa direction, et un numéro indiquant sa place dans la séquence. Les flèches représentant les messages sont tracées à côté des connecteurs qui les supportent. Il faut bien faire la distinction entre les messages et les connecteurs : on pourrait avoir un connecteur sans message mais jamais l'inverse.

Dans un diagramme de communication les messages ont la syntaxe suivante :

$num - seq[expression] : message(param)$

où,

num-seq : numéro de séquence du message
expression : choix ou itération
message : nom du message
param : paramètre du message

Numérotation des messages

La numérotation des messages à l'aide de numéros de séquence permet de représenter l'ordre chronologique des messages.

Des messages successifs sont ordonnés selon un numéro de séquence croissant.

La numérotation des messages permet également d'exprimer que des appels de méthode se font à l'intérieur d'une méthode. On dit alors que la numérotation est pointée et on parle également d'une numérotation hiérarchique.

Par exemple, les messages 1.1 et 1.2 sont appelés à l'intérieur de l'appel du message 1.

Enfin, lorsque des messages sont envoyés en parallèle, il est également possible d'exprimer cette simultanéité à l'aide de lettres.

Par exemple 1.a et 1.b sont des messages envoyés simultanément en réponse à un message dont le numéro est 1.

Les messages permettent également d'exprimer le choix et l'itération comme suit.

Le choix

Choix : [garde] message où,

— La garde est une condition booléenne.
— Le message n'est envoyé que lorsque la garde est vraie

L'itération

Itération : *[clauseItération] message où,

— La clause d'itération peut être exprimée dans le format i :=1..n
— Si c'est une condition booléenne le message est envoyé tant que la condition est vraie.

Quelques exemples

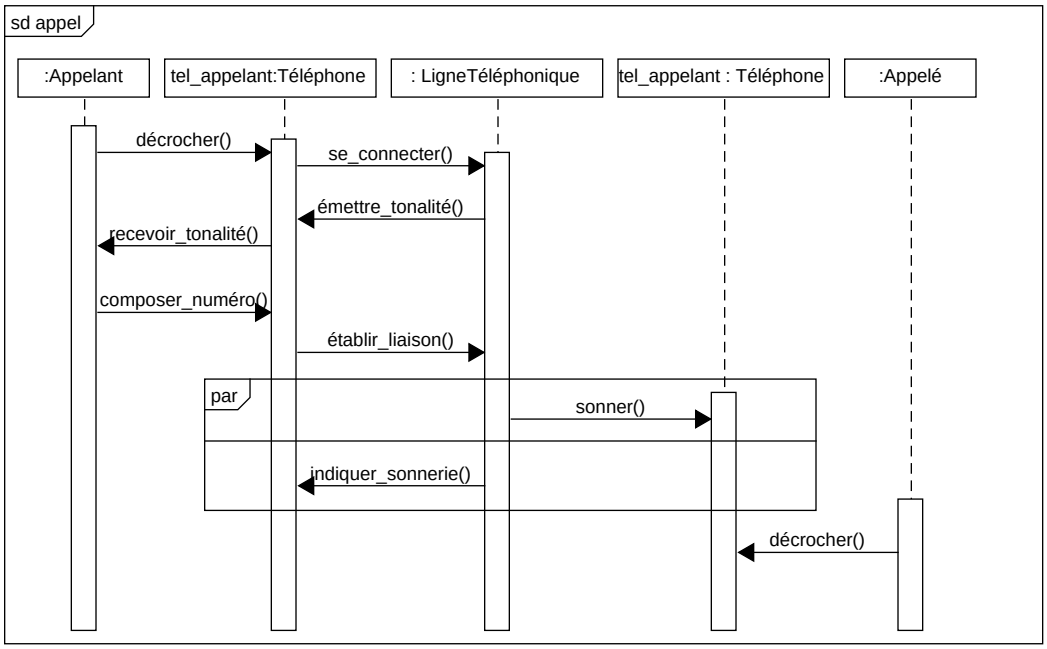
- 2 : affiche(x,y) : message simple avec 2 paramètres x et y.
- 1.3 : trouve() : appel emboîté à l'intérieur d'un message portant le numéro de séquence 1.
- 4 [x < 0] : inverse(x, couleur) : Envoi conditionnel du message avec paramètre portant le numéro de séquence 4 qui n'est émis que si la condition x < 0 est vraie.
- 3.1 *[i :=1..10] : recommencer() : message sans paramètre portant le numéro de séquence 3.1 et qui est émis 10 fois.

5.4.2 Équivalence des diagrammes de séquence et de communication

Grâce à la numérotation des messages et à leurs syntaxes, le diagramme de communication peut facilement être exprimé à l'aide d'un diagramme de séquence. Ces diagrammes deviennent équivalents. Le diagramme de séquence offre néanmoins, une plus grande richesse dans la notation.

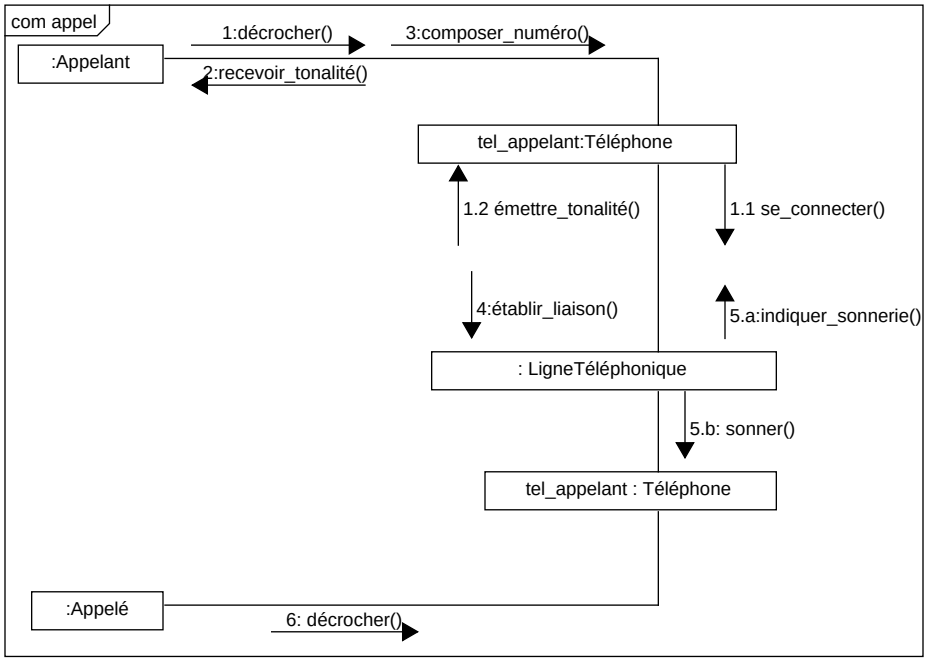
Exemple :

Soit le diagramme de séquence suivant :



En déduire un diagramme de communication équivalent.

Solution :



5.5 Comparatif entre les diagrammes de séquence et de communication

Les diagrammes de séquence et de communication ont tous deux des avantages qui leur sont propres. Dans une démarche de modélisation, il n'y a donc pas un choix « meilleur » que l'autre dans l'absolu et chaque modélisateur aura ses propres préférences.

Néanmoins, la spécification UML est plus centrée sur les diagrammes de séquence que sur les diagrammes de communication. En effet, davantage de réflexion et d'efforts ont été consacrés aux diagrammes de séquence, à leur notation et à leur sémantique. Ceci se reflète également sur de nombreux outils qui prennent mieux en charge ce diagramme et offre plus d'options de notation que pour le diagramme de communication.

Une analyse stricto sensu de ces diagrammes permet de constater que :

— Les diagrammes de séquence permettent une meilleure visualisation des flots d'appels, parce qu'on les lit simplement de haut en bas.
— Les diagrammes de séquence sont excellents pour les besoins de la documentation ou pour suivre facilement le flot d'appels généré automatiquement par la rétro ingénierie.
— Les diagrammes de communication permettent quant à eux de retrouver la séquence des appels en se référant à leur numérotation, comme « 1 », « 2 », ..
— Les diagrammes de communication sont intéressants lorsqu'on applique UML « en mode esquisse », en modélisant au tableau par exemple, parce qu'ils permettent beaucoup mieux d'exploiter l'espace.
— Les diagrammes de communication offrent la possibilité d'ajouter/supprimer des boîtes n'importe où sur l'axe horizontal et vertical. Ceci a son importance vue la fréquence des changements intervenant lors de la conception.
— Au contraire, dans les diagrammes de séquence, les nouveaux objets doivent être ajoutés à droite, ce qui impose des limites car le bord droit de la page (ou du tableau) est rapidement épuisé (tandis que de l'espace vertical est disponible).

Pour résumer, les points positifs et négatifs de ces deux diagrammes sont présentés dans le tableau 5.1.

Diagramme	forces	Faiblesses
Séquence	- Indique clairement la séquence et l'ordonnancement des messages. - Grande richesse de la notation.	- Ajout de nouveaux objets s'effectuant obligatoirement vers la droite ; consomme trop d'espace horizontal.
Communication	- Économique en terme d'espace, permet d'ajouter des objets dans les deux dimensions.	- Rend plus difficile la lecture des séquences de messages. - Moins d'options de notation.

TABLE 5.1 – Comparatif entre diagrammes de séquence et de communication

5.6 Éléments de méthodologie

Les diagrammes de séquence et de communication sont des diagrammes dynamiques et d'interaction. Il est donc important d'identifier les objets qui interagissent selon un scénario bien déterminé car on ne peut pas décrire à travers un seul diagramme (qu'il soit de séquence ou de communication l'ensemble des interactions).

Le processus d'élaboration de ces diagrammes étant itératifs, il est important de considérer au départ, les interactions les plus importantes puis d'enrichir au fur et à mesure les diagrammes à travers l'identification de nouveau objets ou encore d'échange de messages plus précis (identification de paramètres et de leur valeurs,...).

5.7 Conclusion

La modélisation de la dynamique d'un système permet de mettre en avant son évolution dans le temps à travers l'évolution des objets qui le composent et leurs interactions. Dans une démarche de construction d'un logiciel, il est important de connaître le système et de comprendre de façon détaillé son fonctionnement avant de pouvoir s'atteler à décrire sa dynamique. C'est pour ces raisons, que dans une telle démarche, on préfèrera d'abord se focaliser sur la description de l'architecture du système et des éléments qui le composent avant de poursuivre la description de l'évolution de ces composants et de leurs interactions dans le temps. Dans ce chapitre, nous nous sommes intéressés aux diagrammes d'interaction à travers l'étude des diagrammes de séquence et de communication.

Les diagrammes d'interaction proposés dans UML 2 permettent de mettre en évidence le comportement du système à travers les interactions des différents objets qui le composent. Ces diagrammes sont très utiles pour modéliser des scénarios de cas d'utilisation par exemple ce qui correspond à une vision très réaliste du comportement du système à travers ces interactions.

Les diagrammes de séquence et de communication possèdent tous deux leurs points forts : il n'y a pas de choix « correct » dans l'absolu et chaque modélisateur aura ses propres préférences. Néanmoins, davantage de réflexion et d'efforts ont été consacrés aux diagrammes de séquence, à leur notation et à leur sémantique. En conséquence, les options de notation sont plus nombreuses et les outils les prennent mieux en charge.

5.8 Exercices

Exercice 1 :

On s'intéresse à la gestion de l'activité commerciale d'une société et plus particulièrement à la gestion des marchés privilégiés (contrats que la société passe avec ses gros clients). L'intérêt de ces marchés privilégiés est de faire profiter les clients importants ou les clients stables de rabais sur leurs achats.

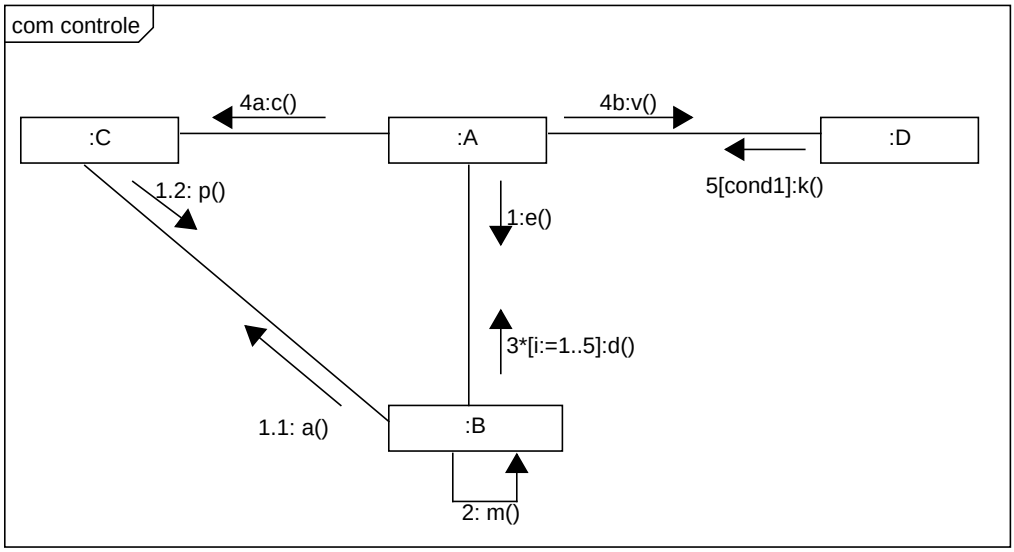
Le scénario de gestion d'un marché privilégié est décrit comme suit :

Lorsqu'un client désire bénéficier d'un marché, il envoie une demande de marché qui sera contrôlée par un ingénieur commercial. L'ingénieur vérifie l'existence et la solvabilité du client auprès des instances compétentes (base de données et banque respectivement) : si ces conditions ne sont pas remplies, la demande est refusée. Dans le cas contraire, l'ingénieur complète la demande et l'envoie au directeur régional, qui, après avoir étudié la demande, donne son avis à l'ingénieur commercial. Dans le cas où cet avis est positif, l'ingénieur envoie une proposition de marché au client qui va l'étudier. Si cette proposition est acceptée par le client, elle est signée et envoyée à l'ingénieur et le marché est créé par l'ingénieur.

1. Réaliser un diagramme de séquence

Exercice 2 :

Soit le diagramme de communication suivant :



1. Réaliser un diagramme de séquence équivalent

Exercice 3 :

Une entreprise a mis en œuvre une application où des gens peuvent s'inscrire pour faire des échanges de services gratuitement (sans payer).

Par exemple, quelqu'un sait parler en français et veut apprendre l'anglais et un autre le contraire. Ils peuvent alors, grâce à l'application, s'échanger des cours sans aucun frais.

Un autre exemple, quelqu'un sait cuisiner et a besoin de faire du bricolage et un autre est nul en cuisine mais très fort en bricolage. Ils peuvent aussi faire un échange de services.

Hanene s'est inscrite au niveau de l'application d'échange de services et commence à l'utiliser comme suit :

Elle ouvre l'application et reçoit une demande d'authentification pour saisir son login et son mot de passe. Elle saisit son login et son mot de passe. L'application vérifie alors que le login et le mot de passe sont corrects auprès d'un serveur d'authentification. Si c'est le cas, le serveur d'authentification indique à l'application que Hanene est bien « valide ». L'application ouvre alors une session en envoyant un signal à l'utilisateur pour lui signaler qu'elle est en attente d'une commande de sa part.

Hanene lance alors une recherche pour trouver quelqu'un qui sait utiliser l'outil GanttProject pour la gestion de projet. Si elle trouve un utilisateur qui connaît cet outil, elle lui envoie une demande à l'aide de l'application. Ce dernier consulte la demande de Hanene. S'il est intéressé par un des services proposés par Hanene, il accepte sa demande.

1. Modéliser ce scénario à l'aide d'un diagramme de séquence.

Chapitre 6

Diagrammes UML d'états-transitions : vue dynamique

La seconde catégorie de diagrammes modélisant la dynamique du système est celle des diagrammes d'états-transition. Ces diagrammes sont utilisés en particulier pour représenter la dynamique de certains objets participant au système. Ils font également partie des diagrammes comportementaux dans UML 2.

Néanmoins, contrairement aux diagrammes d'interaction qui n'offrent qu'une vision partielle du système puisqu'ils ne s'intéressent qu'à un scénario donné, le diagramme d'états-transitions offre une vue plus complète des comportements de l'élément auquel il est attaché.

Utilisés pour modéliser le cycle de vie des objets, ils permettent de modéliser le comportement interne d'un objet à l'aide d'un automate à états finis déterministe.

L'automate permet une représentation des changements d'états d'un objet en réponse à des évènements (interaction avec d'autres objets ou avec des acteurs).

6.1 Définition

Un diagramme d'états-transitions est un automate à états finis déterministe. Ce diagramme UML s'appuie sur les Statecharts de David Harel [1]. Les automates à états finis sont des graphes d'états, reliés par des arcs orientés qui décrivent les transitions. Concrètement, un diagramme d'états-transitions est une machine dont le comportement des sorties ne dépend pas seulement de l'état de ses entrées, mais aussi d'un historique des sollicitations passées.

Le modèle dynamique du système comprend plusieurs diagrammes d'états-transitions. Généralement, un diagramme d'états-transitions est construit pour chaque classe dont le comportement est fortement dynamique. L'état d'un objet à un instant donné se traduit par les valeurs de ses propriétés à cet instant. Le passage d'un état à un autre est appelé transition. Ces transitions peuvent être déclenchées par des événements Les états et les transitions sont donc les principaux éléments constitutifs d'un diagramme d'états-transitions (figure 6.1).

1. Harel, D. 1987. Statecharts : A Visual Formalism for Complex Systems. Science of Computer Programming vol.8

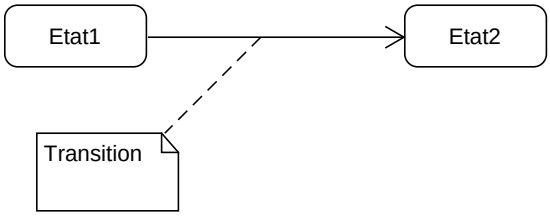


FIGURE 6.1 – Représentation d'un diagramme d'états-transitions

6.2 Intérêts des diagrammes d'états-transitions

Les diagrammes d'états-transitions font partie des diagrammes comportementaux. Ces diagrammes ont pour avantage de montrer l'évolution d'un objet du système dans le temps et face à la présence de stimulis.

Ils permettent pour cela de représenter les différentes situations (états) dans lesquels peut se trouver un objet ainsi que la manière dont cet objet passe (ou transite) d'un état à un autre en réponse à des évènements identifiés au préalable.

6.3 Les états

Un état représente une période dans la vie d'un objet pendant laquelle ce dernier est soit en train d'accomplir une certaine activité, soit attend qu'il se produise un certain événement.

Un état se caractérise par sa durée qui est finie et qui varie selon la vie de l'objet.

Le cycle de vie d'un objet correspond à une succession d'états qu'il faut identifier afin de ne conserver que les états significatifs.

En UML, un état, qui n'est ni initial ni final est représenté par un rectangle dont les coins sont arrondis (voir figure 6.2).

Un état peut être partitionné en plusieurs compartiments séparés par une ligne horizontale. Le premier compartiment contient le nom de l'état et les autres peuvent recevoir des transitions internes (cf. section 6.5), ou des sous-états (cf. section 6.7), quand il s'agit d'un état composite. Dans le cas d'un état simple (i.e. sans transition interne ni sous-états), on peut omettre toute barre de séparation.

6.3.1 État initial et état final

En plus de la succession d'états intermédiaires correspondant au cycle de vie d'un objet, le diagramme d'états-transitions comprend généralement deux pseudo-états : l'état initial et l'état final.

L'état initial indique l'état de départ du diagramme d'états-transitions et qui correspond à la création de l'instance.

L'état final du diagramme d'états correspond à la destruction de l'instance.

Les différentes représentations d'états sont illustrées par la figure 6.2.

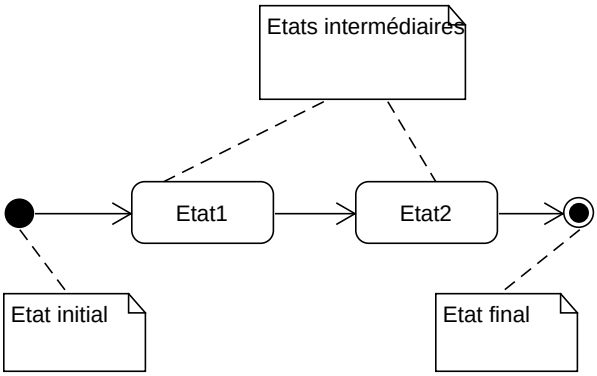


FIGURE 6.2 – Représentations des états

Remarque :

Un diagramme d'états-transitions a toujours un et un seul état initial pour un niveau hiérarchique donné, par contre, il peut n'avoir aucun état final comme il peut en avoir plusieurs.

6.4 Les transitions

Dans un diagramme d'états-transitions, une transition relie deux états et indique que l'objet change d'état. Elle représente le passage instantané d'un état (source) vers un autre état (cible). Une transition n'a donc pas de durée. Si la transition est réflexive on parle d'auto-transition ou de transition propre. Une transition est représentée par une flèche orientée de l'état source vers l'état cible.

En règle générale, une transition possède un événement déclencheur, une condition de garde et un effet. Une transition est souvent déclenchée par un évènement spécifique. Néanmoins, une transition peut s'opérer sans que l'évènement qui la déclenche ne soit spécifié, on parle alors d'une transition automatique (6.3).

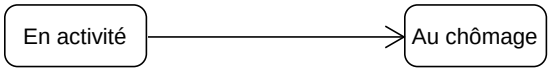


FIGURE 6.3 – Exemple de passage instantané d'un état à un autre

L'évènement déclencheur peut être spécifié avec une condition de garde et les effets qui en résultent selon la syntaxe suivante : événement [garde] / effet

Chaque partie étant optionnelle.

6.4.1 Les évènements

Les diagrammes d'états-transitions permettent de spécifier les réactions d'une partie du système à des événements discrets. Un évènement peut se produire pendant l'exécution d'un système entraînant des changements. Ils sont donc intéressants à modéliser.

Un événement se produit à un instant précis et est dépourvu de durée.

Exemple 1 :

L'exemple illustré par la figure 6.3 montre que c'est l'évènement « perte d'emploi » qui cause la transition de l'état *en activité* à l'état *au chômage*. Inversement, l'évènement « embauche » actionne une transition de l'état *au chômage* à l'état *en activité* (voir figure 6.4).

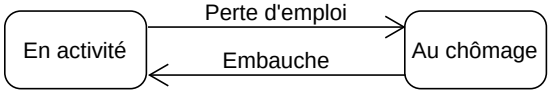


FIGURE 6.4 – Exemple 1 d'une transition causée par un évènement

Exemple 2 :

Considérons, cette fois, une lampe munie de deux boutons poussoirs On et OFF. La réaction de l'éclairage associé dépendra de son état courant (ou historique) : si la lumière est allumée, elle s'éteindra, si elle est éteinte, elle s'allumera. Le digramme d'états-transitions qui modélise ce fonctionnement est illustré par la figure6.5.

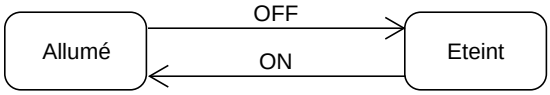


FIGURE 6.5 – Exemple 2 d'une transition causée par un évènement

6.4.1.1 Les différents types d'évènements

Dans un diagramme d'états-transitions, on distingue quatre sortes d'événements :

1. Événement signal (signal event) : causé par la réception d'un signal (message asynchrone).
2. Événement appel (call event) : causé par la réception d'un appel d'opération. L'événement d'appel est en général synchrone
3. Événement temporel (time event) : causé par l'expiration d'un temporisateur. Ex : after (3 seconds)

4. Événement de changement (change event) : généré par la satisfaction (i.e. passage de faux à vrai) d'une expression booléenne sur des valeurs d'attributs. Ex : when (age>60)

Les figures 6.6 et 6.7 illustrent quelques exemples d'utilisation de ces évènements.

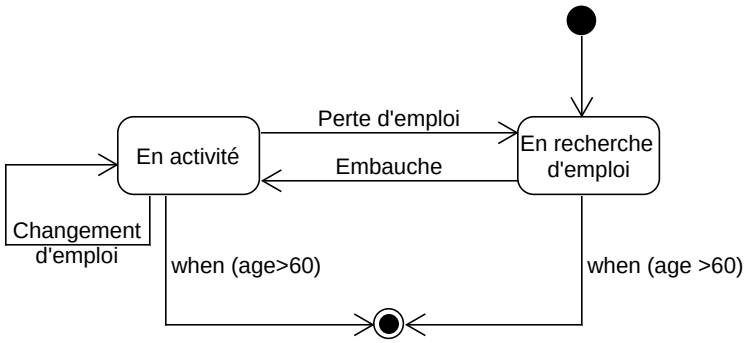


FIGURE 6.6 – Exemple de transitions causées par un évènement de changement

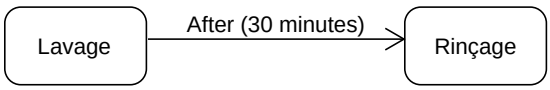


FIGURE 6.7 – Exemple de transition causée par un évènement temporel

6.4.2 Condition de garde

Il est possible d'exprimer des conditions dont dépend le déclenchement d'une transition lors de l'occurrence d'un événement.

Ces conditions sont appelées des gardes et sont notées entre crochets.

La condition de garde est évaluée uniquement lorsque l'événement déclencheur se produit. Si l'expression est fausse à ce moment-là, la transition ne se déclenche pas, si elle est vraie, la transition se déclenche et ses effets se produisent. Un exemple est illustré par la figure 6.8.

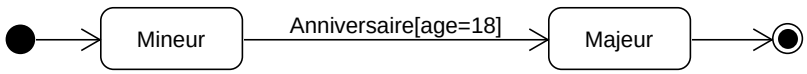


FIGURE 6.8 – Exemple de transition causée par un évènement conditionnel

6.4.2.1 Différence entre condition de garde et évènement de changement

La condition de garde est évaluée une fois que l'événement déclencheur de la transition a lieu et que le destinataire le traite. Si elle est fausse, la transition ne se déclenche pas et la condition n'est pas réévaluée.

L'événement de changement est, quant à lui, évalué continuellement jusqu'à ce qu'il devienne vrai, et c'est à ce moment-là que la transition se déclenche.

6.4.3 Les effets

Une transition peut spécifier un comportement optionnel réalisé par l'objet lorsque la transition est déclenchée. Ce comportement est appelé « effet » en UML 2. Cela peut être une simple action ou une succession d'actions (activité)

6.4.3.1 Les actions

Une action est une opération instantanée et atomique donc ininterruptible. Elle peut représenter la mise à jour d'un attribut, un appel d'opération, la création ou la destruction d'un objet ou encore l'envoi d'un signal à un objet

6.4.3.2 Les activités

Une activité représente une opération qui nécessite un certain temps d'exécution. Elle peut être interrompue à tout moment par un événement générant une transition. Une activité est associée à un état mais un état peut ne pas avoir d'activité.

Un état peut avoir une activité interne. Si c'est le cas, il est séparé en deux compartiments : le nom de l'état dans le premier compartiment et les activités internes dans le second compartiment.

6.5 Dynamique d'un état

La dynamique d'un état est modélisée par les actions ou les activités propres à cet état.

Ces dernières peuvent être renseignées directement à l'intérieur de l'état. L'état possèdera alors deux compartiments. Un premier compartiment contenant le nom de l'état, un second compartiment contenant les actions/activités internes à l'état.

Les actions/activités internes sont déclenchées par des évènements internes prédéfinis correspondant à des déclencheurs particuliers :

« entry »/action : action exécutée à l'entrée de l'état

« exit »/action : action exécutée à la sortie de l'état

« do »/activité : action durable, récurrente et significative exécutée dans l'état

« on » évènement/action : action exécutée à chaque fois que l'évènement cité survient. L'objet reste dans l'état courant.

Remarque :

Contrairement aux évènements externes, un évènement interne spécifié par le mot clé « on » ne change pas l'état courant de l'objet. Si l'évènement spécifié par « on » survient, les actions liées au déclencheurs « entry » et « exit » ne sont donc pas exécutées. Par contre, en cas de transition propre l'objet change d'état (même si l'état cible est le même que l'état source). Dans ce cas, on effectue les actions correspondant à une sortie puis les actions correspondant à une entrée dans l'état.

Exemple 1 :

On reprend l'exemple illustré par la figure 6.8 mais en supposant cette fois-ci qu'à sa majorité, une personne peut voter. On modélise alors cela par l'utilisation de l'évènement interne « on » à l'intérieur de l'état *Majeur* comme cela est illustré par la figure 6.9.

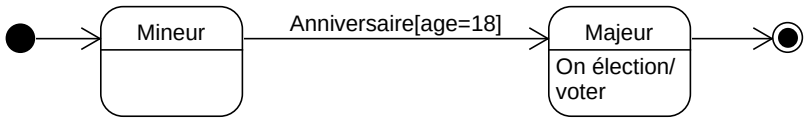


FIGURE 6.9 – Exemple 1 d'utilisation d'un évènement interne

Exemple 2 :

Si on considère un terminal bancaire pour le retrait d'argent. Lorsque le terminal entre en mode « saisie de mot de passe », l'affichage des caractères est désactivé, puis réactivé ensuite. Le diagramme d'états-transitions obtenu est illustré par la figure 6.10.

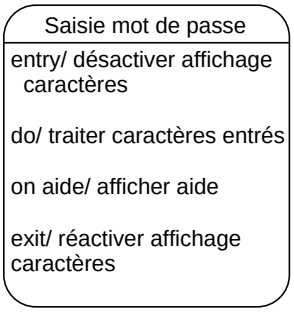


FIGURE 6.10 – Exemple 2 d'utilisation d'un évènement interne

6.5.1 L'ordonnancement des effets

L'ordre d'exécution des actions et activités d'un état donné est fixé, selon le type d'événement déclencheur, comme suit :

En entrée : On commence par réaliser l'action sur la transition d'entrée, puis l'action d'entrée, puis l'activité associée à l'état

En interne : On commence par interrompre l'activité en cours, on exécute l'action interne, puis on reprend l'activité. Le contexte de l'activité est sauvegardé lors de son interruption, en vue de la reprise.

En sortie : on commence par interrompre l'activité en cours, puis on exécute l'action de sortie, puis l'action sur la transition de sortie. Cette fois, le contexte de l'activité n'est pas sauvé lors de son interruption

Auto-transition (transition propre) : On commence par interrompre l'activité en cours, puis on exécute :

1. l'action de sortie
2. l'action associée à la transition propre
3. l'action d'entrée
4. l'activité associée à l'état.

Exemple :

Donner l'ordre d'exécution des actions de l'état x (voir figure 6.11) selon les différents scénarios (en entrée, en interne, en sortie, auto-transition)

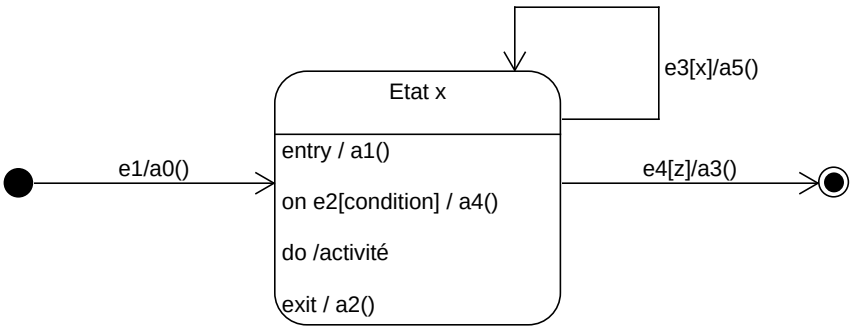


FIGURE 6.11 – Exemple d'ordonnancement des effets

6.6 Transition composite

Il est possible de représenter des alternatives pour le franchissement d'une transition en utilisant des pseudo-états particuliers : les points de jonction et les points de décision.

6.6.1 Point de jonction

Un point de jonction est représenté par un cercle plein.

Il permet à plusieurs transitions d'avoir une partie commune en partageant des segments de transition. L'utilisation de points de jonction a pour but de rendre la notation des transitions alternatives plus lisible comme peut l'illustrer l'exemple de la figure 6.12.

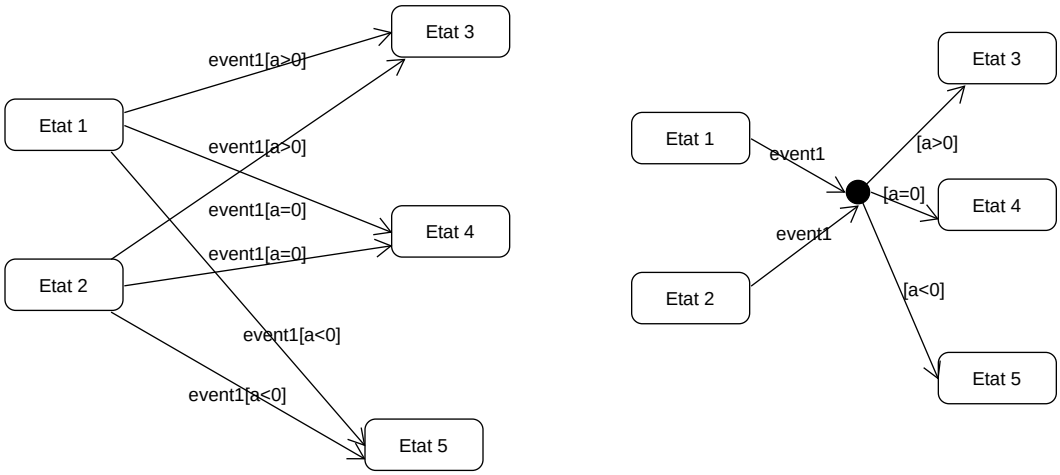


FIGURE 6.12 – Exemple d'un diagramme sans et avec un point de jonction

6.6.2 Point de choix

Un point de décision est représenté par un losange.

Il a un fonctionnement similaire à celui du point de jonction mise à part que nous pouvons sortir de l'état d'origine dès que le segment avant le point de décision est franchissable (même si aucun des segments après le point de décision n'est franchissable).

Le choix du segment à franchir derrière le point de décision se fait au moment de l'arrivée sur le point de décision. Cela permet aux segments qui sont derrières le point de décision d'avoir une condition de garde qui dépend d'éléments qui sont définis par une activité effectuée lors du franchissement du segment de transition avant le point de décision.

Il est important de noter que lorsque nous arrivons sur un point de décision, il faut qu'un des segments de transition qui suit, soit franchissable (une transition n'ayant pas de durée, nous ne pouvons pas rester à attendre quelque chose au niveau du point de décision).

Pour éviter ce problème nous pouvons ajouter un segment avec la garde [else] qui est automatiquement franchie si aucune des gardes des autres segments n'est vraie.

la figure 6.13 illustre un exemple d'utilisation d'un point de décision.

6.7 État composite

Certains états sont complexes et correspondent à la réalisation de plusieurs activités (séquentielles ou simultanées) qui ne pourront pas être définis par des transitions internes. Il peut alors être intéressant de les décomposer en sous-états. On parle alors d'état composite.

Un état composite est un état qui peut être décomposé en sous états (décomposition) c'est-à-dire qu'il contient des sous-états. Ce genre d'état permet de hiérarchiser les états et de structurer les comportements complexes tout en occultant certains détails et en factorisant les actions.

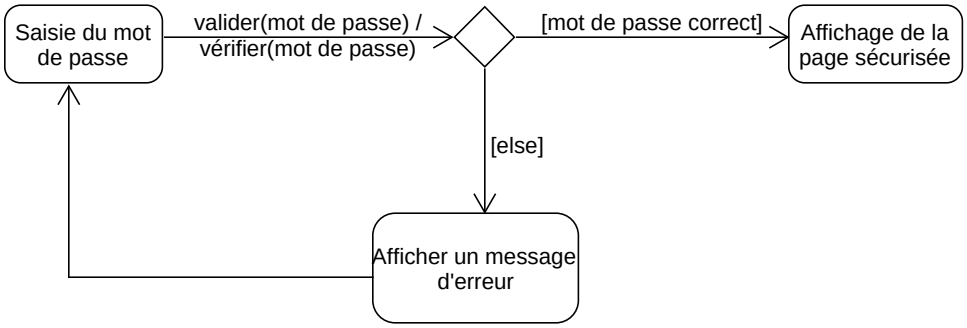


FIGURE 6.13 – Exemple d'utilisation d'un point de décision

Ces sous-états sont séquentielles (ou disjoints) lorsqu'un l'objet doit être dans un seul sous-état à la fois. On parle alors de décomposition disjonctive (voir figure 6.14).

Ils sont concurrents (ou simultanées) lorsque l'objet doit être simultanément dans plusieurs sous-états. On parle alors de décomposition conjonctive (voir figure 6.15).

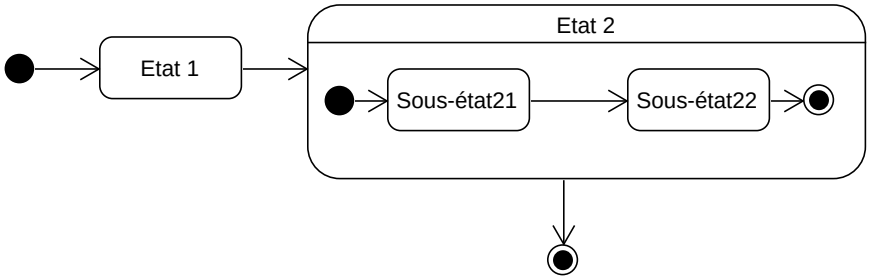


FIGURE 6.14 – Exemple de décomposition avec sous-états séquentiels

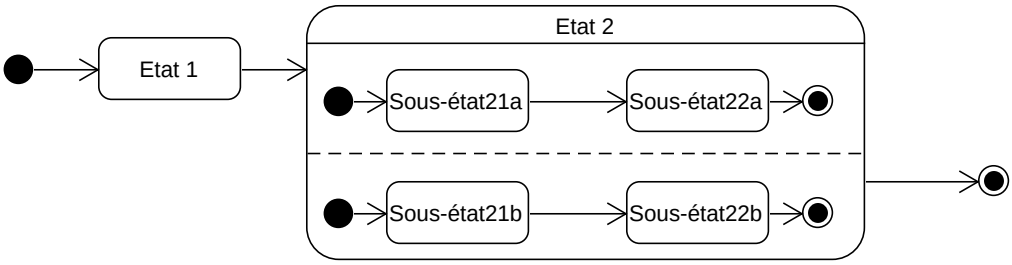


FIGURE 6.15 – Exemple de décomposition avec sous-états concurrents

Il existe une notation abrégée d'un état composite qui permet de masquer les sous-états. Placer le symbole (o-o) à l'intérieur d'un état composite permet, en effet, d'éviter de surcharger le diagramme d'états-transitions. Le contenu de l'état composite pourra être spécifié ailleurs par la suite (voir figure 6.16).

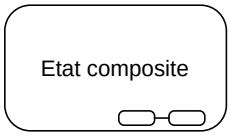


FIGURE 6.16 – Notation abrégée d'un état composite

6.7.1 États disjoints et transitions

Dans le cas d'une décomposition disjointe, les transitions qui ont pour cible la frontière d'un état composite sont équivalentes à une transition ayant pour cible l'état initial de l'état composite. Cela implique que dès qu'un état composite est actif, il active son sous-état initial.

Chaque transition de sortie ayant pour source la frontière de l'état composite s'applique à tous les sous-états.

Si un état composite est raccordé à une transition automatique, elle est franchie lorsque nous atteignons le sous-état final de l'état composite.

Si une des transitions externes attenantes à l'état composite est franchissable alors elle est franchie et tous les sous-états deviennent inactifs.

Exemple :

Soit la figure 6.17 qui illustre un diagramme d'états-transitions composé des états composites Etat1 et Etat2.

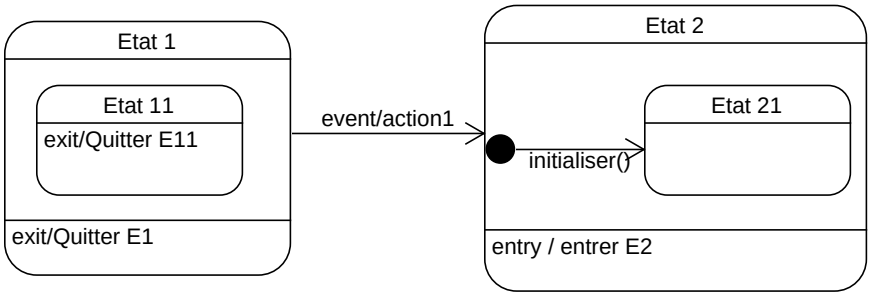


FIGURE 6.17 – Exemple d'états disjoints et de transitions

Depuis Etat11, quand event survient :
- On produit la séquence d'activités QuitterE11, QuitterE1, action1, EntrerE2, initialiser()
- L'objet se trouve alors est dans Etat21.

6.7.1.1 État Historique

Nous avons vu que lorsqu'une transition raccordée à un état composite est franchissable alors elle est franchie même si les activités en cours (sous-états) ne sont pas terminées. Si nous

désirons qu'un état composite reprenne son activité à l'endroit où elle s'était interrompue lors de la précédente activation de l'état, il faut lui définir un nouveau sous état d'entrée que nous appelons état historique et que nous désignons par H ou H* :

H : pour reprendre au début du sous-état du plus haut niveau dans lequel nous nous étions arrêté.

H* : désigne un historique profond qui est valable quel que soit le niveau d'imbrication.

Exemple : Lavage automatique d'une voiture

Dans l'état composite *Lavage automatique*, le client peut appuyer sur le bouton d'arrêt d'urgence que l'on soit en phase de lavage ou en phase de séchage (ces deux phases englobent eux-mêmes d'autres états). La machine se met alors en attente. Le client a alors 2 min pour reprendre le lavage ou le séchage exactement là où le programme a été interrompu c'est-à-dire au niveau du dernier sous-état actif des états de lavage ou de séchage. Il est donc intéressant d'utiliser ici un état historique profond afin de reprendre l'exécution là où elle s'était arrêtée. En effet, si on s'était appuyé sur un état historique plat, on aurait repris l'exécution au dernier sous-état actif mais du niveau le plus haut.

En phase de lustrage, le client peut également interrompre la machine, mais cela conduit à l'arrêt de la machine. Le diagramme d'état-transitions obtenu est illustré par la figure (6.18).

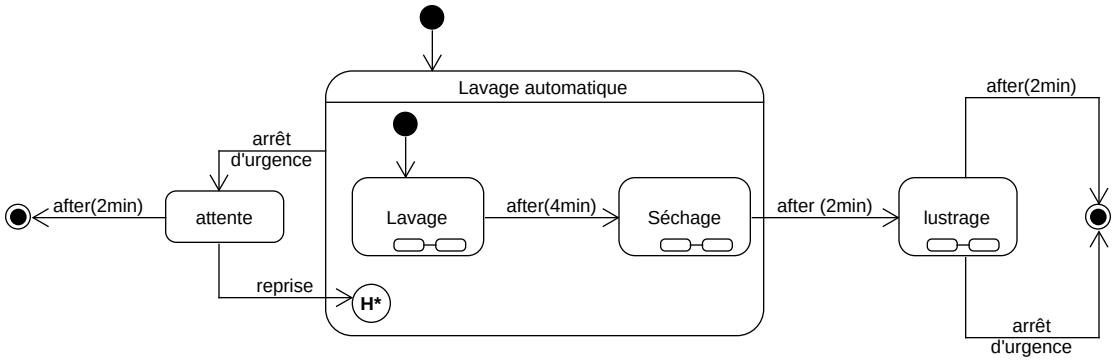


FIGURE 6.18 – Exemple du lavage automatique avec état historique profond

6.7.2 États concurrents et transitions

Lorsque l'état composite se décompose en sous-états concurrents (ou parallèles), il comporte des régions appelées régions concurrentes et séparées par une ligne en pointillée. Chaque région représente un flot d'exécution et peut posséder un état initial et un état final.

Un évènement peut déclencher une transition dans plusieurs sous-états.

Une transition qui atteint la bordure d'un état composite est équivalente à une transition qui atteint les états initiaux de toutes ses régions concurrentes.

La sortie de l'état composite est possible quand tous les sous-automates sont dans un état final.

Il est possible d'utiliser la notation des transitions concurrentes (barres épaisses) pour représenter les flots d'exécution parallèles d'un état composite.

La transition fork correspond à la création de deux états concurrents.

La transition join correspond à la barrière de synchronisation qui supprime la concurrence.

L'exemple illustré par la figure 6.15 est donc équivalent à celui illustré par la figure 6.19 qui s'appuie sur les transitions fork et join.

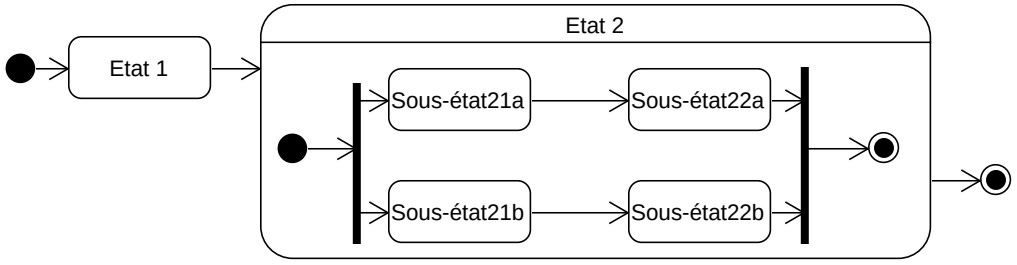


FIGURE 6.19 – Exemple d'utilisation des transitions fork et join

6.8 Éléments de méthodologie

Avant tout, il est important de rappeler que le diagramme d'état-transitions est construit pour décrire la dynamique (ou le cycle de vie d'un objet), il est par conséquent toujours associé à une et une seule classe.

Pour construire un diagramme d'états-transitions, il est important d'identifier et de représenter la séquence d'états, qui décrit le comportement d'un objet, et les transitions associées. L'état d'un objet représente un moment de son cycle de vie ; une période dans la vie d'un objet pendant laquelle ce dernier attend un évènement ou accomplit une activité. De plus, il est important de noter que :

— Tous les états doivent être accessibles depuis l'état initial.
— La présence d'un état terminal est optionnelle. Néanmoins, s'il y a des états terminaux alors il doit exister un chemin de cet état vers un état terminal.

Par ailleurs, un diagramme d'états-transitions doit être déterministe, c'est d'ailleurs pour cette raison qu'il n'y a qu'un seul état initial. En outre, si plusieurs transitions partant d'un même état ont le même évènement, alors il doit y avoir des gardes afin de garantir le déterminisme.

De manière générale, la construction d'un diagramme est un processus itératif, il faudra compléter le diagramme en ajoutant au fur et à mesure des transitions, des effets sur les transitions ou encore des activités dans les états. S'il devient trop complexe, il ne faut pas hésiter à structurer le diagramme en sous-états afin d'améliorer sa lisibilité.

6.9 Conclusion

Dans ce chapitre, nous nous sommes intéressés à l'étude des diagrammes d'états-transitions afin de compléter la démarche de modélisation par des vues dynamiques.

Les diagrammes d'états-transitions permettent de décrire l'évolution d'un système dans le temps à travers la modélisation du cycle de vie des objets le constituant.

Les diagrammes d'états-transitions décrivent tous les états possibles d'un objet. Ils s'intéressent aux changements d'états d'un seul objet à la fois. Ils représentent donc une vue synthétique du fonctionnement dynamique d'un objet.

6.10 Exercices

Exercice 1 :

Le changement de saisons constitue une boucle continue. On considère un objet de la classe Saison de durée de vie infinie. Donner le diagramme des états-transitions de la classe Saison correspondant aux états de l'année climatique des pays tempérés (printemps, été, automne, hiver).

Exercice 2 :

Considérons le fonctionnement d'un réveil matin simplifié :

- L'alarme est par défaut à l'état « off »
- Si on désire activer l'alarme (état « on »), il faut spécifier l'heure du réveil
- Quand l'heure courante devient égale à l'heure du réveil et si l'alarme est à « on », le réveil passe à l'état sonnerie et sonne sans s'arrêter.
- Seul l'appui sur le bouton arrêt sonnerie permet d'arrêter le réveil.
- Lorsque la sonnerie est arrêtée le réveil reste à l'état « on »

1. Donner le diagramme d'états-transitions correspondant.

Exercice 3 :

Considérons une classe Partie dont la responsabilité est de gérer le déroulement d'une partie de jeu d'échecs. Cette classe peut être dans deux états :

(a) le tour des blancs

(b) le tour des noirs.

Les événements à prendre en considération sont :

- un déplacement de pièces de la part du joueur noir
- un déplacement de pièces de la part du joueur blanc
- la demande de prise en compte d'un échec et mat par un joueur. S'il est validé par la classe partie, un échec et mat assure la victoire du dernier joueur. Dans ce cas, la partie aboutie par la victoire des noirs (noirs gagnants) ou la victoire des blancs (blancs gagnants)
- la demande de prise en compte d'un pat qui mène aussi à une fin de partie, avec une égalité.

1. Donner le diagramme d'états/transitions associé à la classe Partie.

Conclusion générale

Le génie logiciel est un domaine vaste de l'informatique qui couvre de nombreuses activités techniques, mais également organisationnelles. L'objectif principal du génie logiciel est de mieux maîtriser le processus de production d'un logiciel en termes de coût, de délai et de qualité. Pour cela, il s'intéresse à tout le cycle de fabrication d'un logiciel et à la manière de le rationaliser et de le rendre plus efficace à travers une meilleure organisation. Il est par conséquent très lié à la conduite de projet et reste toujours en évolution afin de satisfaire les nombreuses contraintes.

Le génie logiciel reste un domaine jeune que certains considèrent comme peu mature car personne n'a réussi à appliquer une méthodologie suffisamment satisfaisante pour atteindre les objectifs de qualité, de coût et de délai.

De nombreuses études ont montré que les principales causes d'échec du logiciel étaient, en grande partie, dues à son incapacité à répondre aux besoins des utilisateurs.

De fait, ces besoins sont souvent changeants et parfois difficiles à identifier ou à prévoir.

On comprend alors l'intérêt que peut avoir une démarche de modélisation qui permettra de mieux cerner les attentes des utilisateurs à travers la mise en œuvre de modèles divers du système à réaliser.

La modélisation orientée-objet offre de nombreux avantages et n'est pas nécessairement associée à la programmation orientée objet car elle peut servir dans un contexte plus large. Nous avons étudié tout au long de ce polycopié plusieurs diagrammes UML afin de pouvoir décrire différents aspects du système (structurel, fonctionnel, dynamique). Le langage UML, qui est une synthèse des approches précédentes (OMT, OOA/OOD, OOSE,...), est puissant et permet d'offrir une bonne modélisation du système à construire même si ce dernier ne fournit pas de mode d'emploi.

Qu'est-ce qu'une bonne modélisation en UML ?

Une bonne modélisation à travers les diagrammes UML sera caractérisée par une vue fidèle à la réalité tout en étant la plus simple possible. L'idée est de représenter uniquement les caractéristiques du monde réel que le diagramme cherche à modéliser sans dénaturer pour autant cette réalité.

Il faudra donc éviter les redondances et les détails non pertinents afin que le modèle soit facile à comprendre et à valider.

Par ailleurs, les modèles réalisés devront tenir compte, dans la mesure du possible, de la capacité du système à changer et à évoluer. L'ajout d'une nouvelle fonctionnalité ou d'une sous classe ne devra pas bouleverser tout le système.

Quels diagrammes UML utiliser ?

Les diagrammes UML à utiliser dépendent du contexte et des objectifs de l'application.

Ainsi, pour réaliser une application sans contraintes particulières, on s'appuiera tout d'abord sur un diagramme de cas d'utilisation afin de bien montrer les fonctionnalités attendues du système par ses utilisateurs.

On pourra illustrer différents scénarios d'utilisation à travers des diagrammes de séquence ou de communication.

On pourra ensuite s'appuyer sur un diagramme de classes pour montrer la structure interne du système.

L'utilisation d'un diagramme d'états-transitions peut également être intéressante particulièrement lorsqu'il s'agit de fonctionnalités très évolutives. Ces digrammes sont d'ailleurs particulièrement indiqués pour les systèmes réactifs.

Pour des systèmes où la contrainte de temps est importante (les systèmes temps réel par exemple), les digrammes de temps seront également d'une grande utilité.

Les digrammes que nous n'avons pas étudiés

Il existe de nombreux diagrammes UML, néanmoins, nous en avons sélectionné six, ce polycopié se voulant davantage une introduction à UML qu'un cours dédié à ce langage de modélisation. Les diagrammes que nous avons choisi d'étudier s'insèrent dans le cadre de la mise en œuvre d'un logiciel. Plus précisément, notre choix s'est porté sur les principaux diagrammes utilisés dans les phases d'analyse de besoins et de spécification.

Les lecteurs pourront se reporter vers d'autres ouvrages pour développer leurs connaissances des autres diagrammes. Un bref aperçu de quelques uns de ces digrammes est donné dans ce qui suit.

Les diagrammes d'activité permettent de modéliser des workflows complexes et sont considérés comme une variante des diagrammes d'états-transitions. Ils ressemblent à des organigrammes.

Les diagrammes de composants visent à décomposer un système complexe en grands blocs fonctionnels. Les composants sont des unités logiques connectés via leurs interfaces qui peuvent être placés sur des ressources physiques.

Les digrammes de déploiement sont très proches de l'implémentation. Il s'agit d'affecter des artefacts à des nœuds. Un artéfact est une unité physique de réalisation d'un système alors qu'un nœud est une ressource physique de traitement qui peut être relié à d'autres nœuds (réseau local, ...)

Corrigés des exercices

Corrigés des exercices du chapitre 1

Corrigé de l'exercice 1 :

1. b)
2. d)
3. b)
4. — facilité de maintenance : Développeur
   — facilité d'utilisation : Utilisateur
   — performance : Utilisateur
   — extensibilité : Développeur
   — réutilisabilité : Développeur

Corrigé de l'exercice 2 :

1. Les antériorités des tâches sont données dans le tableau 6.1.

Tâche (pour faire)	Durée	Tâche(s) antérieure(s) (il faut faire)
A	3	-
B	4	-
C	2	A
D	3	A
E	4	B, C

TABLE 6.1 – Table des antériorités

2. Il y a trois niveaux d'exécution :
   Niveau 1 : A et B
   Niveau 2 : C et D
   Niveau 3 : E
3. les dates au plus et au plus tard ainsi que le chemin critique sont donnés dans le tableau 6.2.

Tâche	DTO	FTO	DTA	FTA	Chemin critique
A	0	3	0	3	A
B	0	4	1	5	-
C	3	5	3	5	C
D	3	6	6	9	-
E	5	9	5	9	E

TABLE 6.2 – Table des dates au plus tôt et au plus tard

4. Le graphe PERT des potentielles tâches est donné dans la figure 6.20.

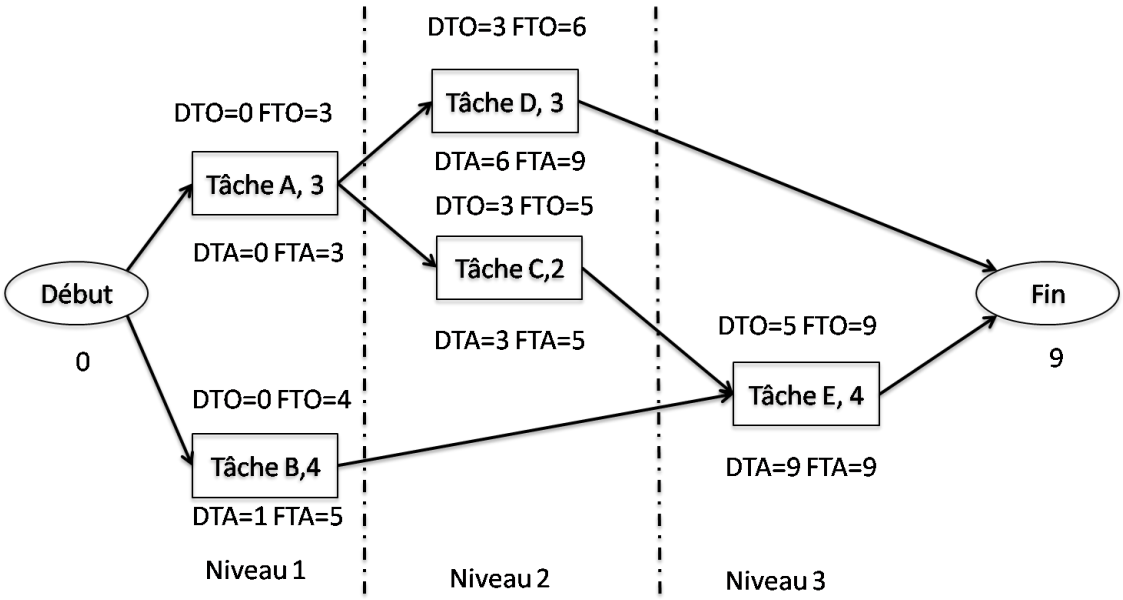


FIGURE 6.20 – Graphe PERT des potentielles tâches

Corrigé de l'exercice 3 :

1. Les dates de début et de fin au plus tôt du projet sont fournies dans le tableau 6.3

Tâche	Début au plus tôt (DTO)	Fin au plus tôt (FTO)
A	0	1
B	0	2
C	2	3
D	0	3
E	2	4
F	4	9
G	3	5
H	0	5
I	5	7
J	0	1
K	7	11
L	9	14
M	14	18

TABLE 6.3 – Table des dates de début et de fin

2. Ce projet dure 18 jours.
3. La figure 6.21 illustre le diagramme de Gantt.

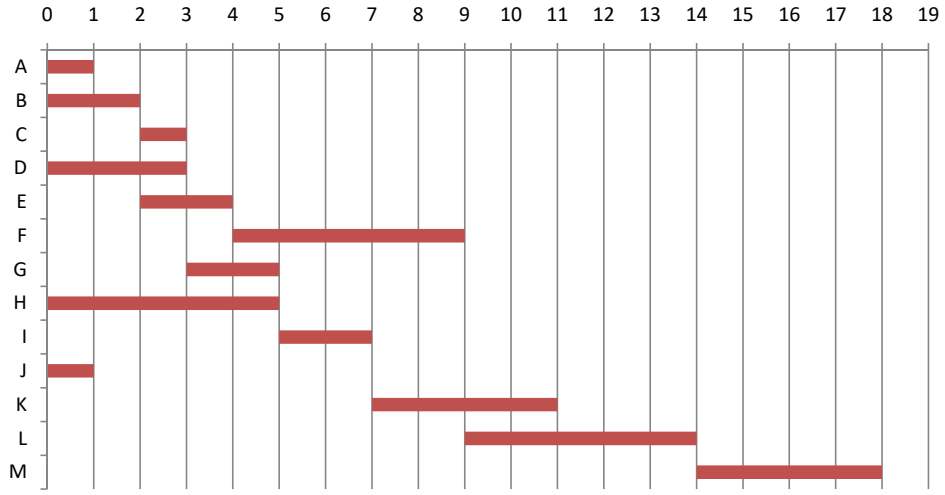


FIGURE 6.21 – Diagramme de Gantt

4. Le chemin critique est constitué des tâches : B E F L M

Corrigés des exercices du chapitre 2

Corrigé de l'exercice 1 :

1. UML est né du besoin d'unifier les nombreuses méthodes de conception orientée objet qui existaient à l'époque et du manque d'interopérabilité entre ces méthodes.
2. UML signifie Unified Modeling Language
3. les concepts de base sur lesquels repose la conception orientée objet sont
   — le concept de classe et d'objet
   — l'encapsulation
   — l'héritage
   — le polymorphisme
4. Les trois diagrammes UML structurels sont :
   — diagramme de classe
   — diagramme d'objets
   — diagramme de composants
   Les trois diagramme UML de comportement sont :
   — diagramme de cas d'utilisation
   — diagramme d'états-transitions
   — diagramme de séquence
5. En modélisation orientée objet, on appelle ce concept l'héritage multiple.

Corrigé de l'exercice 2 :

1. (c)
2. (c)
3. (d)
4. (a)

Corrigés des exercices du chapitre 3

Corrigé de l'exercice 1 :

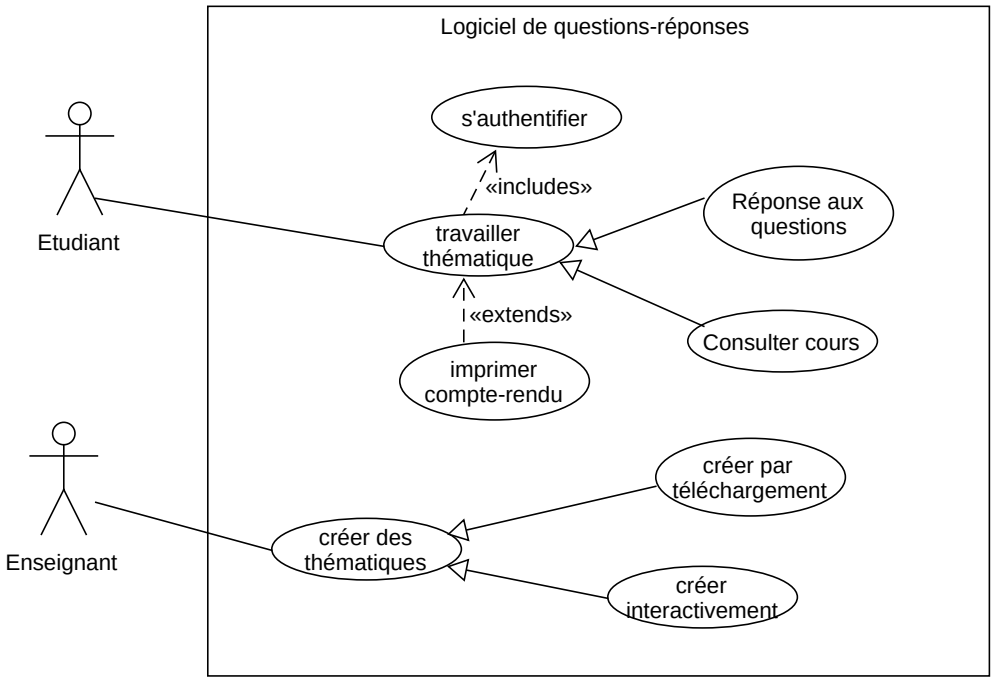


FIGURE 6.22 – Diagramme de cas d'utilisation corrigé

Corrigé de l'exercice 2 :

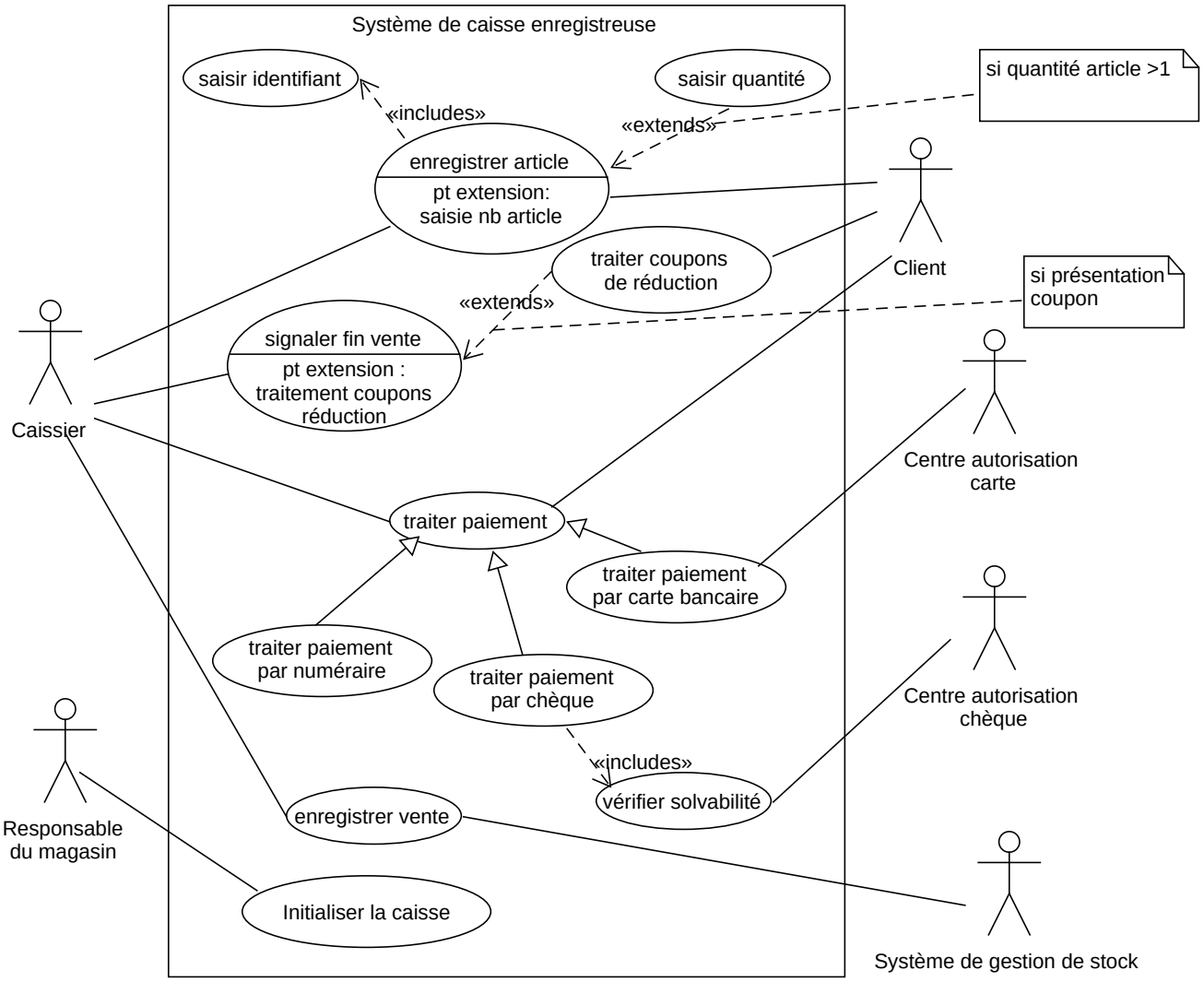
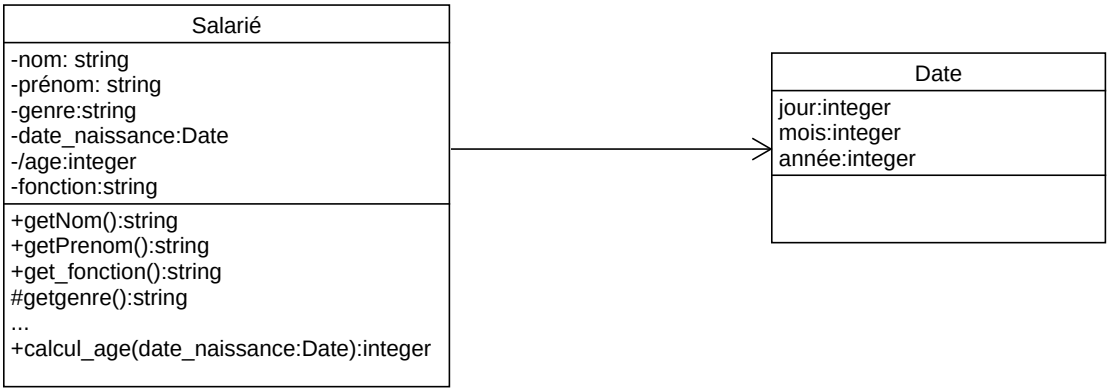


FIGURE 6.23 – Diagramme de cas d'utilisation de la caisse enregistreuse

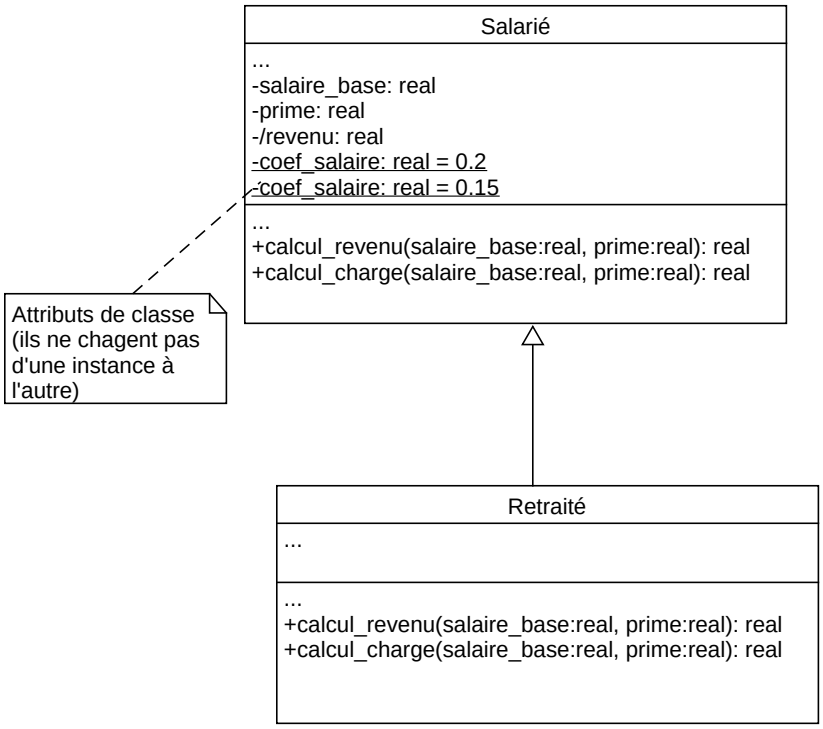
Corrigés des exercices du chapitre 4

Corrigé de l'exercice 1 :

1. La classe " Salarié "

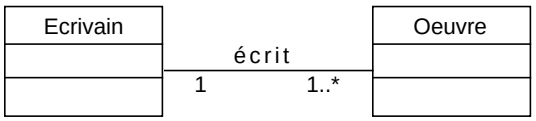


2. Modélisation des classes " Salarié " et " Retraité "

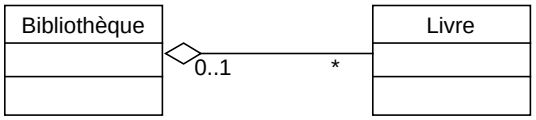


Corrigé de l'exercice 2 :

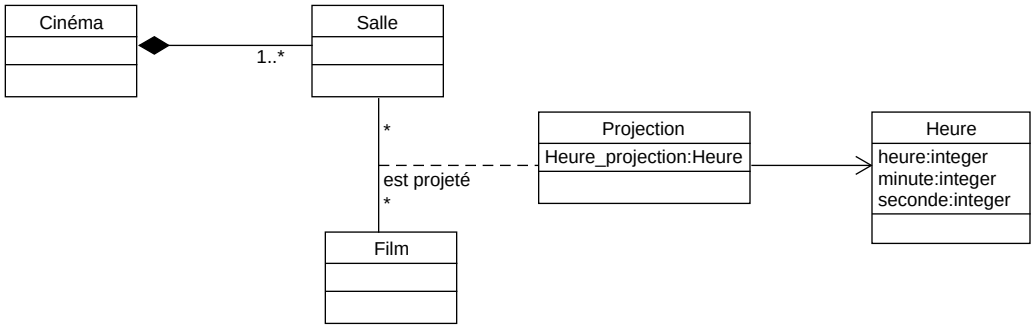
1) Tout écrivain a écrit au moins une œuvre



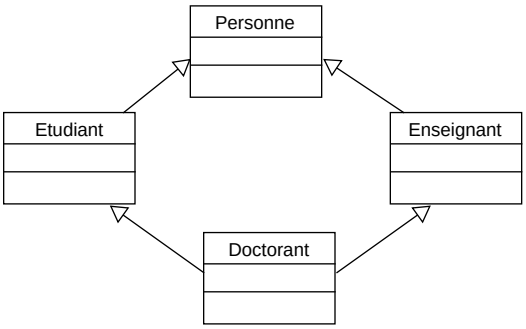
2) Une bibliothèque contient des livres



3) Les cinémas sont composés de plusieurs salles. Les films sont projetés dans des salles. Les projections des films dans des salles ont lieu à une heure déterminé.



4) Les étudiants et les enseignants sont deux sortes de personnes. Un doctorant est un étudiant qui assure des enseignements.



Corrigé de l'exercice 3 :

Le diagramme de classes obtenu est illustré par la figure 6.24.

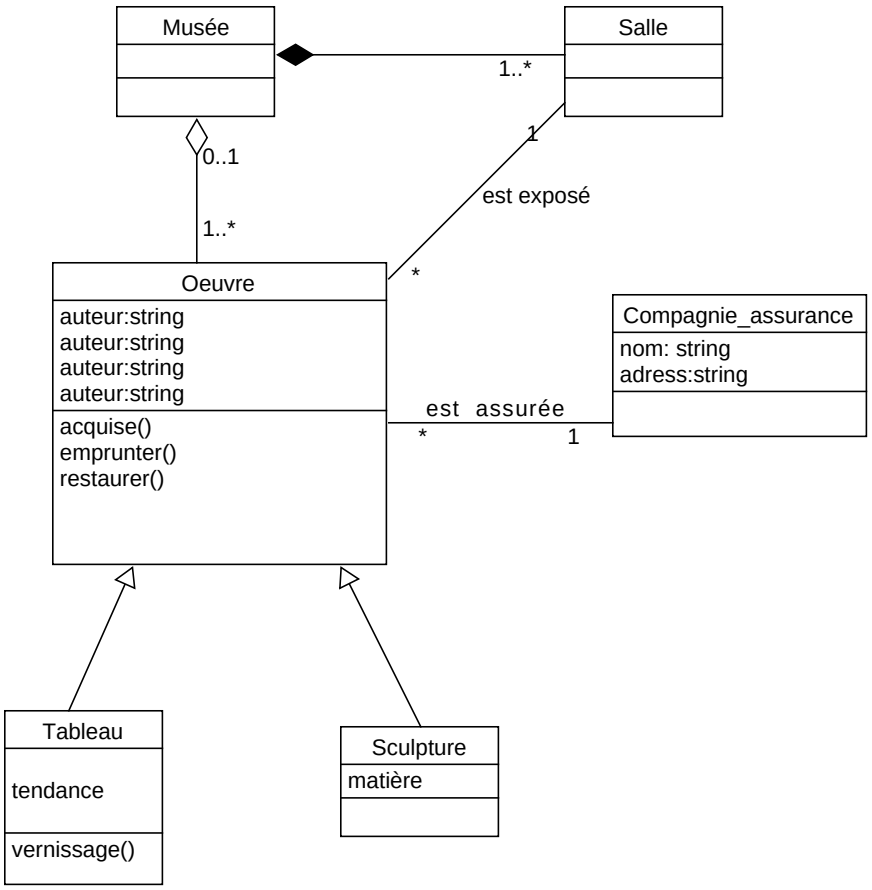
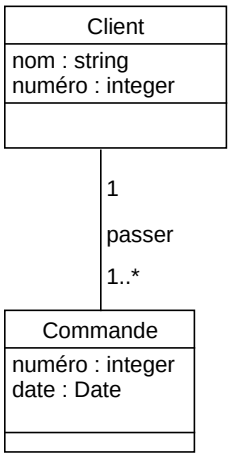


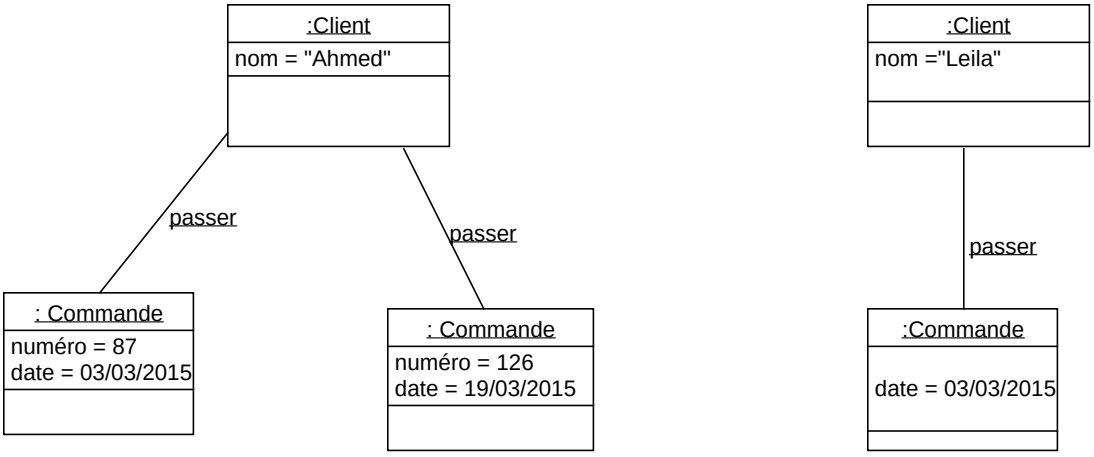
FIGURE 6.24 – Diagramme de classes obtenu

Corrigé de l'exercice 4 :

1. Diagramme de classes



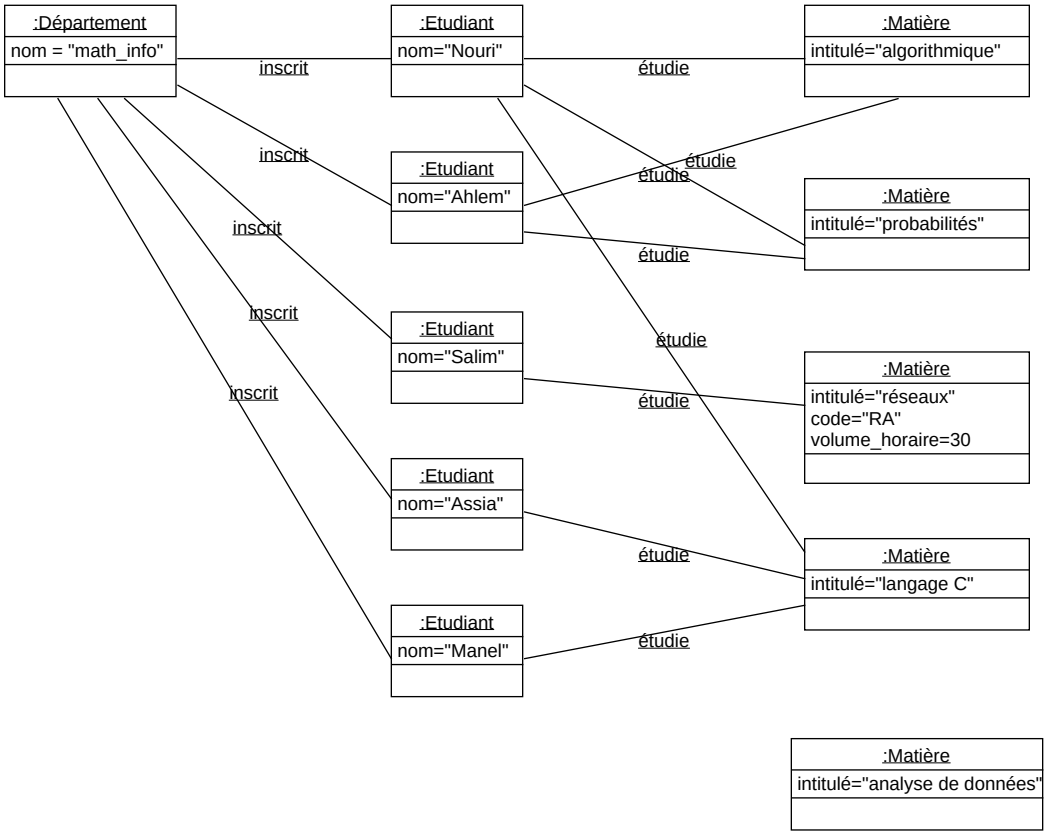
Diagrammes d'objets



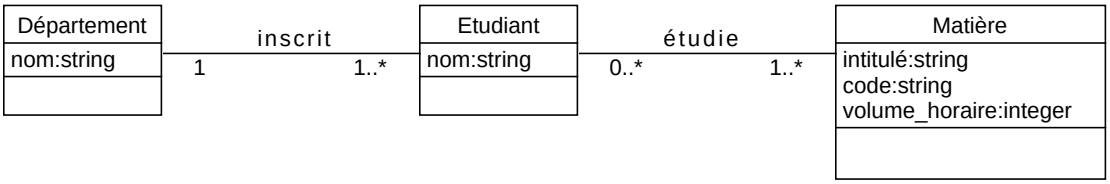
2. Non ce diagramme n'est pas cohérent avec le diagramme de classe est les contraintes d'association. En effet, une commande n'est associée qu'à un seul client.

Corrigé de l'exercice 5 :

1. Diagramme d'objets



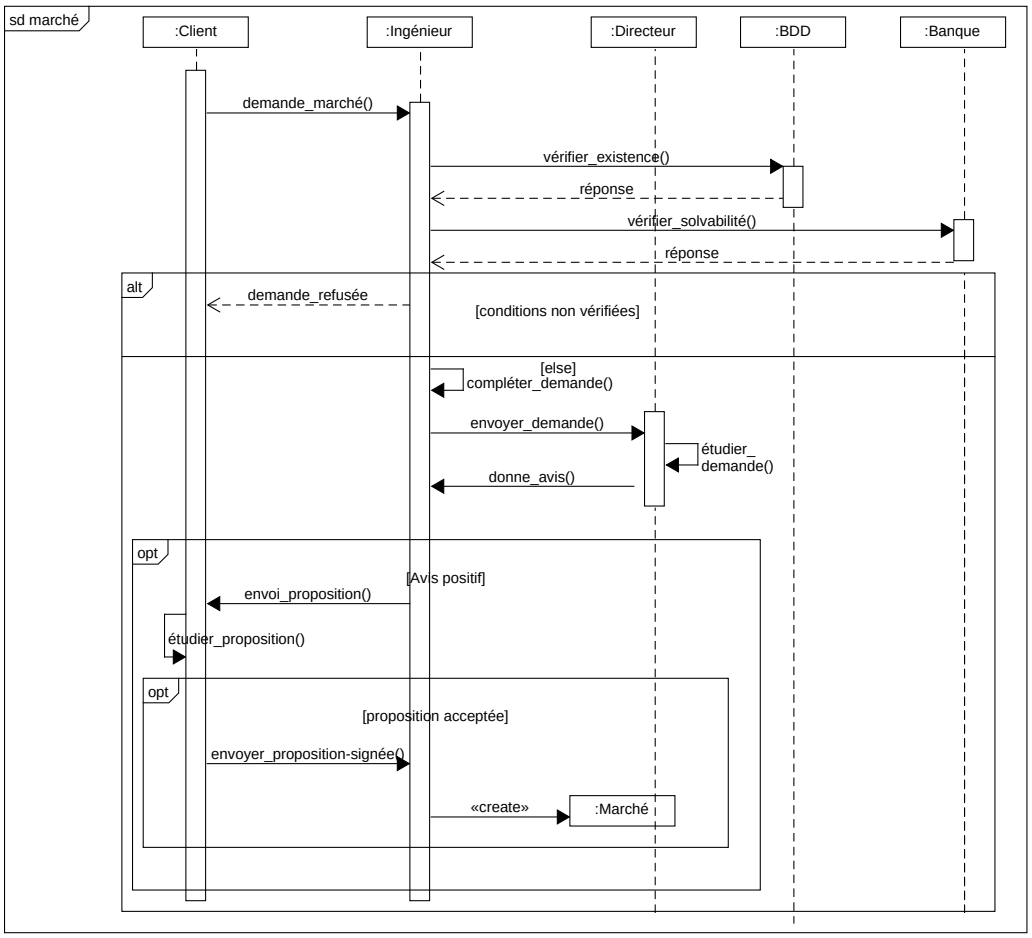
2. Diagramme de classes



Corrigés des exercices du chapitre 5

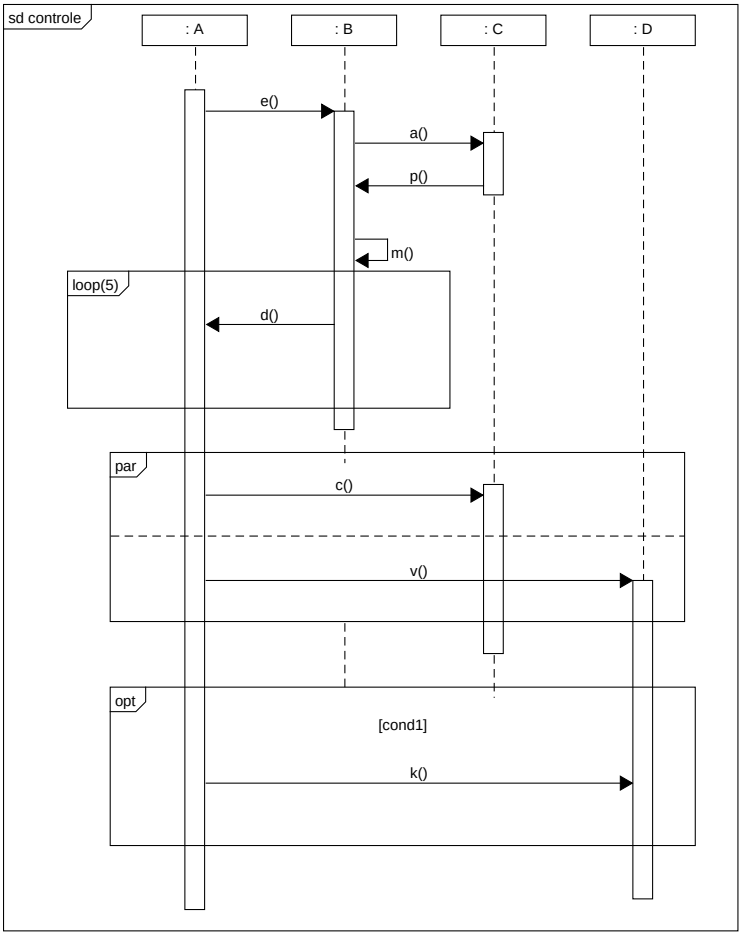
Corrigé de l'exercice 1 :

Le diagramme de séquence obtenu est :



Corrigé de l'exercice 2 :

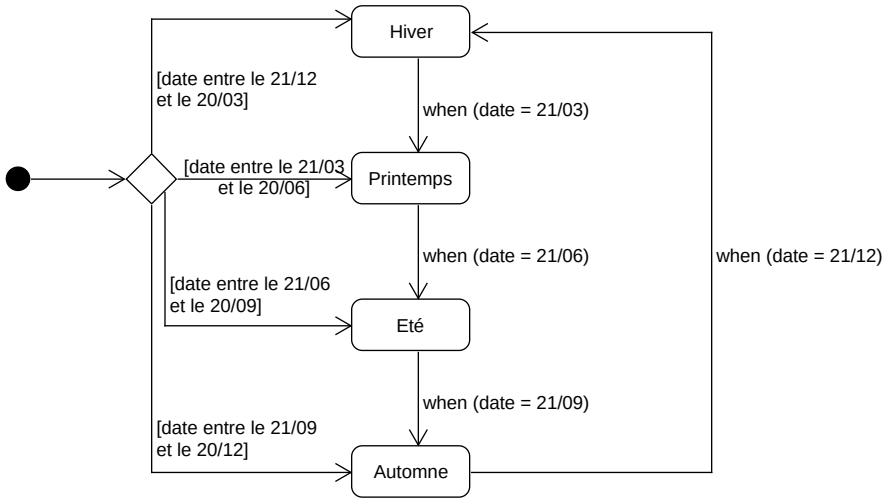
Le diagramme de séquence équivalent est :



Corrigés des exercices du chapitre 6

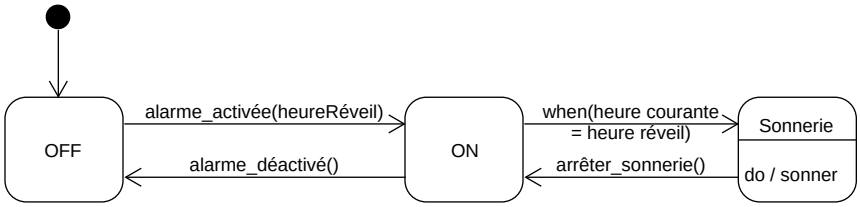
Corrigé de l'exercice 1 :

Le diagramme d'états-transitions obtenu est :



Corrigé de l'exercice 2 :

Le diagramme d'états-transitions obtenu est :



Bibliographie

[1] Pierre-Alain Muller. *Modélisation objet avec UML*. Eyrolles, 2003.

[2] Joseph Gabay. *UML 2. Analyse et conception*. Dunos, 2008.

[3] Pascal Roques. *UML 2 par la pratique*. 6ème édition, Eyrolles, 2008.

[4] Grady Booch James Rumbaugh, Ivar Jacobson. *The Unified Modeling Language. Reference Manual*. Addison-Wesley, 2004.

[5] Uml 2. https ://www.uml-diagrams.org.

[6] Omg (object management group), specification uml, version 2.5.1. https ://www.omg.org/spec/UML/.

[7] Laurent Audibert. Cours uml 2.0. https ://laurent-audibert.developpez.com/.

[8] Shari Lawrence Pfleeger and Joanne M. Atlee. *Software Engineering*. Fourth Edition, Pearson, 2010.

[9] Bern Bruegge and Allen H. Dutoit. *Object-Oriented Software Engineering – using UML, Patterns and Java*. Third Edition, Pearson, 2010.