République Algérienne Démocratique et Populaire
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique
Université Abderrahmane Mira Bejaia
Faculté des Sciences Exactes
Département d'Informatique

Support de cours

Module : Génie Logiciel

Niveau : 2ème année licence

Etabli par Dr. HAMZA Lamia

2018/2019

Préface

Depuis que nous enseignons Génie Logiciel aux étudiants nous avons utilisé un grand nombre d'ouvrages, à chacun sa spécificité. En tant qu'enseignante du module Génie Logiciel depuis 15 ans de manière continue sans interruption à l'université de Bejaia, j'ai essayé de rassembler les éléments les plus pertinents à la compréhension et à la maîtrise de cette matière par les étudiants. En qualité de responsable du Master "Génie logiciel professionnel" pendant 2015-2017, je pense que ce cours sera d'un grand apport à toute personne désirant accroitre ses connaissances dans le domaine de Génie Logiciel grâce à sa qualité pédagogique et sa clarté.

Ce support de cours a le mérite de présenter clairement :

- La naissance et l'importance de Génie Logiciel
- Le contexte objet dans lequel est né UML
- Les principaux diagrammes d'UML

Enfin, chaque chapitre est illustré d'exercices corrigés ayant déjà fait l'objet pour certains d'entre eux de tests auprès de mes étudiants.

Table des matières

Table des figures

Liste des tableaux

Liste des abréviations

OMG : Object Management Group

OMT : Object Modeling Technique

OOSE : Object Oriented Software Engineering

UML : Unified Modeling Language

Introduction générale

Le Génie Logiciel (GL) est une science récente dont l'origine remonte aux années 1970. À cette époque, l'augmentation de la puissance matérielle a permis de réaliser des logiciels plus complexes mais souffrant de nouveaux défauts : délais non respectés, coûts de production et d'entretien élevés, manque de fiabilité et de performances. Cette tendance se poursuit encore aujourd'hui.

L'apparition du GL est une réponse aux défis posés par la complexification des logiciels et de l'activité qui vise à les produire.

L'objectif du module GL est de comprendre le processus de développement du logiciel, en particulier les phases d'analyse et de conception orientée objet en utilisant UML (Unified Modeling Language). Ce support de cours introduit le domaine de GL pour des étudiants informaticiens novices dans ce domaine. Ce document s'adresse aux étudiants de 2ème année licence informatique et peut être utile à tout étudiant en informatique (Licence, Master ou études d'ingénieur) au cours de sa formation ou au cours de la réalisation de son projet de fin de cycle, ainsi qu'à tout professionnel dans le domaine de la conception et du développement informatique.

Ce support de cours est organisé en trois chapitres. Le contenu du document se résume comme suit :

Chapitre 1 : intitulé « Introduction au Génie Logiciel ». Ce chapitre présente plusieurs raisons de l'apparition du GL. Ce chapitre étudie toutes les activités qui mènent d'un besoin à la livraison du logiciel, il couvre les principaux cycles de vie d'un logiciel.

Chapitre 2 : intitulé « Introduction à la conception orientée objet ». Le but de ce chapitre est de se familiariser avec l'approche orientée objet en ce qui concerne la conception. Plus spécifiquement, au terme de ce chapitre, l'étudiant sera en mesure de comprendre les concepts orientés objet.

Chapitre 3 : intitulé « UML (Unified Modeling Language) ». Dans ce chapitre l'étudiant va découvrir le langage UML. Ce langage permet de modéliser de manière claire et précise la structure et le comportement d'un système orienté objet. UML est présenté avec les détails nécessaires permettant à l'étudiant de comprendre son utilité et de pouvoir l'utiliser pour faire des analyses ou des conceptions orientées objet.

Afin de rendre le document encore plus utile, pour chaque chapitre, il y a une série d'exercices ainsi que leurs corrigés. Ces exercices peuvent donner l'occasion à l'étudiant de tester ses connaissances acquises et de vérifier sa compréhension.

Chapitre 1: Introduction au Génie Logiciel

1. Introduction

L'objectif principal de ce chapitre est de faire comprendre à l'étudiant les causes de l'apparition du génie logiciel et les principaux cycles de vie d'un logiciel.

Le terme de Génie logiciel a été introduit à la fin des années 60 lors d'une conférence tenue pour discuter de ce que l'on appelait " la crise du logiciel ".

1.1. Crise du logiciel

Les symptômes les plus caractéristiques de cette crise sont :

- les logiciels réalisés ne correspondent souvent pas aux besoins des utilisateurs ;
- les logiciels contiennent trop d'erreurs (qualité du logiciel insuffisante) ;
- les coûts du développement sont rarement prévisibles et sont généralement exagérés ;
- la maintenance des logiciels est une tâche complexe et coûteuse ;
- les délais de réalisation sont généralement dépassés ;
- les logiciels sont rarement portables.

Tous ces problèmes ont mené à l'émergence d'une discipline appelée "**le génie logiciel**".

1.2. Définition du Génie Logiciel

Le génie logiciel (« Software Engineering » en anglais), 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.

Le terme génie logiciel désigne l'ensemble des **méthodes**, des **techniques** et **outils** contribuant à la production d'un **logiciel** de **qualité** avec maîtrise des **coûts** et **délais**.

1.3. Qualité exigée d'un logiciel

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

– La fiabilité (ou robustesse) : Le logiciel fonctionne raisonnablement en toutes circonstances, rien de catastrophique ne peut survenir, même en dehors des conditions d'utilisation prévues.

– La maintenabilité : Elle correspond au degré de facilité de la maintenance d'un produit logiciel.

– L'efficacité : On dit d'un logiciel qu'il est efficace s'il utilise les ressources d'une manière optimale (comme la mémoire par exemple).

– La facilité d'emploi : Elle est liée à la facilité d'apprentissage, d'utilisation, d'interprétation des erreurs et de rattrapage en cas d'erreur d'utilisation.

– La validité : C'est l'aptitude d'un produit logiciel à remplir exactement ses fonctions, définies par le cahier des charges et les spécifications.

– L'extensibilité : C'est la facilité avec laquelle un logiciel se prête 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.

– La compatibilité : C'est la facilité avec laquelle un logiciel peut être combiné avec d'autres logiciels.

– La portabilité : C'est la facilité avec laquelle un logiciel peut être transféré sous différents environnements matériels et logiciels.

– L'intégrité : C'est l'aptitude d'un logiciel à protéger son code et ses données contre des accès non autorisé.

1.4. Activités principales du processus de développement d'un logiciel

Quelque soit l'approche adoptée pour développer un logiciel, on y retrouve un certain nombre d'activités de base (le lecteur intéressé peut avoir plus de détails dans [2]) :

1.4.1. L'analyse des besoins

Le but de cette activité est d'éviter de développer un logiciel non adéquat. On va donc étudier le domaine d'application, ainsi que l'état actuel de l'environnement du futur système afin d'en déterminer les frontières, le rôle, les ressources disponibles et requises, les contraintes d'utilisation et de performance, etc.

Le résultat de cette activité est un ensemble de documents décrivant les aspects pertinents de l'environnement du futur système, son rôle et sa future utilisation. Un manuel d'utilisation préliminaire est parfois produit.

1.4.2. La spécification globale

Le but de cette activité est d'établir une première description du futur système. Ses données sont les résultats de l'analyse des besoins ainsi que des considérations de technique et de faisabilité informatique. Son résultat est une description de ce que doit faire le logiciel en évitant des décisions prématurées de réalisation (on dit quoi, on ne dit pas comment). Cette description va servir de point de départ au développement. Une première version du manuel de référence est parfois produite, ainsi que des compléments au manuel d'utilisation. Cette activité est fortement corrélée avec l'analyse des besoins avec laquelle des échanges importants sont nécessaires.

1.4.3. Conception architecturale et détaillée

L'activité de conception consiste à enrichir la description du logiciel de détails d'implémentation afin d'aboutir à une description très proche d'un programme. Elle se déroule souvent pendant deux étapes : l'étape de conception architecturale et l'étape de conception détaillée.

L'étape de conception architecturale a pour but de décomposer le logiciel en composants plus simples. On précise les interfaces et les fonctions de chaque composant. A l'issue de cette étape, on obtient une description de l'architecture du logiciel et un ensemble de spécifications de ses divers composants.

L'étape de conception détaillée fournit pour chaque composant une description de la manière dont les fonctions du composant sont réalisées : algorithmes, représentation des données.

1.4.4. Programmation

Cette activité consiste à passer du résultat de la conception détaillée à un ensemble de programmes ou de composants de programmes.

1.4.5. Gestion de configuration et intégration

La gestion de configuration a pour but de permettre la gestion des composants du logiciel, d'en maitriser l'évolution et les mises à jour tout au long du processus de développement. L'intégration consiste à assembler tout ou partie des composants d'un logiciel pour obtenir un système exécutable. Cette activité utilise la gestion de configuration pour assembler des versions cohérentes de chaque composant.

1.4.6. Validation et vérification

Le problème de l'adéquation des résultats de l'analyse des besoins et de la spécification globale est délicat.

- La question posée est : a-t-on décrit le « bon » système, c'est-à-dire un système qui répond à l'attente des utilisateurs et aux contraintes de leur environnement ? L'activité qui a pour but de s'assurer que la réponse à cette question est satisfaisante s'appelle la validation.
- Par opposition, l'activité qui consiste à s'assurer que les descriptions successives du logiciel, et le logiciel lui-même, satisfont la spécification globale s'appelle la vérification. La question à laquelle on répond dans le cas de la vérification est : le développement est-il correct par rapport à la spécification de départ ?

La validation consiste essentiellement en des revues et inspections de spécifications ou de manuels, et du prototypage rapide (voir la section 1.4.7).

La vérification inclut également des inspections de spécifications ou de programmes, ainsi que de la preuve et du test.

Une preuve porte sur une spécification détaillée ou un programme et permet de prouver que celle-ci ou celui-ci satisfait bien la spécification de départ.

Le test consiste à rechercher des erreurs dans une spécification ou un programme.

1.4.7. Rôle du maquettage

Quand les besoins ne sont pas précis, l'activité de validation devient difficile, pour cela, on adopte la solution du maquettage (prototypage rapide), il s'agit de développer un programme qui est une ébauche du futur logiciel (il n'en a pas les performances ni toutes les fonctionnalités d'un produit fini). Ce programme est ensuite soumis à des scénarios en liaison avec les futurs utilisateurs afin de préciser leurs besoins. On parle alors de maquette exploratoire.

Le maquettage peut aussi intervenir lors d'une étape de conception : des choix différents peuvent être expérimentés et comparés au moyen de maquettes (dans ce cas, on parle de maquette expérimentale).

2. Cycle de vie du logiciel

Le cycle de vie d'un logiciel est constitué de l'enchaînement des différentes activités nécessaires à son développement.

Le cycle de vie permet de détecter les erreurs le plus tôt possible et ainsi de maîtriser la qualité du produit, les délais de sa réalisation et les coûts associés.

Il existe plusieurs modèles de cycle de vie d'un logiciel (le lecteur intéressé peut avoir plus de détails dans [1, 2]).

2.1. Le modèle en " Cascade "

Le cycle de vie dit de la « cascade » date de 1970, il est l'œuvre de Royce.

Le principe du modèle en cascade est très simple : on convient d'avoir un certain nombre d'étapes (ou phases). Une étape doit se terminer à une date donnée par la production de certains documents ou logiciels.

Les résultats de l'étape sont soumis à une revue approfondie, et on ne passe à l'étape suivante que quand ils sont jugés satisfaisants.

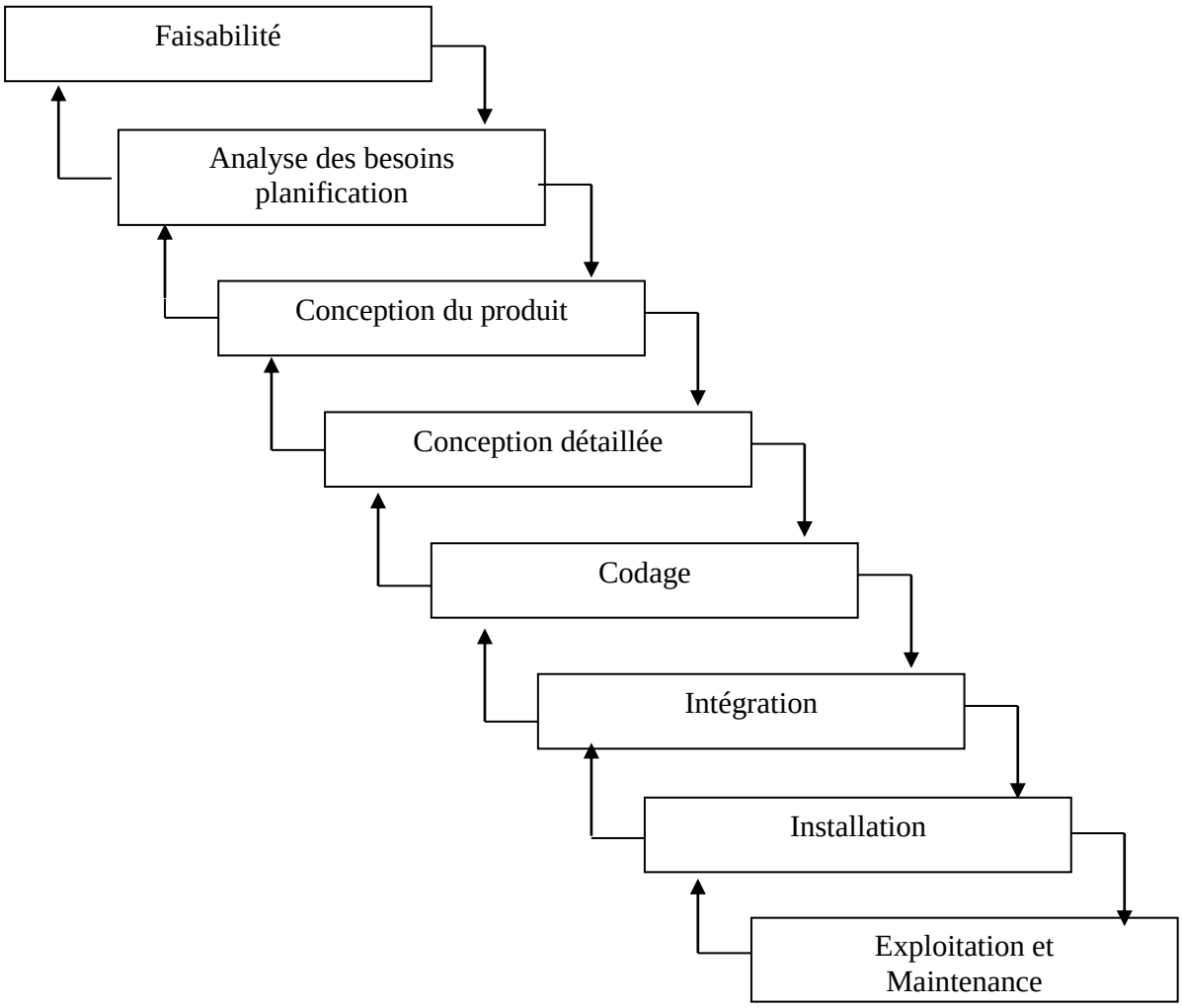


Figure 1.1 - modèle en cascade

.

Le modèle original ne comportait pas de possibilité de retour en arrière. Celle-ci a été rajoutée ultérieurement sur la base qu'une étape ne remet en cause que l'étape précédente, ce qui, dans la pratique, s'avère insuffisant.

Les versions actuelles de ce modèle font apparaître la validation-vérification à chaque étape. On trouve donc successivement :

- avec la faisabilité et l'analyse des besoins, de **la validation** ;
- avec la conception du produit et la conception détaillée, de **la vérification** ;
- avec le codage, du test **unitaire** ;
- avec l'intégration, du **test d'intégration**, puis du **test d'acceptation** ;
- avec l'installation, du **test système**.

- *Avantages*

– Simple à mettre en œuvre,

– La documentation est produite à chaque étape.

- *Inconvénients*

– Difficulté d'avoir toutes les spécifications du client,

– Preuve tardive du bon fonctionnement,

– Pas transparent au client lors du développement.

Ce modèle est mieux adapté aux petits projets ou à ceux dont les spécifications sont bien connues et fixes.

2.2. Le modèle en " V "

Dérivé du modèle de la cascade, le modèle en V du cycle de développement montre non seulement l'enchaînement des phases successives, mais aussi les relations logiques entre phases plus éloignées.

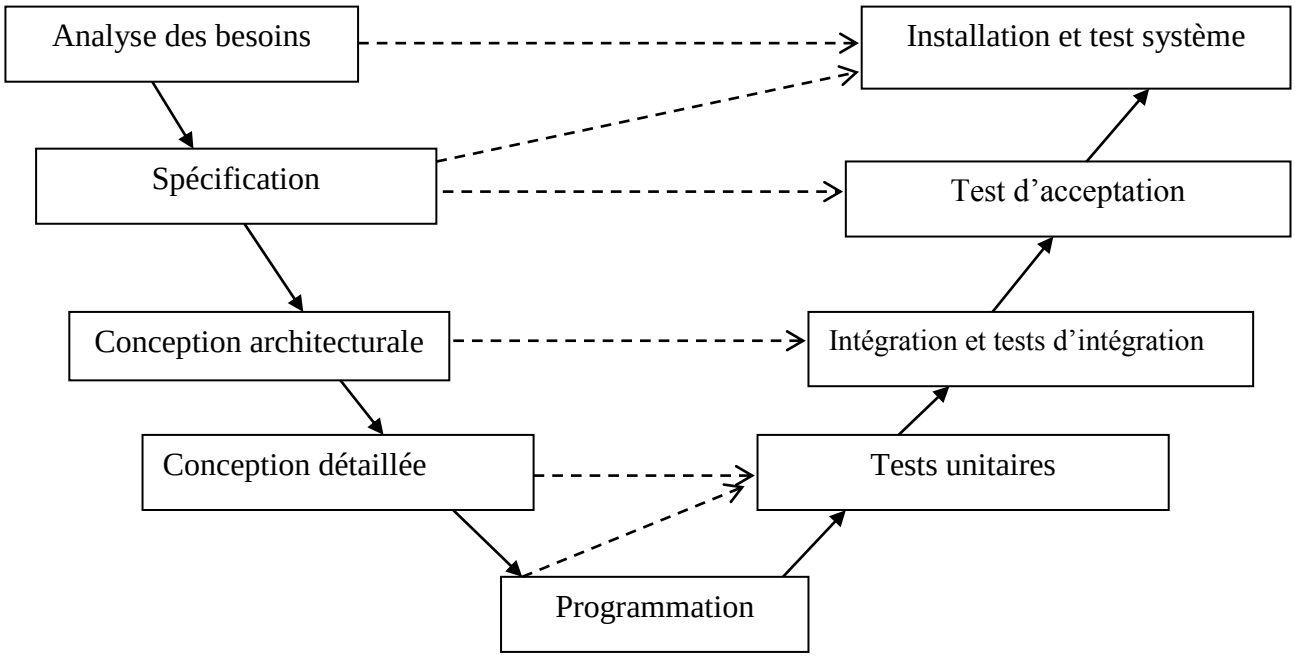


Figure 1.2 - modèle en V.

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

- *Avantages :*

– Les plans de test sont meilleurs,

– Les éventuelles erreurs peuvent être détectées plus tôt.

- *Inconvénients :*

– Les plans de test et leurs résultats obligent à une réflexion et à des retours sur la description en cours,

– La partie droite peut être mieux préparée et planifiée.

Ce modèle est idéal quand les besoins sont bien connus, quand l'analyse et la conception sont claires, ce modèle est adapté aux projets de taille et de complexité moyenne.

2.3. Le modèle par prototypage

Le prototypage est également considéré comme un modèle de développement de logiciels.
Il s'agit d'écrire une première spécification et de réaliser un sous-ensemble du produit logiciel final. Ce sous ensemble est alors de plus en plus raffiné et évalué jusqu'à obtenir le produit final.

Deux types de prototypage :

1. Jetable : squelette du logiciel qui n'est créé que dans un but et dans une phase particulière du développement

2. Evolutif : conservé tout au long du cycle de développement. Il est amélioré et complété pour obtenir le logiciel final

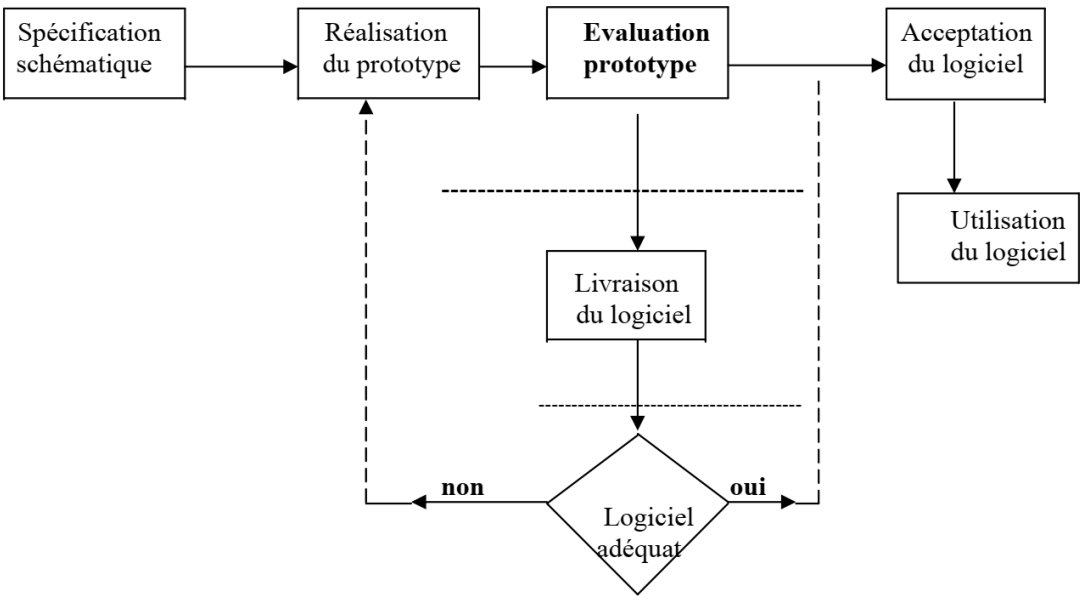


Figure 1.3 - modèle par prototypage.

- *Avantages :*

– Validation très tôt dans le processus,

– Visibilité (donc compréhension) du produit final,

– Anticipation des changements : maintenance incluse tout le long du projet.

- *Inconvénients :*

– Risque de déstructuration du code,

– Difficulté de planifier et de gérer le développement.

Ce modèle est bien adapté pour les projets innovants.

2.4. Le modèle par incrément

Dans ce type de modèle, un seul sous ensemble des composants d'un système est développé à la fois : un logiciel noyau est tout d'abord développé, puis des incréments sont successivement développés et intégrés.

Le processus de développement de chaque incrément est un des processus classiques.

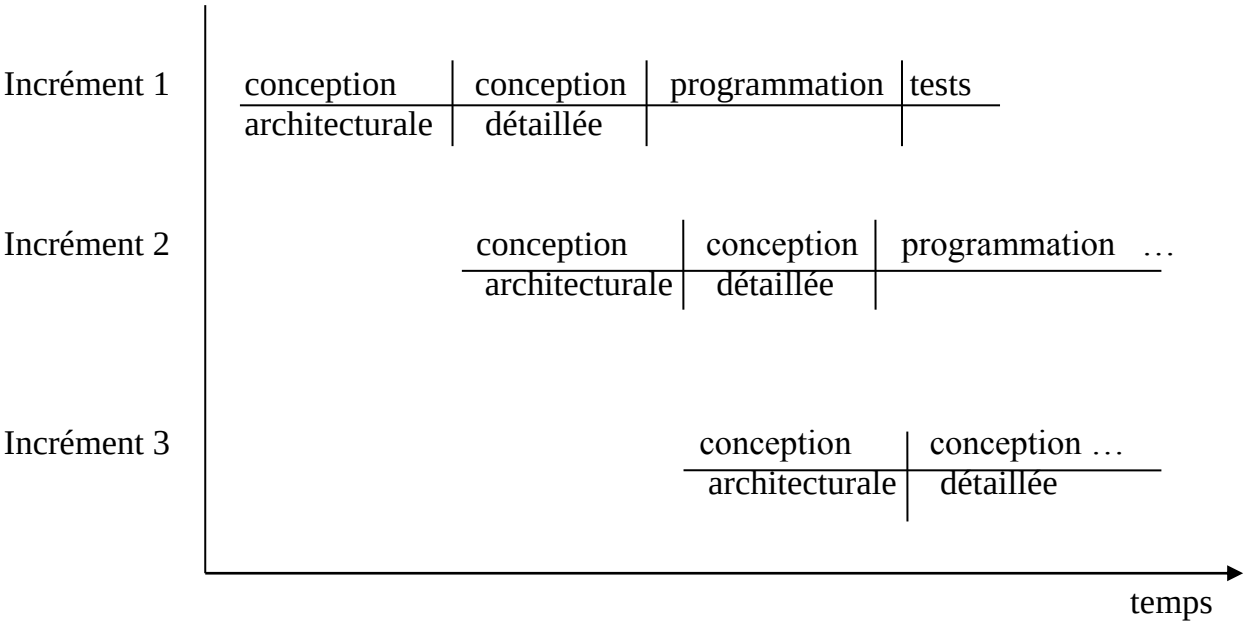


Figure 1.4 - modèle par incrément.

- *Avantages :*

– Chaque développement est moins complexe,

– Les intégrations sont progressives,

– Il peut y avoir des livraisons et des mises en services après chaque intégration d'incrément.

- *Inconvénients :*

– Mettre en cause le noyau ou les incréments précédents,

– Ne pas pouvoir intégrer de nouveaux incréments.

Ce modèle est utilisé dans de grands projets nécessitant des livraisons rapides.

2.5. Le modèle en spirale

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

Chaque cycle de la spirale se déroule en quatre phases :

1. Détermination des objectifs du cycle, des alternatives pour les atteindre, des contraintes, à partir des résultats des cycles précédents ou s'il n'y a pas, d'une analyse préliminaire des besoins ;

2. Analyse des risques, évaluation des alternatives, éventuellement maquettage (prototypage) ;

3. Développement et vérification de la solution retenue ;

4. Revue des résultats et planification du cycle suivant.

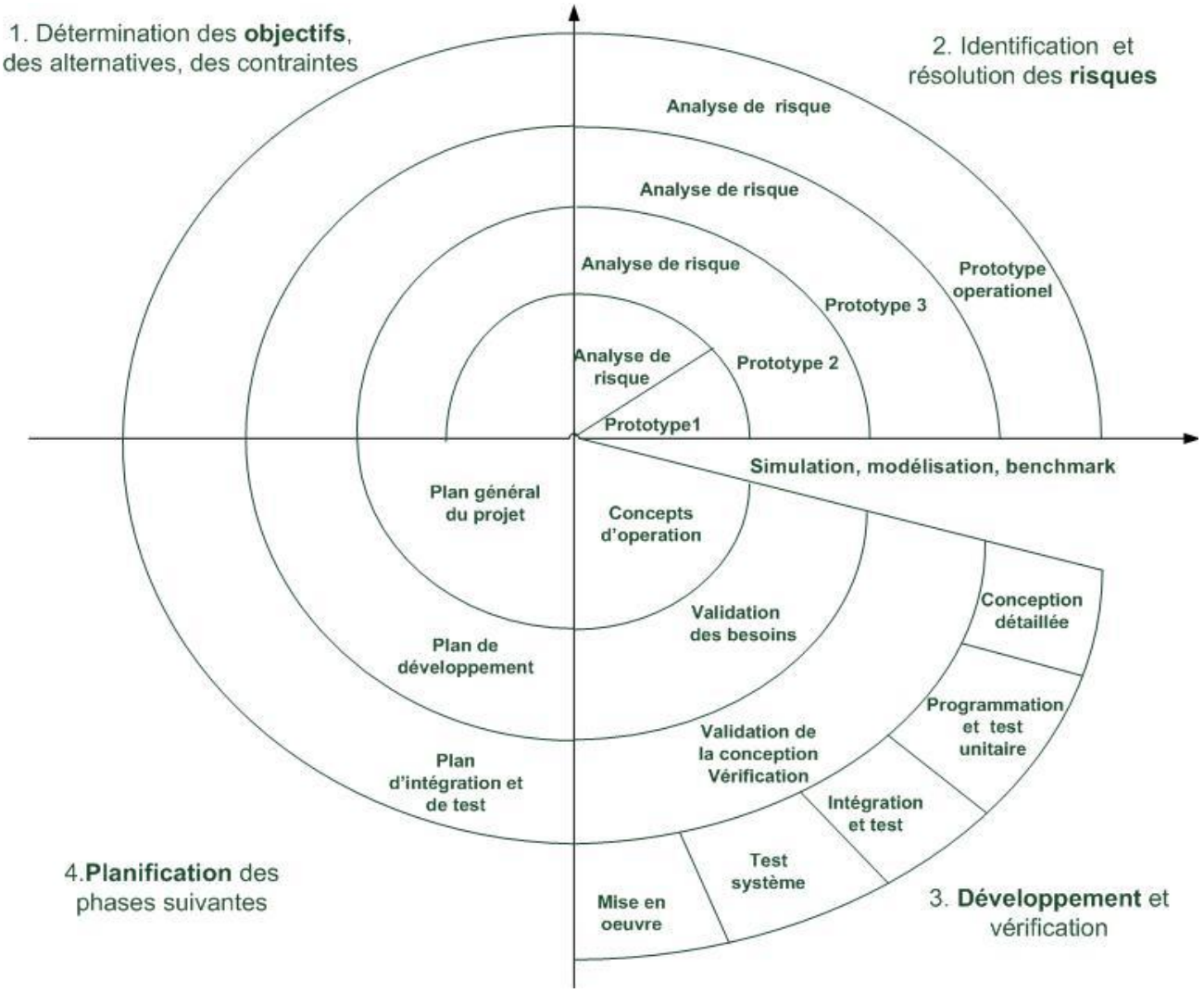


Figure 1.5 - modèle en spirale.

- *Avantages :*

– Validité des besoins,

– Inclut l'analyse de risque et le prototypage.

- *Inconvénients :*

– Calendrier et budget souvent irréalistes,

– Difficulté d'anticiper les composants nécessaires dans les cycles ultérieurs,

– Sa mise en œuvre demande de grandes compétences et devrait être limitée aux projets innovants à cause de l'importance qu'il accorde à l'analyse des risques.

Ce modèle est utilisé pour des projets innovants, à risques, et dont les enjeux sont importants.

3. Conclusion

Le choix d'un modèle dépend fortement du domaine d'application, lorsque ce dernier est critique, les modèles adéquats sont le modèle de la spirale et à un degré moins le prototypage. Pour des applications classiques (gestion), le modèle en V ou le modèle en cascade peuvent être appropriés. Pour des domaines nécessitant les livraisons rapides mais non critiques le modèle à incrément pourra se justifier.

4. Exercices

Exercice 1:

Comparer le prototypage jetable par rapport au prototypage non jetable.

Exercice 2:

Dresser dans un tableau comparatif, les avantages, les inconvénients ainsi que le type de projet correspondant à chaque modèle de cycle de vie.

5. Corrigés des exercices

Corrigé de l'exercice 1

Comparaison entre le prototypage jetable et le prototypage non jetable.

a) Prototypage jetable (maquettage) :

Suppose que l'on construise un prototype pour approfondir la compréhension du problème et pour trouver la solution puis le jeter après usage.

b) Prototypage non jetable (évolutif) :

Le prototype n'est pas abandonné mais adapté à la nouvelle configuration des besoins, ce processus est répété successivement jusqu'à la satisfaction des besoins de l'utilisateur.

	Prototypage jetable	Prototypage non jetable
ce que l'on récolte	Rapidité Provisoire (sans rigueur)	Rigueur
ce que l'on construit	Modéliser seulement les parties difficiles	Modéliser d'abord les parties faciles ensuite celles difficiles
ce que l'on gagne	Optimiser le temps de développement	Optimisation et flexibilité du temps de développement
conclusion	Jeter après usage	Faire évoluer

Corrigé de l'exercice 2 : Tableau 1 - Tableau comparatif contenant les avantages, les inconvénients ainsi que le type de projet correspondant à chaque modèle de cycle de vie.

MCVL (Modèle de Cycle de Vie d'un Logiciel)	Avantages	Inconvénients	Type de projet
Modèle en cascade	- Simplicité - La documentation est produite à chaque étape (cahier des charges, document de conception, etc.)	- Difficulté d'avoir toutes les spécifications du client, - Preuve tardive du bon fonctionnement, - Pas transparent au client lors du développement.	- Petits projets, - Projets dont les spécifications sont bien connues et fixes.
Modèle en V	- Les plans de test sont meilleurs, - Les éventuelles erreurs peuvent être détectées plus tôt.	- Les plans de test et leurs résultats obligent à une réflexion et à des retours sur la description en cours, - La partie droite peut être mieux préparée et planifiée.	Projets de taille et de complexité moyenne (les besoins sont bien connus, l'analyse et la conception sont claires).
Modèle par prototypage	- Validation très tôt dans le processus, - Visibilité du produit final, - Anticipation des changements (maintenance incluse tout le long du projet).	- Risque de déstructuration du code - Difficulté de planifier et de gérer le développement.	Projets innovants.
Modèle par incrément	- Chaque développement (incrément) est moins complexe, - Les intégrations sont progressives, - Il peut y avoir des livraisons et des mises en services après chaque intégration d'incrément.	- Mettre en cause le noyau ou les incréments précédents, - Ne pas pouvoir intégrer de nouveaux incréments.	Grand projet et nécessité de livraisons rapide.
Modèle en spirale	- Validité des besoins, - Inclut l'analyse de risque et le prototypage.	- Calendrier et budget souvent irréalistes, - Difficulté d'anticiper les composants nécessaires dans les cycles ultérieurs, - Sa mise en œuvre demande de grandes compétences et devrait être limitée aux projets innovants à cause de l'importance qu'il accorde à l'analyse des risques.	Projets innovants, à risques, dont les enjeux sont importants.

Chapitre 2 : Introduction à la conception orientée objet

1. Introduction

L'objectif principal de ce chapitre est de faire comprendre à l'étudiant les concepts orientés objet.

La conception propose une solution au problème spécifié lors de l'analyse, il existe deux grandes approches de conception, l'approche orientée fonction (approche fonctionnelle) et l'approche orientée objet[1].

1.1. L'approche fonctionnelle

- Raisonnement en termes de fonctions du système ;
- Séparation des données et du code de traitement ;
- Décomposition fonctionnelle descendante.

Les limites de l'approche fonctionnelle

- Un programme est conçu comme un ensemble de modules fonctionnels (procédures ou fonctions) qui manipulent des données ;
- Communication entre fonctions par passage de paramètres ou par variables globales ;
- Accès libre aux données par n'importe quelle fonction ;
- Difficulté de réutiliser du code déjà écrit et testé.

1.2. L'approche orientée objet

- Regroupement données-traitements ;
- Diminution de l'écart entre le monde réel et sa représentation informatique ;
- Décomposition par identification des relations entre objets.

[1]L'approche de conception orientée objet est basée sur la notion d'objet qui représente les entités du monde réel, (le lecteur intéressé peut avoir plus de détails dans [6]).

2. Concepts de l'approche orientée objet

2.1. Objet : Un objet est une entité représentée par un état et un ensemble d'opérations qui manipulent cet état.

Un **objet** est caractérisé par :

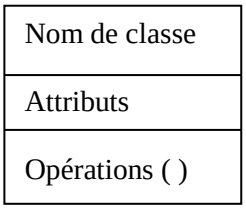
- un **identificateur** (nom de l'objet) ;
- un **état** (sous forme d'un ensemble d'attributs) ;
- un **comportement** (un ensemble d'opérations).

2.2. Classe : Une classe correspond à la description d'une famille d'objet ayant une même structure et un même comportement.

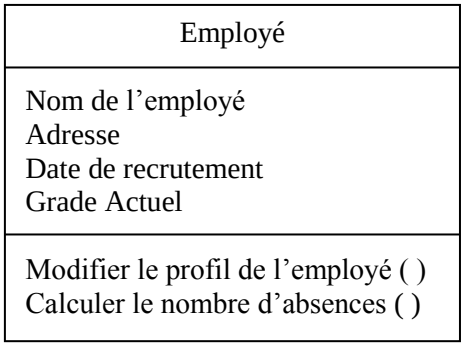
Une classe définit, une partie statique et une partie dynamique :

- La partie statique est représentée par un ensemble **d'attributs** pouvant posséder une valeur (ces attributs caractérisent l'état des objets durant leurs exécutions).
- La partie dynamique est représentée par l'ensemble des **opérations** qui définissent le comportement commun aux objets de la même classe.

Notation graphique[2] :



Exemple de classe :



[2]Les notations graphiques utilisées dans ce cours sont inspirées d'UML (Unified Modeling Language).

*Remarque :*Une classe peut être citée avec uniquement son nom, sans en préciser les détails.

Instance d'une classe : Une instance est un objet créé à partir d'une classe. La création d'un objet à partir d'une classe s'appelle l'instanciation.

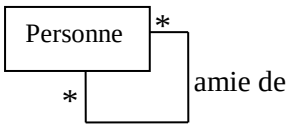
2.2.1. Association : L'association représente une relation entre plusieurs classes. Elle correspond à l'abstraction[3] des liens qui existent entre les objets dans le monde réel. Une association peut être identifiée par son nom. Il est possible d'exprimer les multiplicités (cardinalités) sur le lien d'association.

Exemple :

Dans cet exemple, on représente le fait qu'un employé est affecté à un seul service mais que ce dernier peut accueillir plusieurs employés à la fois.

Remarque : Lorsque le lien existe entre des objets de la même classe, on parle d'association réflexive.

Exemple : Deux personnes peuvent être amies (on considère que l'amitié est réciproque).

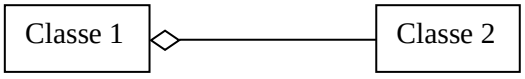


2.2.1.1. Agrégation et Composition :

Il existe des liens d'association particuliers, tel que l'agrégation et la composition :

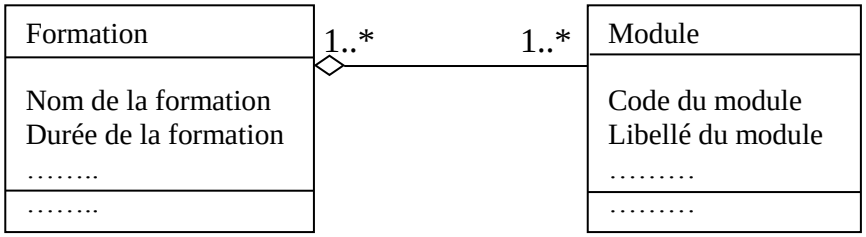
1) Agrégation : L'agrégation est une association qui décrit une relation d'inclusion entre une partie et un tout (l'agrégat). L'agrégation se représente par un petit losange blanc du côté de l'agrégat.

Notation graphique :



[3]L'abstraction consiste à identifier les caractéristiques intéressantes d'une entité en vue d'une utilisation précise.

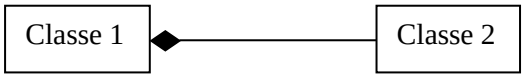
Exemple :
Dans le cadre d'une formation, le cursus de cette dernière est une agrégation de modules à enseigner.



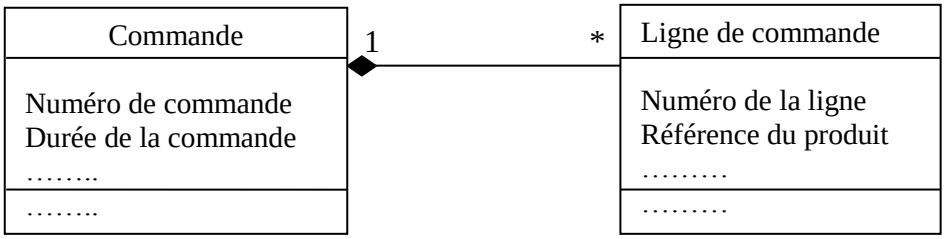
Remarquons dans cet exemple, que la suppression d'une formation ne conduit pas automatiquement à la suppression des modules étant donné que ces derniers peuvent très bien être enseignés dans d'autres formations.

2) Composition : Une composition est une forme forte d'agrégation. C'est-à-dire que la suppression de l'objet agrégat mène à la suppression des objets agrégés. La cardinalité du côté composite ne doit pas être supérieure à 1 (1 ou 0..1). La composition se représente par un petit losange de couleur noire.

Notation graphique :

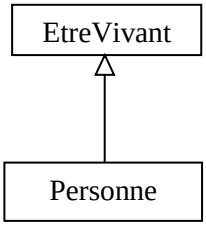


Exemple :
Une commande est composée d'un ensemble de lignes de commande décrivant les produits commandés. La suppression d'une commande conduira obligatoirement à la suppression de toutes ses lignes.



2.2.2. Héritage : L'héritage permet un partage hiérarchique de propriété (attributs et opérations). La relation d'héritage entre deux classes est représentée par une flèche à la tête en forme de triangle blanc.

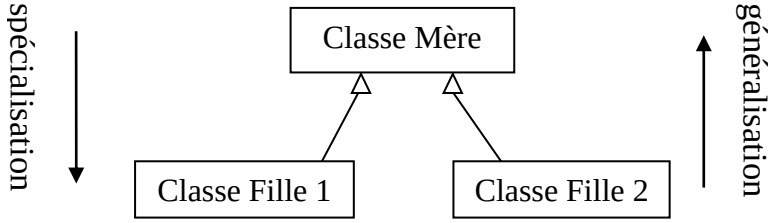
Exemple : L'exemple suivant représente la classe Personne qui hérite de la classe EtreVivant.



L'héritage est mis en œuvre grâce à deux propriétés qui sont : la **généralisation** et la **spécialisation**.

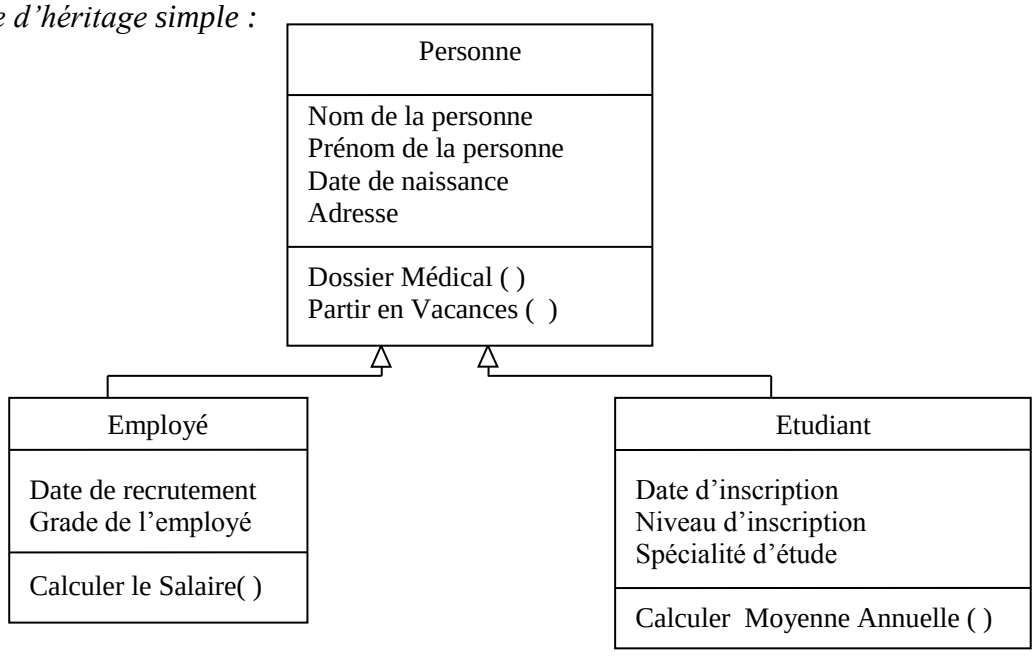
- La généralisation décrit le fait de pouvoir regrouper un ensemble de classes partageant des éléments en commun en une seule super-classe (ou classe mère)
- La spécialisation représente le phénomène inverse, c'est-à-dire, pouvoir dériver à partir d'une classe ou super-classe des sous-classes (ou classes filles) ayant des propriétés spécifiques les distinguant les unes des autres.

Exemple :



L'héritage est simple lorsqu'une classe hérite d'une super-classe.

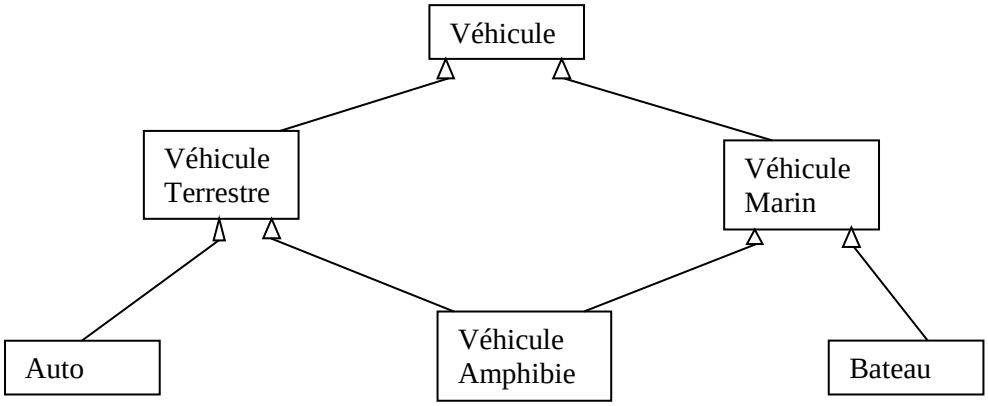
Exemple d'héritage simple :



Les classes « Employé » et « Etudiant » sont des dérivés de la classe « Personne » et héritent des propriétés de cette dernière. Cependant, « Employé» se distingue par d'autres attributs tels que la date de recrutement, le grade et le salaire alors que « Etudiant » possède une date d'inscription, un niveau d'étude, une spécialité et une moyenne annuelle.

L'héritage est multiple quand il y a deux ou plusieurs super-classes pour une même sous-classe.

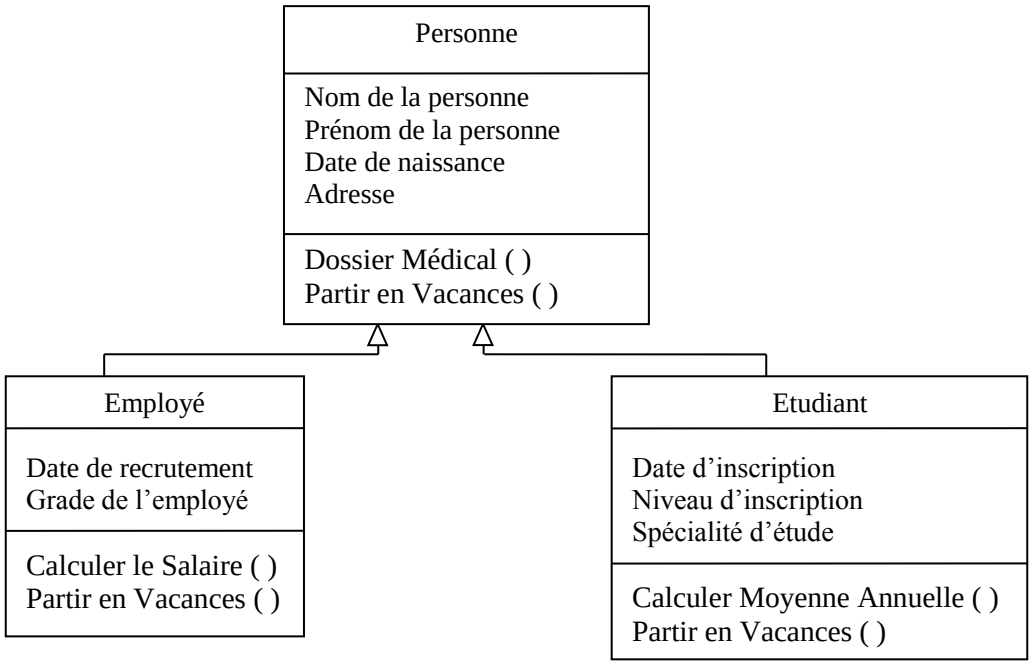
Exemple d'héritage multiple:



La classe « Véhicule Amphibie » hérite à la fois de la classe « Véhicule Terrestre » et à la fois de la classe « Véhicule Marin ».

2.2.3. Polymorphisme : Il consiste, tout en gardant le même nom pour une méthode héritée, à associer un code spécifique qui vient ainsi se substituer à celui de la méthode héritée.

Exemple :
Si l'on reprend l'exemple de l'héritage simple, l'opération « Partir en Vacances ( ) » dont héritent les classes « Employé » et « Etudiant » pourrait avoir des implémentations différentes pour ces deux sous-classes. D'où, il serait préférable d'implémenter cette opération par deux méthodes différentes : une pour la sous-classe « Employé » et une pour la sous-classe « Etudiant ».

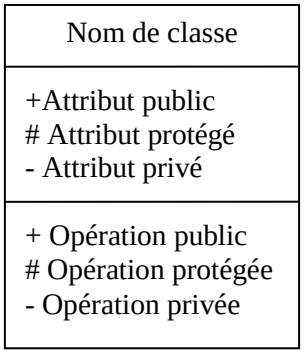


2.2.4. Encapsulation : L'encapsulation est un mécanisme consistant à rassembler les données et les méthodes au sein d'une structure en cachant l'implémentation de l'objet. L'encapsulation permet de définir des niveaux de visibilité des éléments de la classe.

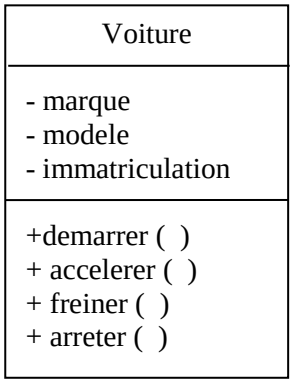
On distingue trois niveaux différents de visibilité *:* public, protégé et privé. L'encapsulation est représentée par un signe plus « + » dans le cas de pubic, un signe moins « - » dans le cas de privé et un dièse « # » dans le cas de protégé. Le tableau suivant détaille la signification de ces signes :

public	+	élément non encapsulé visible par tous
protégé	#	élément encapsulé visible dans les sous-classes de la classe
privé	-	élément encapsulé visible seulement dans la classe

Notation graphique :



Exemple :



3. Conclusion

Ce chapitre introduit la pensée orientée objet. En résumé :

- L'approche orientée objet permet de modéliser son application sous la forme d'interactions entre objets.
- Les objets ont des propriétés et peuvent faire des actions.
- Les objets masquent la complexité d'une implémentation grâce à l'encapsulation.
- La notion d'héritage correspond à la réutilisation d'un concept pour la spécification d'un nouveau concept, soit par la spécialisation d'un concept plus général en plusieurs concepts plus spécifiques, soit par la généralisation de plusieurs concepts dans un concept plus général.

4. Exercices

Exercice 1 :

L'université comporte des personnels administratifs et techniques, des enseignants, des étudiants et des chercheurs (qui sont tous des personnes). Certains étudiants peuvent être des chercheurs (les doctorants) ou des enseignants (les assistants enseignants). Certaines personnes (étudiants ou non) peuvent être à la fois chercheurs et enseignants.

- Proposer un modèle conceptuel (diagramme de classes) qui répond à la description fournie ci-dessus.

Exercice 2 :

Une carte géographique est caractérisée par une échelle, la longitude et la latitude de son coin inférieur gauche, la hauteur et la largeur de la zone couverte par la carte.

La carte comporte un ensemble de données géographiques de natures diverses :

Les villes et les montagnes sont repérées par un point unique. Chaque point a 2 coordonnées x et y calculées par rapport au coin inférieur gauche de la carte.

Un nom est associé à chaque donnée géographique repérée par un point.

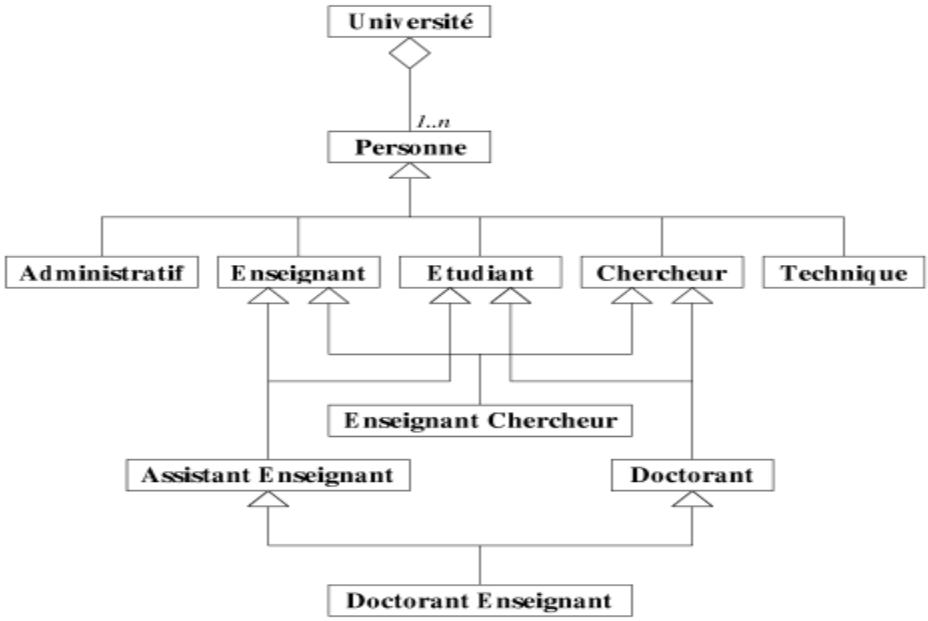
Les routes et les rivières sont repérées par des lignes brisées, c'est à dire par un ensemble de points correspondant aux extrémités de ses segments de droite. Les routes et les rivières ont des noms et des épaisseurs de trait.

Les lacs, mers et forêts sont représentés par des régions caractérisées par un nom et une couleur de remplissage. Une région est une ligne brisée refermée sur elle-même.

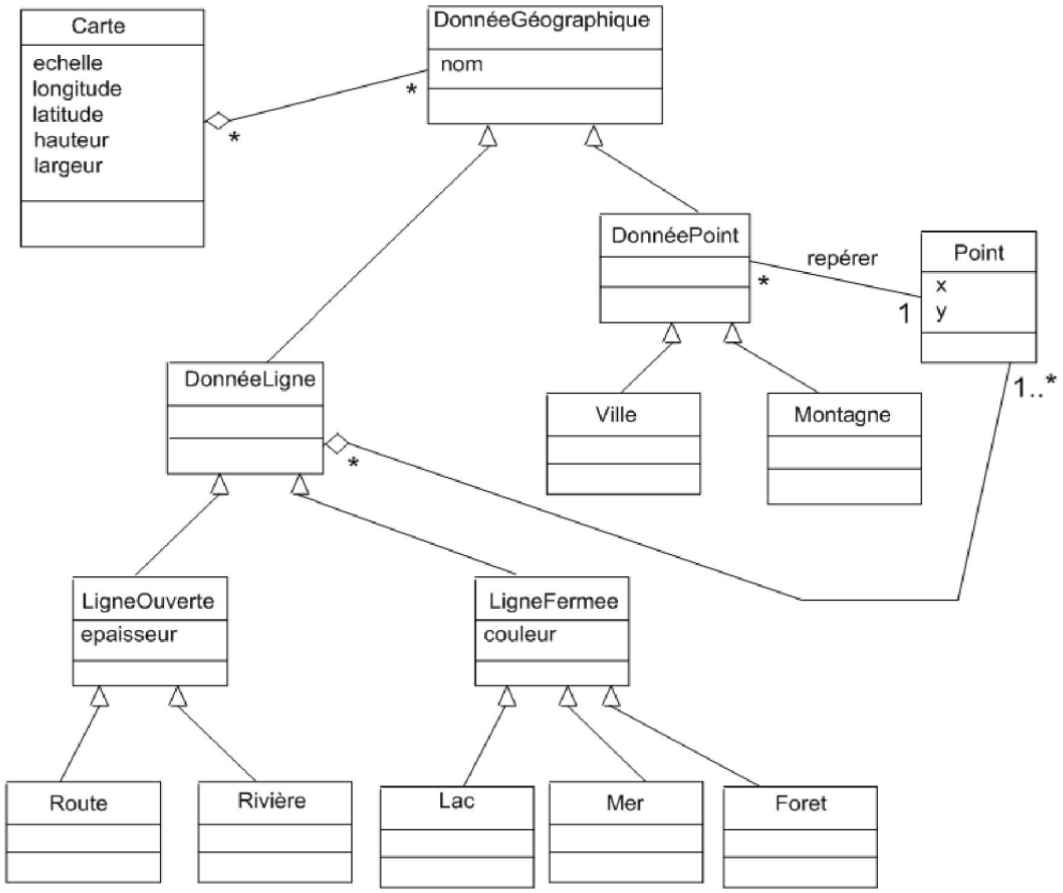
- Proposer un diagramme de classes modélisant les spécifications ci-dessus.

5. Corrigés des exercices

Corrigé de l'exercice 1



Corrigé de l'exercice 2



Chapitre 3: UML (Unified Modeling Language)

1. Introduction

L'objectif principal de ce chapitre est de permettre à l'étudiant de découvrir le langage UML (Unified Modeling Language ou langage de modélisation unifié).

UML est un langage de modélisation permettant la spécification, la construction, la visualisation et la documentation de logiciels.

1.1. Un bref historique

Les grandes étapes de la diffusion d'UML peuvent se résumer comme suit :

1994 - 1996 : rapprochement des méthodes OMT[4], BOOCH[5] et OOSE[6] et naissance de la première version d'UML.
23 novembre 1997 : version 1.1 d'UML adoptée par l'OMG[7].
1998 - 1999 : sortie des versions 1.2 à 1.3 d'UML.
2000 - 2001 : sortie des dernières versions suivantes 1.x.
2002 - 2003 : préparation de la version 2.
10 octobre 2004 : sortie de la version 2.1.
5 février 2007 : sortie de la version 2.1.1.
Les travaux d'amélioration d'UML 2 se poursuivent …
UML s'articule autour de plusieurs types de diagrammes, (le lecteur intéressé peut avoir plus de détails dans [3, 4, 5, 6, 7, 8]).

2. Diagrammes de classes

Ce diagramme représente la description statique du système en intégrant dans chaque classe la partie dédiée aux données et celle consacrée aux traitements. C'est le diagramme pivot de l'ensemble de la modélisation d'un système.

[4]OMT : Object Modeling Technique.
[5]Booch : Le nom de cette méthode vient de celui de son concepteur GradyBooch.
[6]OOSE : Object Oriented Software Engineering.
[7]OMG : Object Management Group.

2.1. Caractéristiques des attributs

La syntaxe d'un attribut est la suivante :

[Visibilité] nom [: type] [multiplicité] [= valeurParDefaut]

Où, les crochets indiquent les clauses facultatives : seul nom est obligatoire.

- *Nom d'attribut* : nom donné à l'attribut.
- *Type* : type de l'attribut.
- *Visibilité* : se reporter aux explications données à la section 2.2.4. du chapitre 2.
- *Multipilicité* : indique le nombre de valeurs possibles de l'attribut pour un objet.
- *valeurParDefaut* : valeur par défaut associée à l'attribut d'un objet s'il n'y a pas de valeur spécifiée à sa création.

2.2. Caractéristiques des opérations

La syntaxe d'une opération est la suivante :

[Visibilité] nom ([liste de paramètres]) [: type]

Où,

- *Visibilité* : se reporter aux explications données à la section 2.2.4. du chapitre 2.
- *Nom* : nom donné à l'opération.
- *liste de paramètres* : définition d'un ou de plusieurs paramètres. L'absence de paramètre est indiquée par ( ).
- *Type* : type de valeur retournée par l'opération. Une opération qui ne retourne pas de valeur est indiquée par exemple par le mot réservé « void » dans le langage C++ ou Java.

2.3. Association entre classes

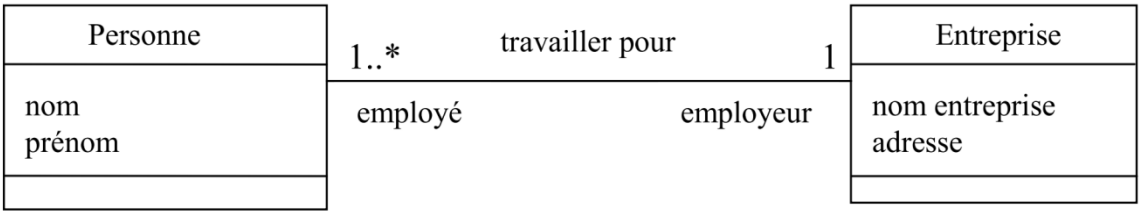
Une association entre classes représente les liens qui existent entre les instances de ces classes. Une association peut être identifiée par son nom. Il est possible d'exprimer les multiplicités (cardinalités) sur le lien d'association.

La multiplicité indique un domaine de valeurs pour préciser le nombre d'instance d'une classe vis-à-vis d'une autre classe pour une association donnée.

Notation des multiplicités d'association :

1	un et un seul
0..1	zéro ou un
*	de zéro à plusieurs
0..*	de zéro à plusieurs
1..*	De un à plusieurs
N	Exactement N
M..N	de M à N (entier)

Il est possible de préciser le rôle d'une classe au sein d'une association. Le rôle est placé à une extrémité du lien d'association, il se distingue ainsi du nom de l'association situé au centre du lien.

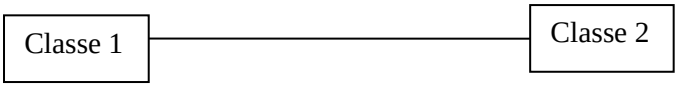


Dans cet exemple, une personne travaille pour une et une seule entreprise. L'entreprise emploie au moins une personne. L'entreprise est l'employeur des personnes qui travaillent pour elle et une personne a un statut d'employé dans l'entreprise.

Une association peut être binaire ou n-aire :

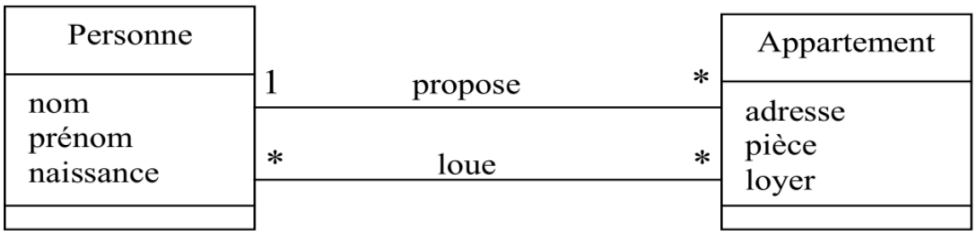
1) Association binaire : Une association binaire est représentée par un simple trait.

Notation graphique :



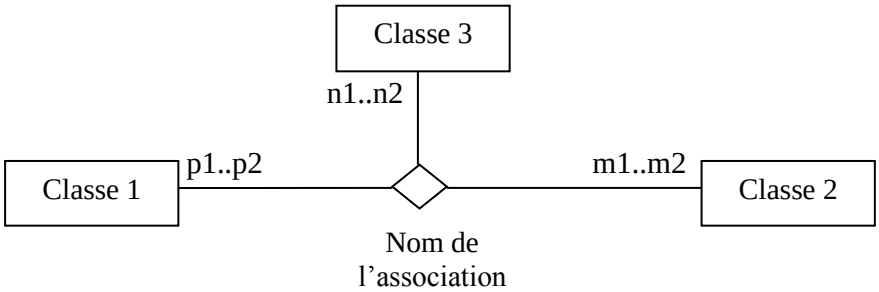
Remarque : On peut avoir plusieurs associations entre deux classes.

Exemple :



2) Association n-aire : Une association n-aire lie plus de deux classes. Elle est graphiquement représentée 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.

Exemple d'association ternaire :

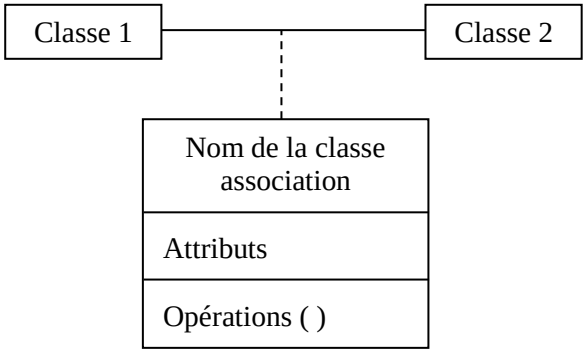


Pour une association ternaire, les cardinalités se lisent de la façon suivante : Pour un couple d'instances de la classe 1 et de la classe 2, il y a au minimum n1 instances de la classe 3 et au maximum n2.

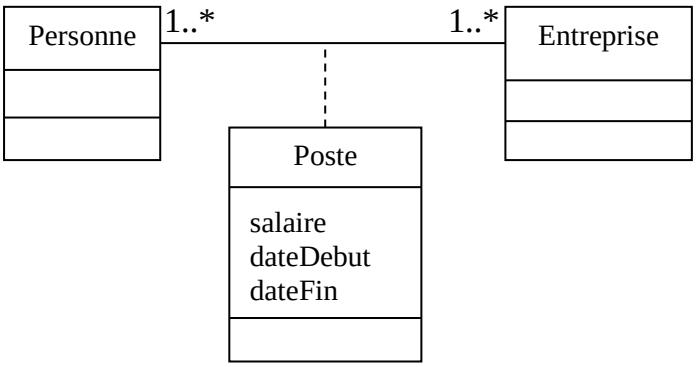
2.4. Classe association

Une classe d'association est nécessaire quand une association doit posséder ces propres propriétés.

Notation graphique :



Exemple :



<u>Remarque</u> :

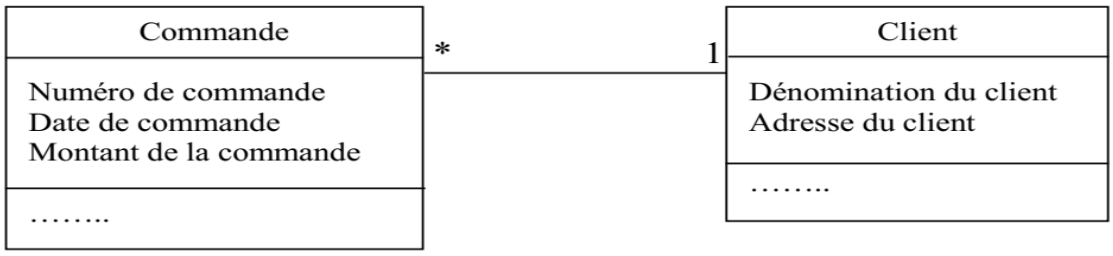
Les notations graphiques et les exemples présentés dans le chapitre 2 « Introduction à la conception orientée objet » font partie de la notation du diagramme de classes.

3. Diagramme d'objets

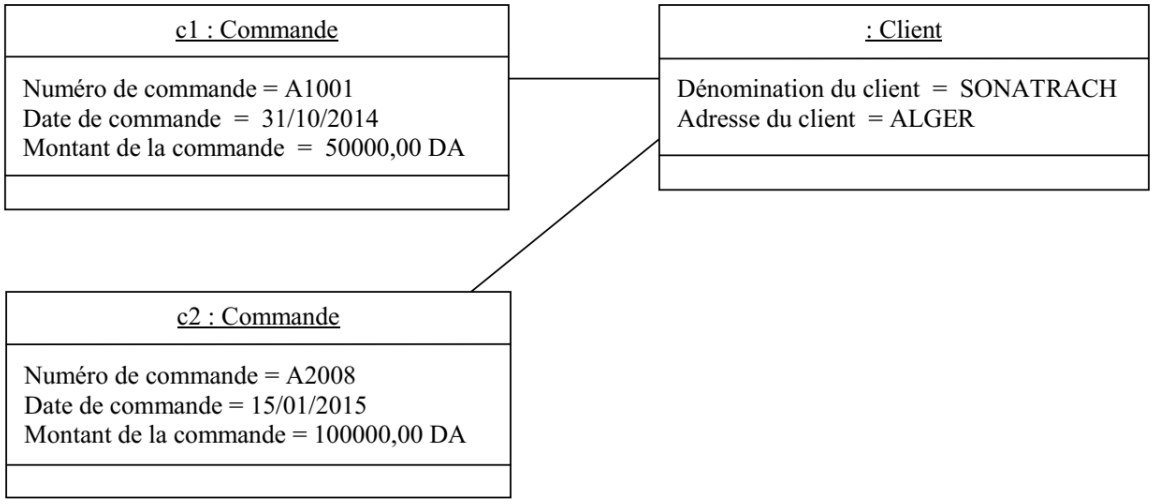
Le diagramme d'objets permet la représentation d'instances des classes et des liens entre instances.

Exemple :

Dans le diagramme de classes ci-dessous, nous représentons un lien entre deux classes Commande et Client.



Si nous nous intéressons à un client en particulier pour voir les commandes qu'il a effectué, nous représentons ceci par un diagramme d'objets donné par la figure suivante :



Remarque :

Le nom[8] d'un objet peut être désigné sous trois formes :

nom de l'objet, désignation directe et explicite d'un objet ;

nom de l'objet : nom de la classe, désignation incluant le nom de la classe ;

: nom de la classe, désignation anonyme d'un objet d'une classe donnée.

4. Diagramme de cas d'utilisation

Ce diagramme est destiné à représenter les besoins des utilisateurs par rapport au système.

[8] Le nom d'un objet est souligné.

4.1. Concepts de base

La représentation d'un diagramme de cas d'utilisation met en jeu trois concepts : l'acteur, le cas d'utilisation et l'interaction entre l'acteur et le cas d'utilisation.

a) Acteur

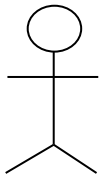
Un acteur représente un rôle joué par une entité externe (utilisateur humain, dispositif matériel ou autre système) qui interagit directement avec le système étudié.

Remarque :

- Plusieurs utilisateurs peuvent jouer le même rôle, ils correspondent dans ce cas à un seul acteur.
- Une même personne peut jouer différents rôles et correspond dans ce cas à plusieurs acteurs.

Notation graphique :

Un acteur se représente par un petit bonhomme ayant son nom inscrit dessous.



Nom de l'acteur

Un acteur peut aussi être formalisé par un rectangle contenant le mot clé « acteur » avec son nom juste en dessous.

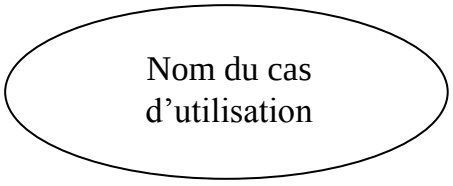
« acteur »
Nom de l'acteur

b) Cas d'utilisation

Un cas d'utilisation correspond à un certain nombre d'actions que le système devra exécuter en réponse à un besoin d'un acteur.

Notation graphique :

Un cas d'utilisation se représente par un ovale dans lequel figure son intitulé.

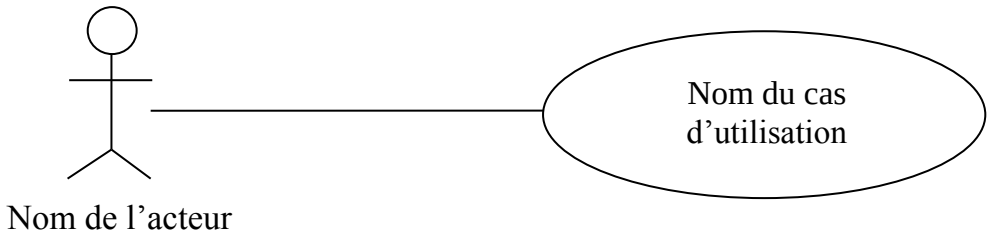


c) Interaction

Une interaction permet de décrire les échanges entre un acteur et un cas d'utilisation.

Notation graphique :

L'interaction entre un acteur et un cas d'utilisation se représente comme une association représentée par une ligne. Elle **peut** comporter des multiplicités comme toute association entre classes.



Remarque :

- Les cas d'utilisation **peuvent être** contenus dans un cadre qui représente les limites du système. Le nom du système figure à l'intérieur du cadre, en haut. Les acteurs sont alors forcément à l'extérieur du cadre puisqu'ils ne font pas partie du système.
- L'acteur est dit principal pour un cas d'utilisation lorsque le cas d'utilisation rend service à cet acteur. Un acteur secondaire est sollicité pour des informations complémentaires. Par convention et dans la mesure du possible ; Les acteurs principaux sont situés à gauche du cas d'utilisation. Les acteurs secondaires sont situés à droite du cas d'utilisation.

4.2. Relations entre cas d'utilisation

Trois relations peuvent être décrites entre cas d'utilisation : une relation d'inclusion, une relation d'extension et une relation de généralisation.

a) Relation d'inclusion

Dans une relation d'inclusion entre cas d'utilisation, une instance du cas d'utilisation source comprend (inclut) également le comportement décrit par le cas d'utilisation destination. Cette relation permet aussi de décomposer des comportements et de définir des comportements partageables entre plusieurs cas d'utilisation.

La relation d'inclusion est représentée par une flèche en traits pointillés et par le mot clé « inclut » (ou « include » en anglais) placé à proximité de cette flèche.

b) Relation d'extension

Une relation d'extension entre cas d'utilisation signifie que le cas d'utilisation source étend le comportement du cas d'utilisation destination.

La relation d'extension est représentée par une flèche en traits pointillés et par le mot clé « étend » (ou « extend » en anglais) placé à proximité de cette flèche.

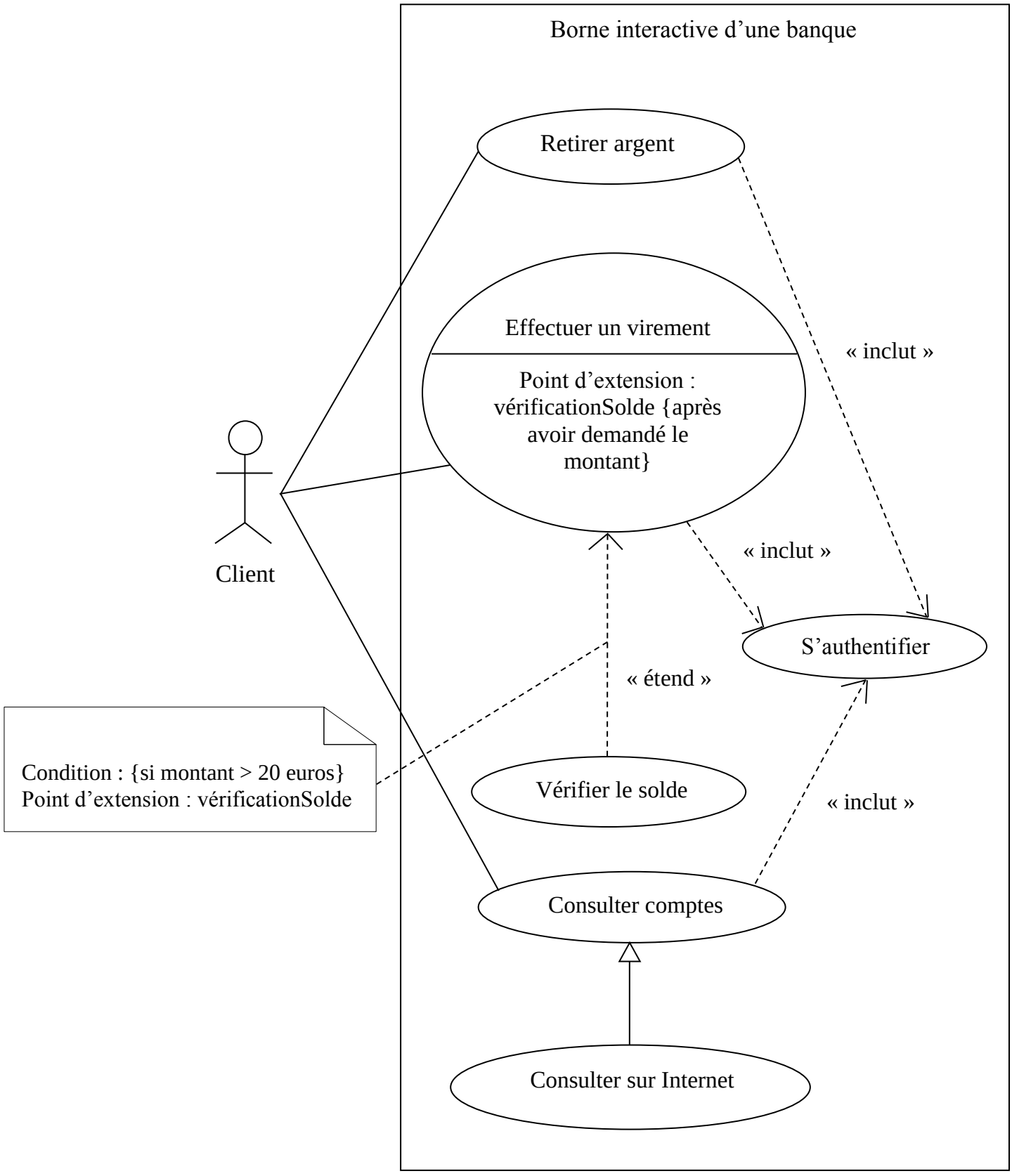
Remarque :

- L'extension peut intervenir à un point précis du cas étendu. Ce point s'appelle 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. L'extension peut être soumise à une condition.
  Graphiquement, la condition est exprimée sous la forme d'une note.
- Une contrainte définit des propositions devant être maintenues à vraies pour garantir la validité du système modélisé. On représente une contrainte sous la forme d'une chaîne de texte placée entre accolades { }.
- Une note en UML est une annotation qui peut concerner n'importe quel élément UML. Une note contient une information textuelle comme un commentaire, un corps de méthode ou une contrainte. Graphiquement, elle est représentée par un rectangle dont l'angle supérieur droit est plié. Une note n'indique pas explicitement le type d'élément qu'elle contient, toute l'intelligibilité d'une note doit être contenue dans le texte même. On peut relier une note à l'élément qu'elle décrit grâce à une ligne en pointillés.

c) Relation de généralisation

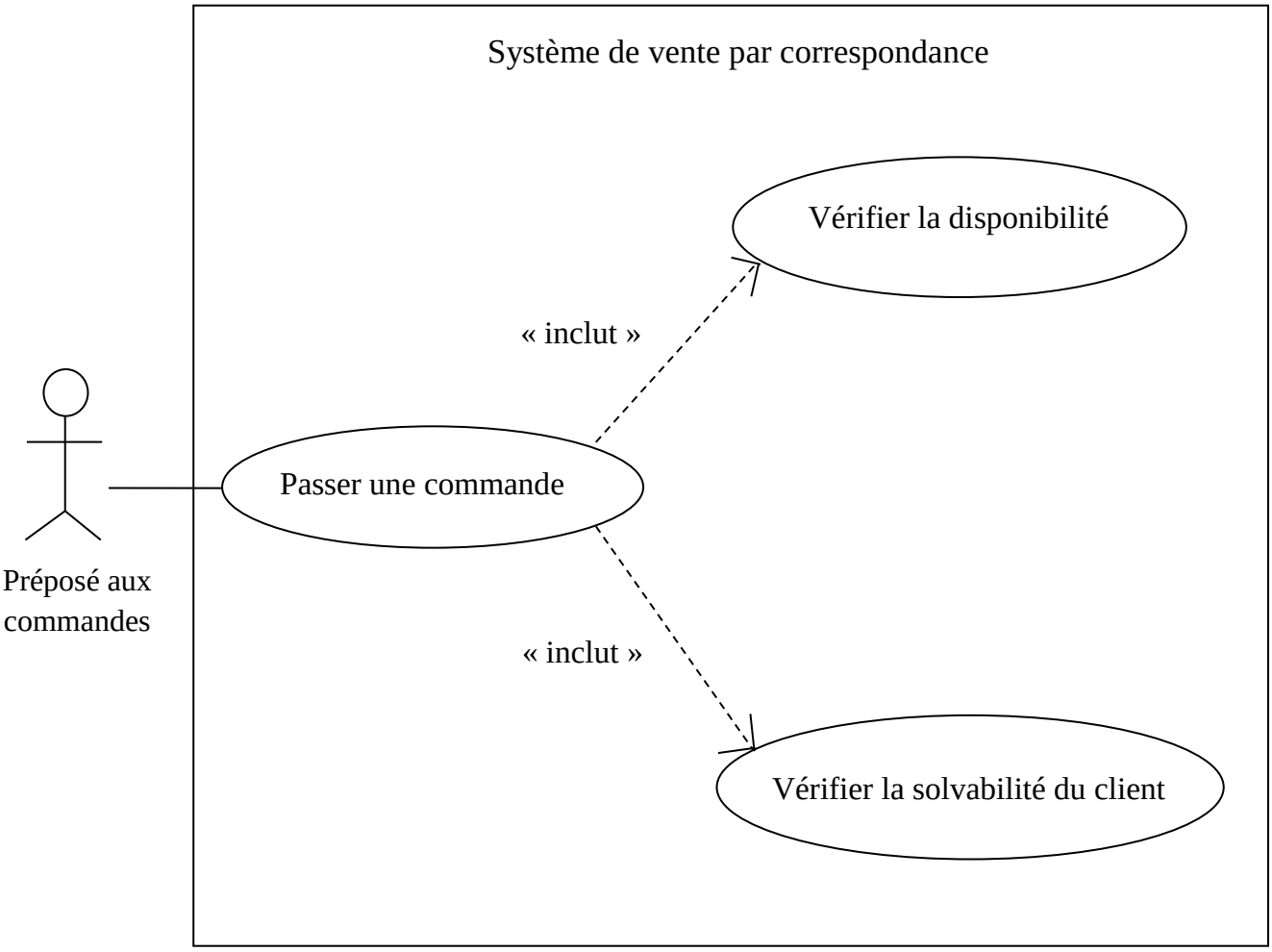
Une relation de généralisation de cas d'utilisation peut être définie conformément au principe de la spécialisation-généralisation déjà présentée pour les classes.

Exemple : (Relations entre cas dans un diagramme de cas d'utilisation)



<u>Remarque</u> : Quand un cas n'est pas directement relié à un acteur, il est qualifié de cas d'utilisation interne.

Exemple : (Relations entre cas pour décomposer un cas complexe)

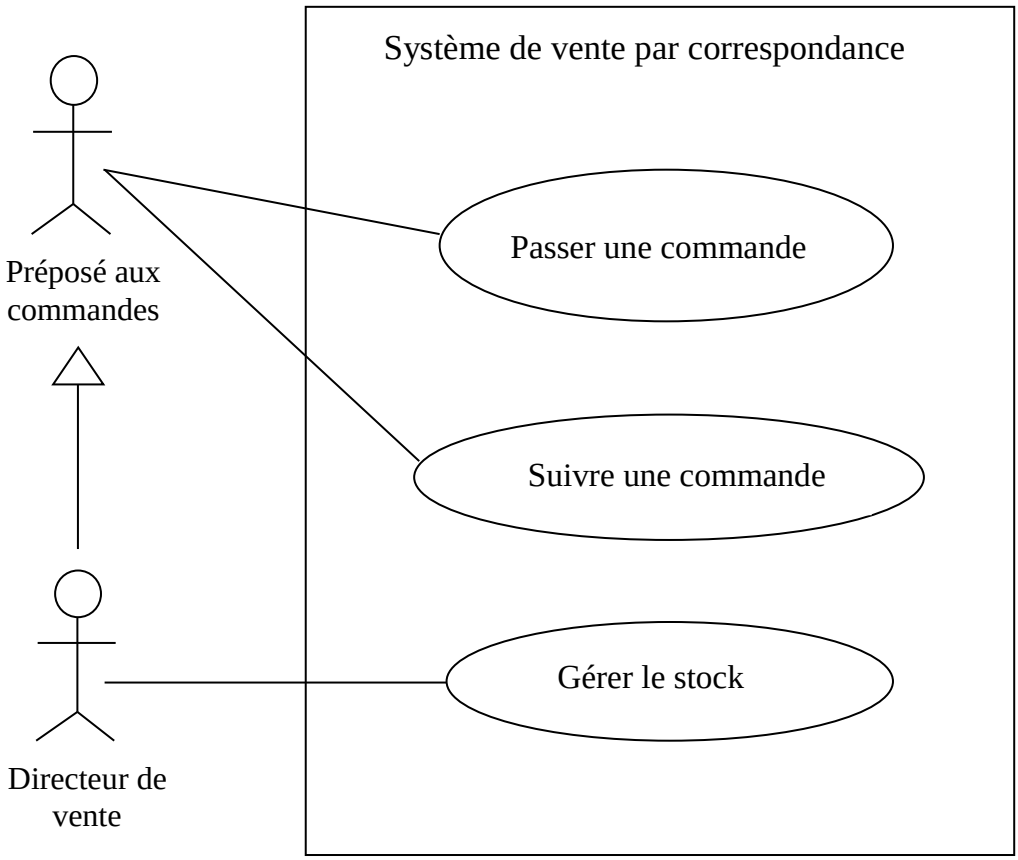


Remarque : Les cas d'utilisation ne s'enchaînent pas, car il n'y a aucune représentation temporelle dans un diagramme de cas d'utilisation.

4.3. Relations entre acteurs

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

Le symbole utilisé pour la généralisation entre acteurs est une flèche avec un trait plein dont la pointe est un triangle fermé désignant l'acteur le plus général.



Le directeur des ventes est un préposé aux commandes avec un pouvoir supplémentaire : en plus de pouvoir passer et suivre une commande, il peut gérer le stock. Par contre, le préposé aux commandes ne peut pas gérer le stock.

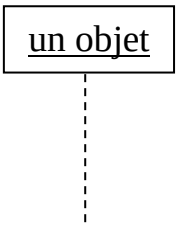
5. Diagramme de séquence

L'objectif du diagramme de séquence est de représenter les interactions entre objets en indiquant la chronologie des échanges.

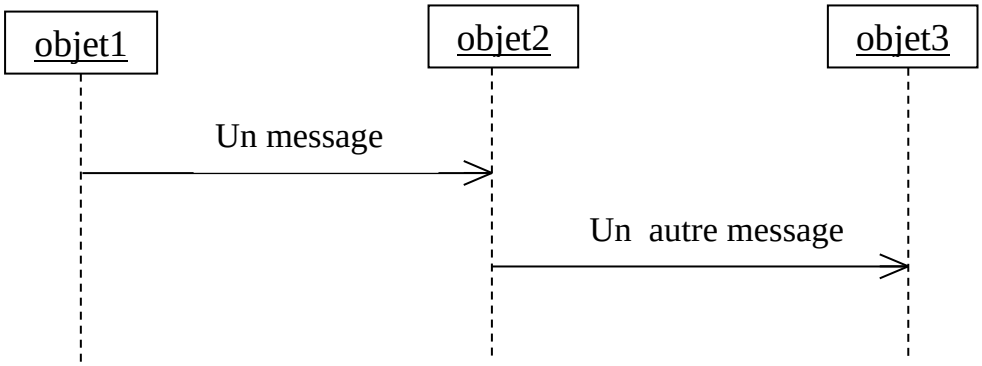
5.1. La ligne de vie des objets

La ligne de vie des objets est représentée par une ligne verticale en traits pointillés, placée sous le symbole de l'objet concerné. Cette ligne de vie précise l'existence de l'objet concerné durant un certain laps du temps.

Notation graphique :



5.2. Exemple de diagramme de séquence

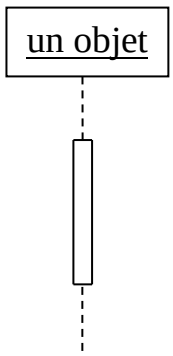


- L'ordre d'envoi d'un message est déterminé par sa position sur l'axe vertical du diagramme ; le temps s'écoule « de haut en bas » de cet axe.
- La disposition des objets sur l'axe horizontal n'a pas de conséquence pour la sémantique du diagramme.
- Les messages sont étiquettés par le nom de l'opération ou du signal invoqué.

5.3. Barre d'activation

Les diagrammes de séquence permettent de représenter les périodes d'activité des objets. Les périodes d'activité se représentent par des bandes rectangulaires placées sur les lignes de vie. Le début et la fin d'une bande correspondent respectivement au début et à la fin d'une période d'activité.

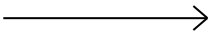
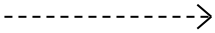
Notation graphique :



5.4. Message synchrone et asynchrone

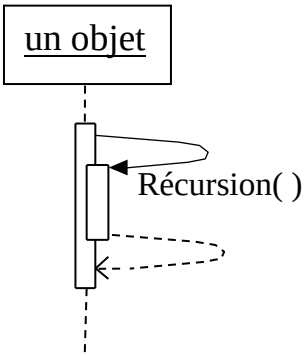
Message synchrone : Dans ce cas l'émetteur reste en attente de la réponse à son message avant de poursuivre ses actions. Le message de retour **peut** ne pas être représenté car il est inclus dans la fin d'exécution de l'opération de l'objet destinataire du message.

Message asynchrone : Dans ce cas, l'émetteur n'attend pas la réponse à son message, il poursuit l'exécution de ses opérations.

Messages	Notation graphique
Synchrone	
Asynchrone	
Retour	

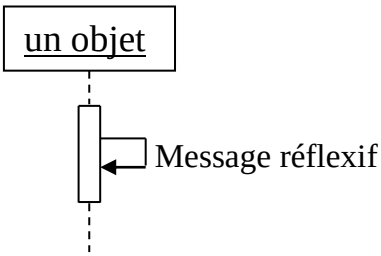
Remarque : Le cas particulier des envois de messages récursifs se représente par un dédoublement de la bande rectangulaire. L'objet apparait alors comme s'il était actif plusieurs fois.

Exemple :



Remarque : Un objet peut également s'envoyer un message. Cette situation se représente par une flèche qui revient en boucle sur la ligne de vie de l'objet.

Exemple :



Remarque :

En modélisation objet, les diagrammes de séquence sont utilisés de deux manières différentes, selon la phase du cycle de vie et le niveau de détail désiré.

- La première utilisation correspond à la documentation des cas d'utilisation ; elle se concentre sur la description de l'interaction, souvent dans des termes proches de

l'utilisation et sans entrer dans les détails de la synchronisation. L'indication portée sur les flèches correspond alors à des événements qui surviennent dans le domaine de l'application. À ce stade de la modélisation, les flèches ne traduisent pas encore des envois de message au sens des langages de programmation.

- La deuxième utilisation correspond à un usage plus informatique. Le concept de message unifie toutes les formes de communication entre objets (appel de procédure par exemple).

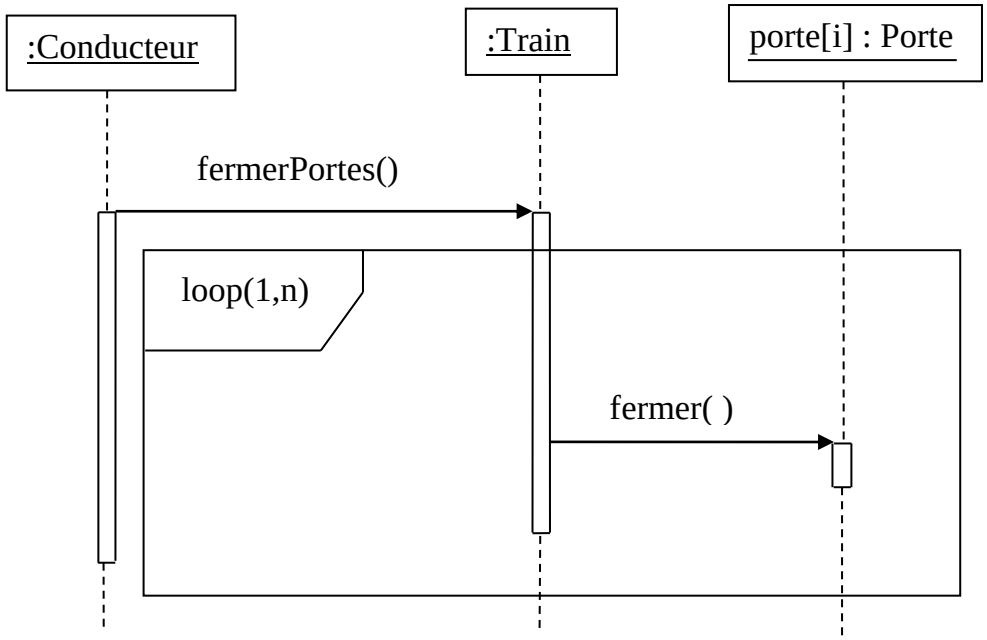
5.5. Fragment d'interaction

Un fragment d'interaction dit combiné correspond à un ensemble d'interaction auquel on applique un opérateur. Plusieurs opérateurs ont été définis dans UML : loop, alt, opt, etc.

- **Opérateur loop**

L'opérateur loop correspond à une instruction de boucle.

Exemple :



- **Opérateur alt**

L'opérateur alt correspond à une instruction de test avec une ou plusieurs alternatives possibles. Il est aussi permis d'utiliser les clauses de type sinon (else).

L'opérateur alt se représente dans un fragment possédant au moins deux parties séparées par des pointillés.

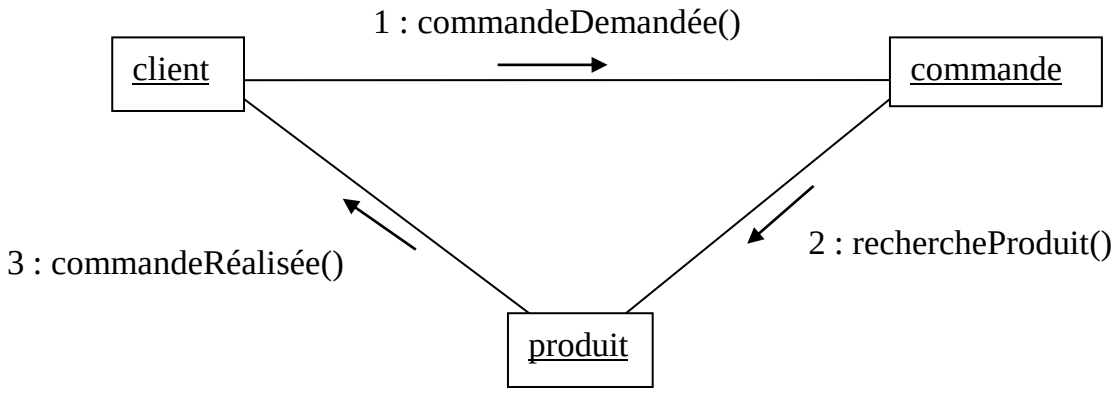
Remarque :

L'opérateur **opt** correspond à une instruction de test sans alternative (sinon). L'opérateur **opt** se représente dans un fragment possédant une seule partie.

6. Diagramme de communication

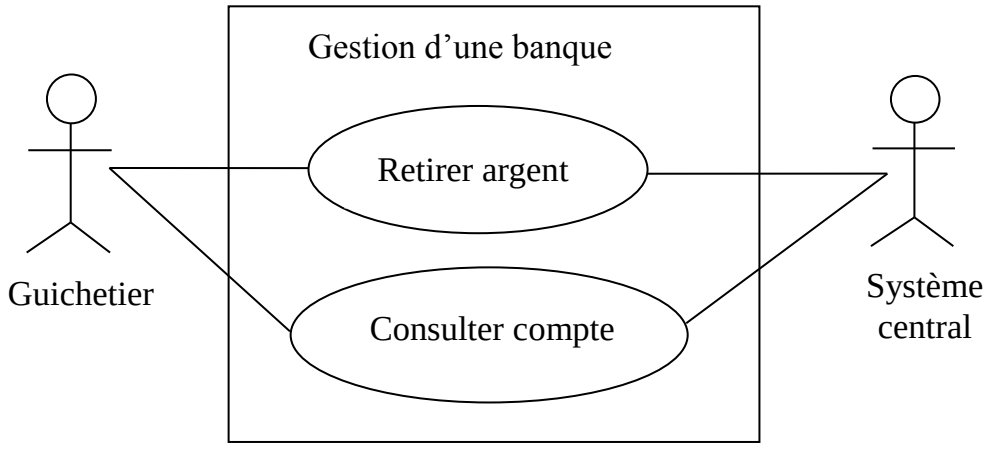
Le diagramme de communication constitue une autre représentation des interactions que celle du diagramme de séquence. En effet, le diagramme de communication met plus l'accent sur l'aspect spatial des échanges que l'aspect temporel.

Exemple :

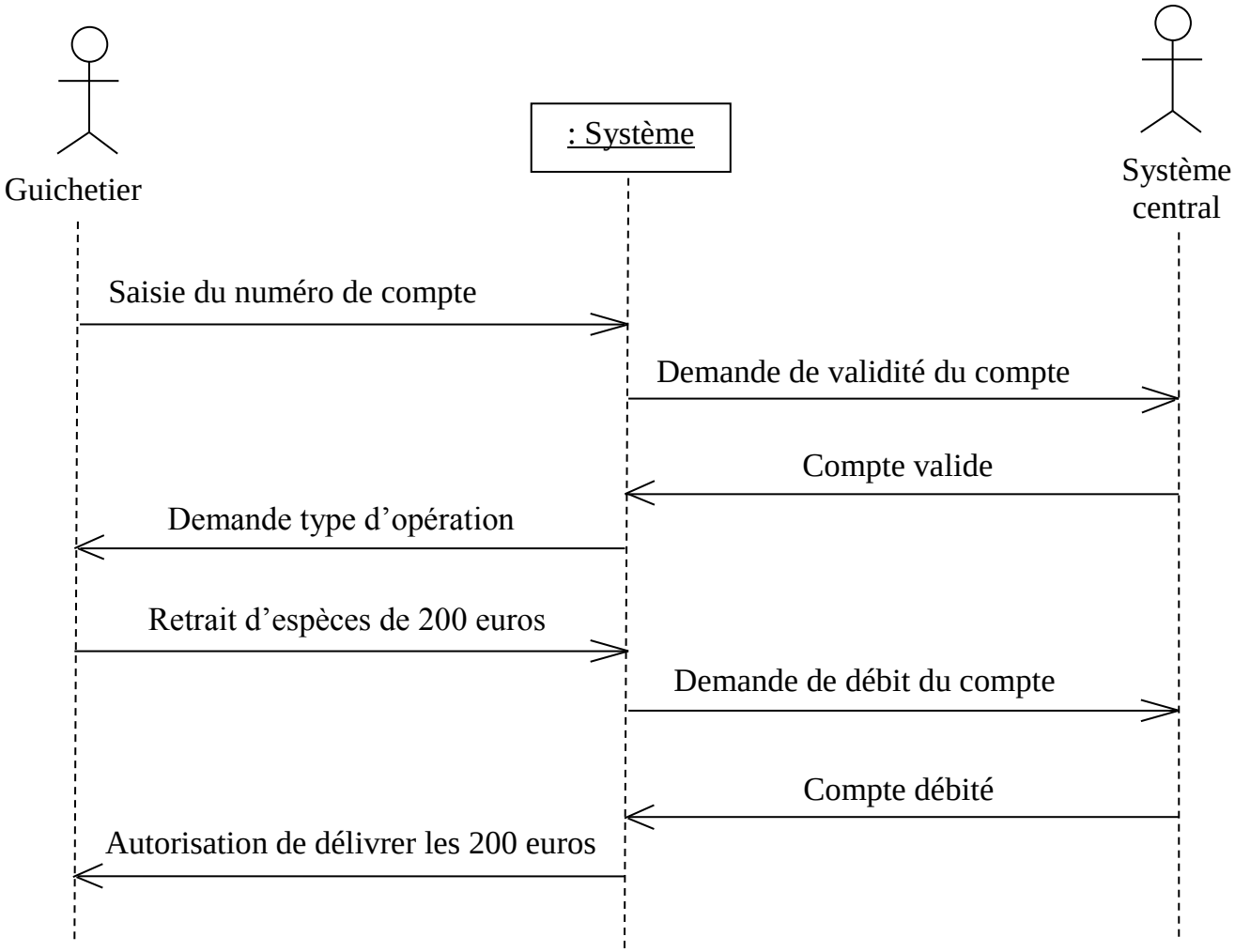


7. Description des cas d'utilisation

À partir du diagramme de cas d'utilisation ci-dessous :



La figure suivante montre une façon de décrire les interactions pour le retrait d'argent (retrait d'espèces en euros). On voit clairement apparaître une séquence de messages.



➢ Description textuelle des cas d'utilisation

La description d'un scénario par un diagramme de séquence n'empêche pas de décrire le cas sous forme textuelle, ce qui permet de faire figurer des renseignements supplémentaires.
Une description textuelle couramment utilisée se compose de trois parties. La première partie permet d'identifier le cas. Elle doit contenir :

- le nom du cas ;
- un résumé de son objectif ;
- les acteurs impliqués (principaux et secondaires) ;
- les dates de création et de mise à jour de la description courante ;
- le nom des responsables ;
- un numéro de version.

La deuxième partie contient la description du fonctionnement du cas sous la forme d'une séquence de messages échangés entre les acteurs et le système. Elle contient **toujours une séquence nominale** qui correspond au fonctionnement nominal du cas. Cette séquence nominale commence par préciser l'événement qui déclenche le cas et se développe en trois points :

- Les pré-conditions : Elles indiquent dans quel état est le système avant que se déroule la séquence.
- L'enchaînement des messages.
- Les post-conditions : Elles indiquent dans quel état se trouve le système après le déroulement de la séquence nominale.

Parfois la séquence correspondant à un cas a besoin d'être appelée dans une autre séquence. Signifier l'appel d'une autre séquence se fait de la façon suivante :
« appel du cas X », où X est le nom du cas.

Les acteurs n'étant pas sous le contrôle du système, ils **peuvent** avoir des comportements imprévisibles. La séquence nominale ne suffit donc pas pour décrire tous les comportements possibles. À la séquence nominale s'ajoutent fréquemment des séquences alternatives et des séquences d'exceptions. Ces deux types de séquences se décrivent de la même façon que la séquence nominale mais il ne faut pas les confondre. Une séquence alternative diverge de la séquence nominale (c'est un embranchement dans une séquence nominale) mais y revient toujours, alors qu'une séquence d'exception intervient quand une erreur se produit (le séquencement nominal s'interrompt, sans retour à la séquence nominale).
Dans le cas d'un retrait d'argent, des séquences alternatives se produisent par exemple dans les situations suivantes :

- Le client choisit d'effectuer un retrait en euros ou en dollars.
- Le client a la possibilité d'obtenir un reçu.

Une exception se produit si la connexion avec le système central de la banque qui doit vérifier la validité du compte est interrompue.
La survenue des erreurs dans les séquences doit être signalée de la façon suivante :
« appel de l'exception Y » où Y est le nom de l'exception.

La séquence nominale, les séquences alternatives, les exceptions, etc., font qu'il existe une multitude de chemins depuis le début du cas jusqu'à la fin. Chaque chemin est appelé

scénario. Un système donné génère peu de cas d'utilisation, mais, en général, beaucoup de scénarios.

La dernière partie de la description d'un cas d'utilisation est une rubrique optionnelle. Elle contient généralement des spécifications non fonctionnelles (ce sont le plus souvent des spécifications techniques, par exemple pour préciser que l'accès aux informations bancaires doit être sécurisé).

Description d'un retrait d'argent

Identification

Nom du cas : Retirer argent

But : détaille les étapes permettant à un guichetier d'effectuer l'opération de retrait d'euros demandé par un client.

Acteur principal : Guichetier.

Acteur secondaire : Système central.

Date : le 18/02/2015.

Responsable : M. Dupont.

Version : 1.0.

Séquencement

Le cas d'utilisation commence lorsqu'un client demande le retrait d'espèces en euros.

Pré-conditions

Le client possède un compte (donne son numéro de compte).

Enchaînement nominal

1. Le guichetier saisit le numéro de compte client.
2. L'application valide le compte auprès du système central.
3. L'application demande le type d'opération au guichetier.
4. Le guichetier sélectionne un retrait d'espèces de 200 euros.
5. L'application demande au système central de débiter le compte.
6. Le système (application) notifie au guichetier qu'il peut délivrer le montant demandé.

Post-conditions

Le guichetier ferme le compte.

Le client récupère l'argent.

Rubriques optionnelles

Contraintes non fonctionnelles

Fiabilité : les accès doivent être extrêmement sûrs et sécurisés.
Confidentialité : les informations concernant le client ne doivent pas être divulguées.
Contraintes liées à l'interface homme-machine.
Donner la possibilité d'accéder aux autres comptes du client.
Toujours demander la validation des opérations de retrait.

8. Diagramme d'états-transitions

Les diagrammes d'états-transitions décrivent le comportement interne d'un objet à l'aide d'un automate à états finis.

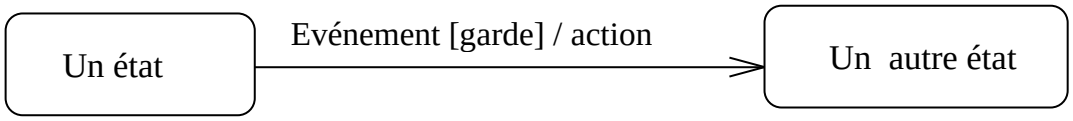
Notation graphique :
Les diagrammes d'états-transitions visualisent des automates d'état finis, du point de vue des états et des transitions. Les états sont représentés par des rectangles aux coins arrondis, tandis que les transitions sont représentées par des arcs orientés liant les états entre eux.

a) Etat :
   Notion abstraite montrant la façon dont réagit un objet à un événement.

b) Transition :
   Une transition représente le passage instantané d'un état vers un autre.

c) Etiquette :
   Une transition peut être déclenchée par un événement, conditionné à l'aide de « gardes » (expression booléennes), et/ou être associée à une action.

   La syntaxe de l'étiquette d'une transition est alors la suivante :



Remarque :

Toutes les parties de l'étiquette de la transition sont facultatives. L'absence d'étiquette indique qu'une transition est automatique.

8.1. L'état initial

L'état initial est représenté par un gros point noir.

Notation graphique : 

8.2. L'état final

L'état final est représenté par un gros point noir encerclé

Notation graphique : 

Remarque :

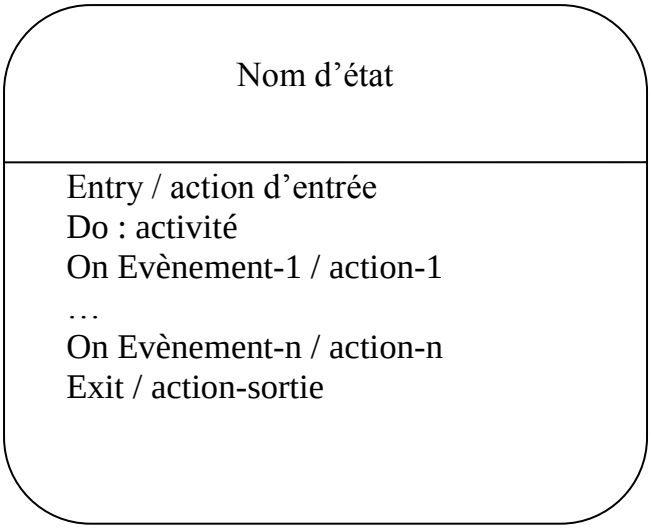
Pour un niveau hiérarchique donné, il y a toujours un et un seul état initial. En revanche, pour un niveau hiérarchique donné, il est possible d'avoir plusieurs états finaux, qui correspondent chacun a une condition de fin différente. Il est également possible de n'avoir aucun état final, dans le cas par exemple d'un système qui ne s'arrête jamais.

8.3. Forme générale d'un état

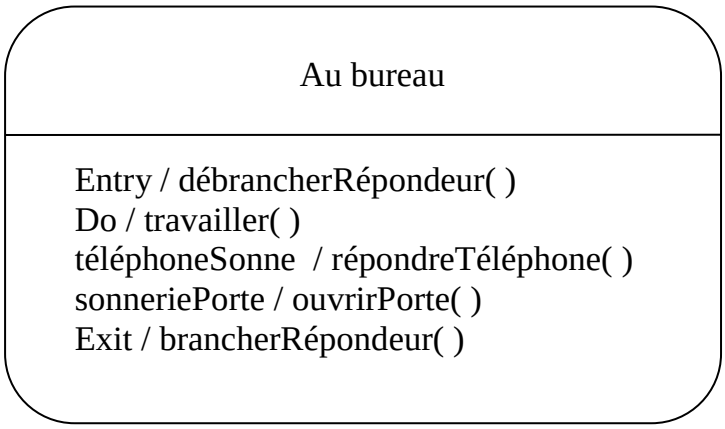
Les états peuvent également contenir des actions ; elles sont exécutées à l'entrée ou à la sortie de l'état ou lors de l'occurrence d'un événement pendant que l'objet est dans l'état en question. Comme les classes, les états peuvent être divisés en compartiments qui contiennent :

- Le non de l'état ;
- Les transitions internes, c'est-à-dire la liste des actions ou des activités internes effectuées pendant que l'élément est dans cet état.

Notation graphique :



Exemple :



Remarque :

Une action est une opération instantanée qui ne peut être interrompue ; elle est associée à une transition.

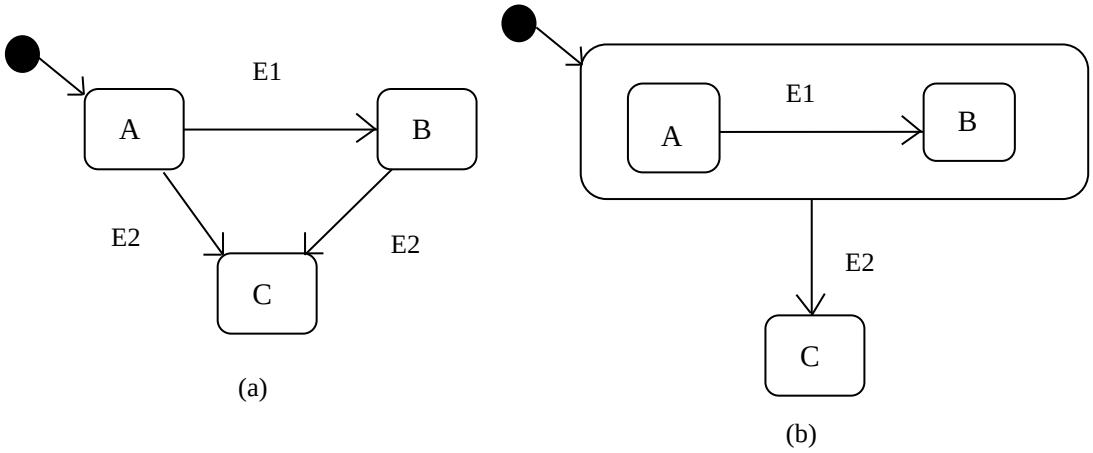
Une activité est une opération d'une certaine durée qui peut être interrompue, elle est associée à un état d'un objet.

8.4. Super-Etat ou généralisation d'états

Un super-état est un élément de structuration des diagrammes d'états-transitions (il s'agit d'un état qui englobe d'autres états et transitions).

Exemple :

On remarque dans la figure (a) que l'objet décrit par cet automate réagit de la même façon à l'arrivée de l'événement E2. Qu'il soit à l'état A ou l'état B, l'objet passe à l'état C. Nous pouvons généraliser ce comportement, en créant un super état englobant A et B.

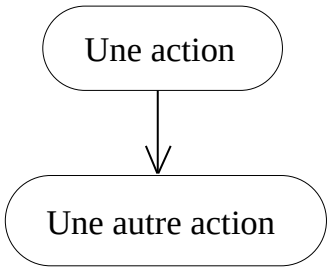


9. Diagramme d'activités

Les diagrammes d'activités décrivent le comportement d'une méthode, le déroulement d'un cas d'utilisation et/ou les enchainements d'activités.

Une activité désigne une suite d'actions. Le passage d'une action vers une autre est matérialisé par une transition. Les transitions sont déclenchées par la fin d'une action et provoquent le début immédiat d'une autre action (elles sont automatiques). Chaque action est représentée par un rectangle dont les coins sont très arrondis. Chaque action est libellée pour décrire ce qui est fait.

Notation graphique :

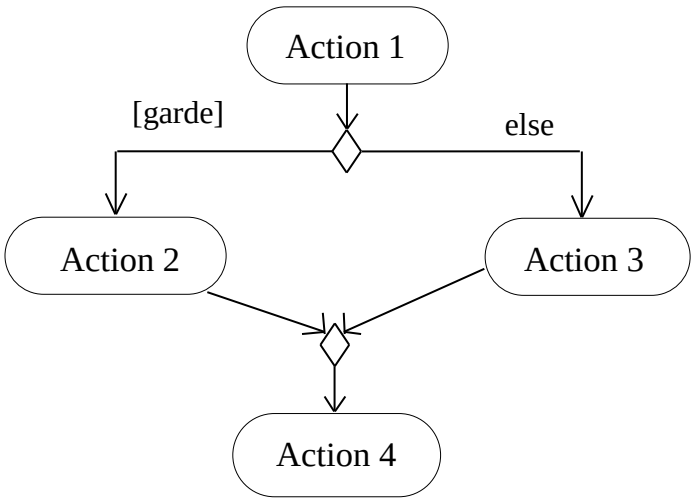


9.1. Décision / Fusion

Le comportement conditionnel est décrit par des décisions/fusions.

Une **décision** (ou branchement) permet de représenter des transactions conditionnelles en utilisant des gardes (expressions booléennes).

Une **fusion** marque la fin d'un comportement conditionnel.



9.2. Débranchement et jonction

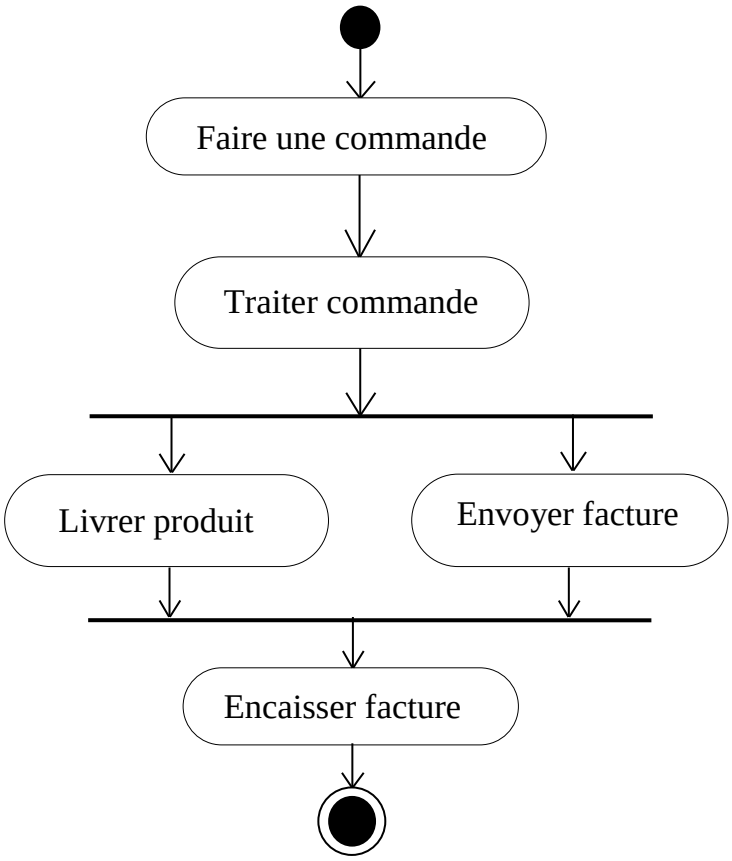
Il est possible de synchroniser les transitions à l'aide des barres de synchronisation. Ces dernières permettent d'ouvrir (**débranchement**) ou de fermer (**jonction**) des branches parallèles au sein d'un flot d'exécution.

Débranchement : Les transitions qui partent d'un branchement ont lieu en même temps.

Jonction : On ne franchit une jonction qu'après réalisation de toutes les transitions qui s'y rattachent.

Exemple :

La figure suivante décrit le traitement d'une commande par un diagramme d'activités.

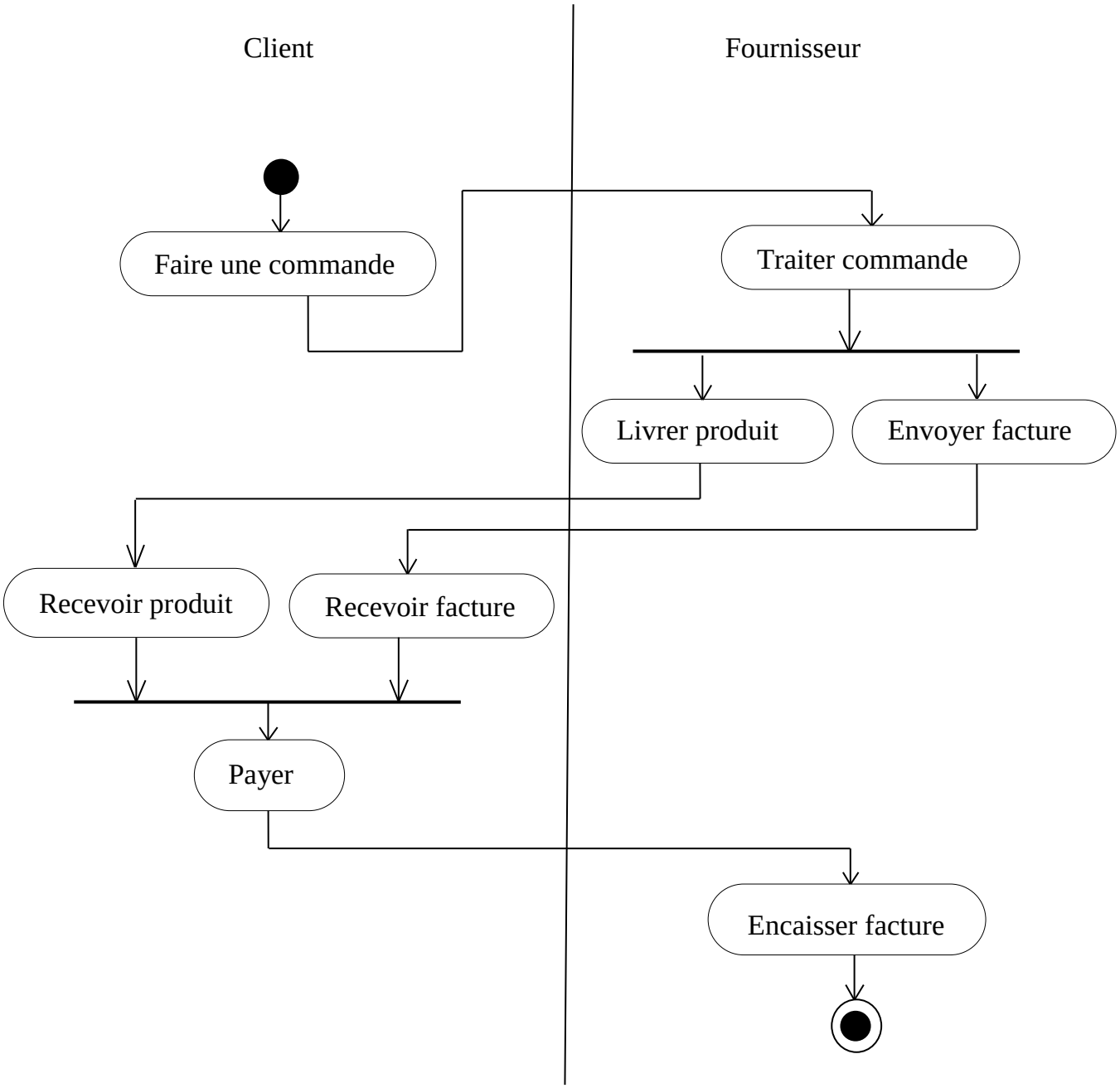


9.3. Couloir d'activités

Les couloirs d'activités servent à organiser un diagramme d'activités selon les acteurs ou responsables des activités représentées. Il est même possible d'identifier les objets principaux, qui sont manipulés d'activités en activités.

Exemple :

La figure suivante décrit le traitement d'une commande faite par un client à son fournisseur.



10. Conclusion

Dans ce chapitre, nous avons présenté le langage UML. Ce langage a été pensé pour être un langage de modélisation visuelle commun, et riche sémantiquement et syntaxiquement. Il est destiné à l'architecture, la conception et la mise en œuvre de systèmes logiciels complexes par leur structure aussi bien que leur comportement. L'UML se compose de différents types de diagrammes. Dans l'ensemble, les diagrammes UML décrivent la limite, la structure et le

comportement du système et des objets qui s'y trouvent. UML est le langage le plus standard et le plus adopté dans la communauté de développement de logiciels et il est parrainé et maintenu par l'organisation OMG[9].

11. Etude de cas

L'entreprise MegaKebab regroupe de nombreux restaurants appelés "Points Kebab". Elle est spécialisée dans la livraison à domicile de Kebabs et autres spécialités. Actuellement, les commandes se font par téléphone directement auprès de chaque restaurant. Un nombre limité de commandes peut être traité et chaque client doit connaître la carte des plats offerts par le Point Kebab contacté (ils varient d'un restaurant à l'autre). La direction de MegaKebab souhaite informatiser le processus de commande/fabrication/livraison via un logiciel baptisé CyberKebab.

Grâce à ce logiciel, MegaKebab souhaite gérer à distance et de manière centralisée toutes les commandes, les Points Kebab et les employés appelés "Collaborateurs". Cette centralisation doit permettre de rendre accessible sur Internet tous les plats disponibles. Chaque plat est décrit par un nom, une photo et un prix (identique partout). Dans le cadre de la politique marketing, une durée est également associée à chaque plat chaud : si le temps écoulé entre la fin de préparation et la livraison est supérieur à cette durée, le client peut se faire rembourser sa commande. Cependant, pour ne pas inciter les clients à utiliser cette possibilité, cette opération n'est pas disponible sur Internet : le client doit remplir une demande écrite sur papier libre et l'envoyer au gérant de MegaKebab. A tout moment il est possible de passer une commande par Internet. Le client doit disposer d'une carte de crédit qui l'identifie de manière unique. Lors d'une première commande il lui est également demandé de saisir son nom et de situer son lieu de résidence sur une carte de la ville. Une même commande peut comporter plusieurs plats. Pour chaque plat sélectionné le client doit indiquer la quantité désirée. Après avoir passé sa commande, le client peut à tout moment consulter l'état de sa commande. Tant que la commande n'est pas partie du PointKebab, il peut l'annuler.

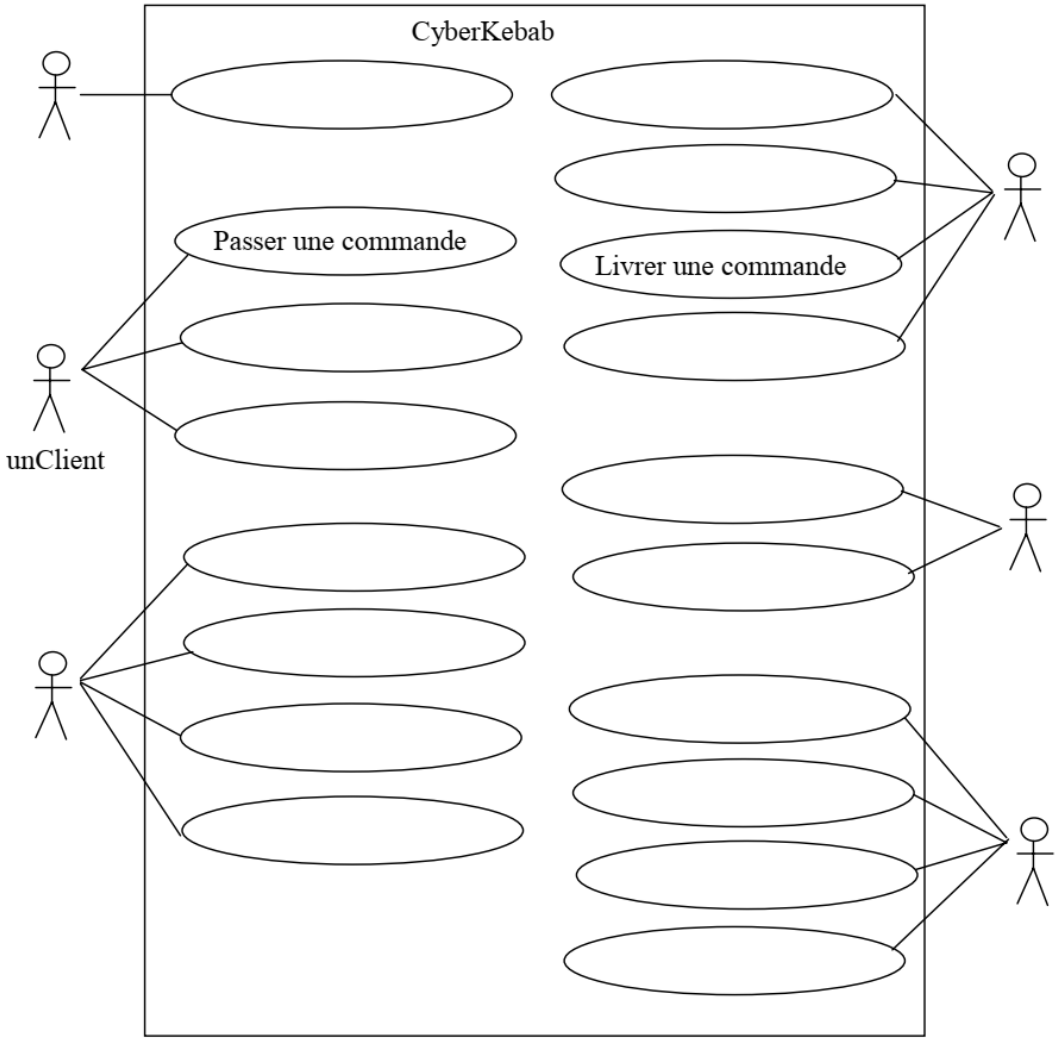
[9] https://www.omg.org/

Les Points Kebab sont ouverts 24h/24. Pour assurer un service 24h/24 dans toute la ville, MegaKebab fait appel à un grand nombre de collaborateurs, souvent étudiants, qui ont des horaires très flexibles. Lors de leur embauche, un téléphone portable leur est remis. Il suffit d'appuyer sur un bouton pour faire part de leur disponibilité auprès de MegaKebab. Un autre bouton permet d'indiquer qu'ils ne le sont plus. A tout moment le gérant peut consulter via Internet l'état du système global. Il peut affecter un collaborateur soit à un Point Kebab soit à la livraison. Un collaborateur peut ainsi changer de lieu de travail ou de rôle plusieurs fois dans une journée : le rôle du gérant est d'optimiser l'attribution de chacun en fonction des commandes. Lorsqu'un client passe une commande, il n'indique pas de PointKebab particulier; c'est le gérant qui affecte la commande à un PointKebab et à un livreur. Le gérant cherche en général à optimiser la distance parcourue ainsi que les activités des PointKebabs et des collaborateurs.

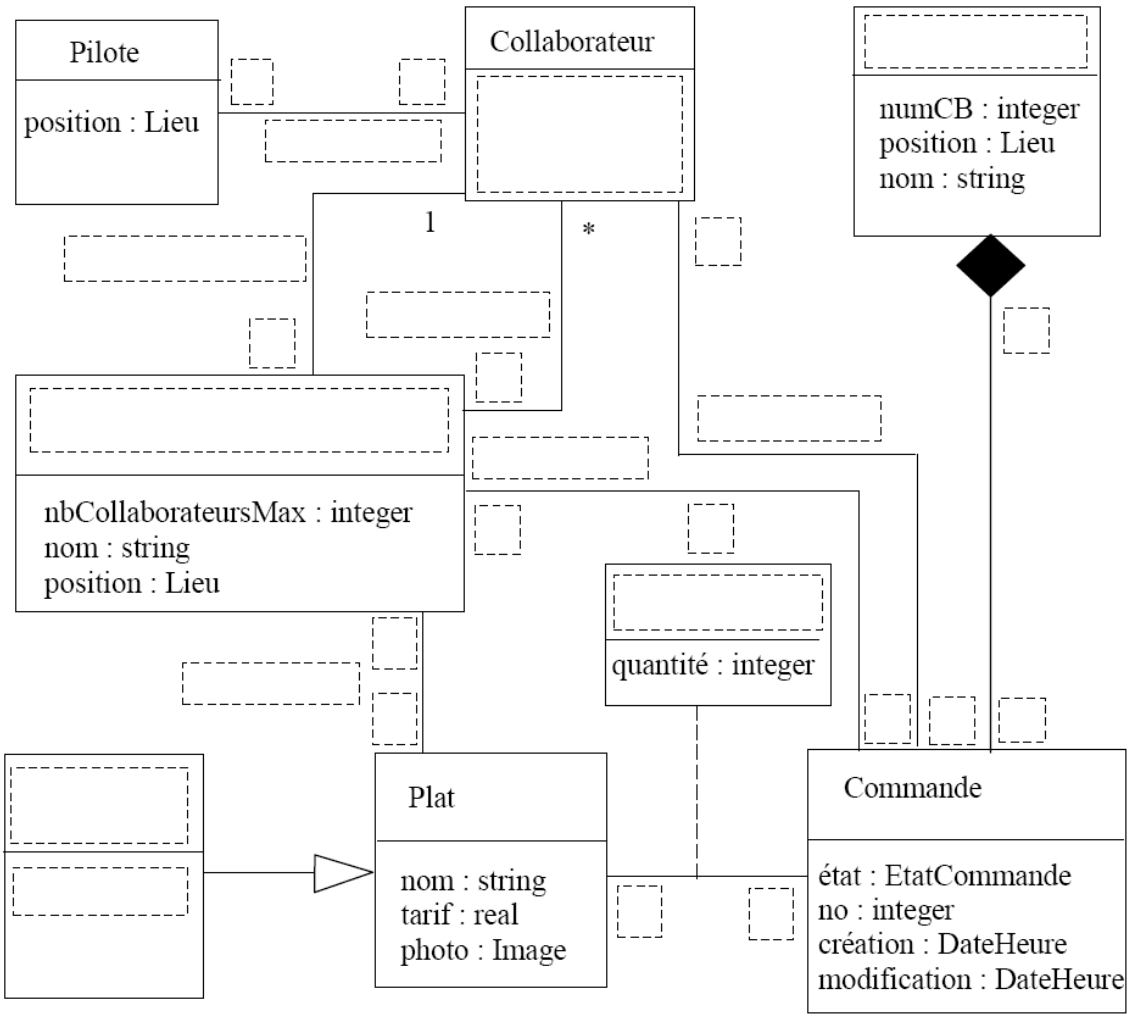
Chaque livreur utilise son propre moyen de transport (bus, vélo, roller, voiture ...). Par contre, un appareil appelé "Pilote" lui est remis lors de son affectation à la livraison. Chaque pilote intègre un GPS permettant de localiser le livreur de manière précise via une liaison satellite. Un écran permet au livreur de consulter les commandes qui lui ont été affectées. Il peut à tout moment consulter la carte et se situer par rapport aux points Kebab et aux clients à livrer. Le livreur utilise également le pilote pour indiquer quand il récupère une commande auprès du Point Kebab et quand il livre la commande au client.

Dans chaque Point Kebab un collaborateur joue le rôle de "coordinateur". C'est le seul du restaurant à agir directement avec CyberKebab : les autres collaborateurs préparent les plats. Le coordinateur consulte les commandes à réaliser et indique pour chaque commande quand sa préparation débute, quand elle se termine et quand elle est remise au livreur.

a) Compléter le diagramme des cas d'utilisation du système CyberKebab. Seuls les acteurs humains sont pris en compte (ni le Pilote, ni le téléphone ne sont représentés).



b) Compléter le diagramme de classes sans ajouter ni classes, ni associations mais en complétant les zones en pointillés. Les zones de petite taille correspondent à des cardinalités.



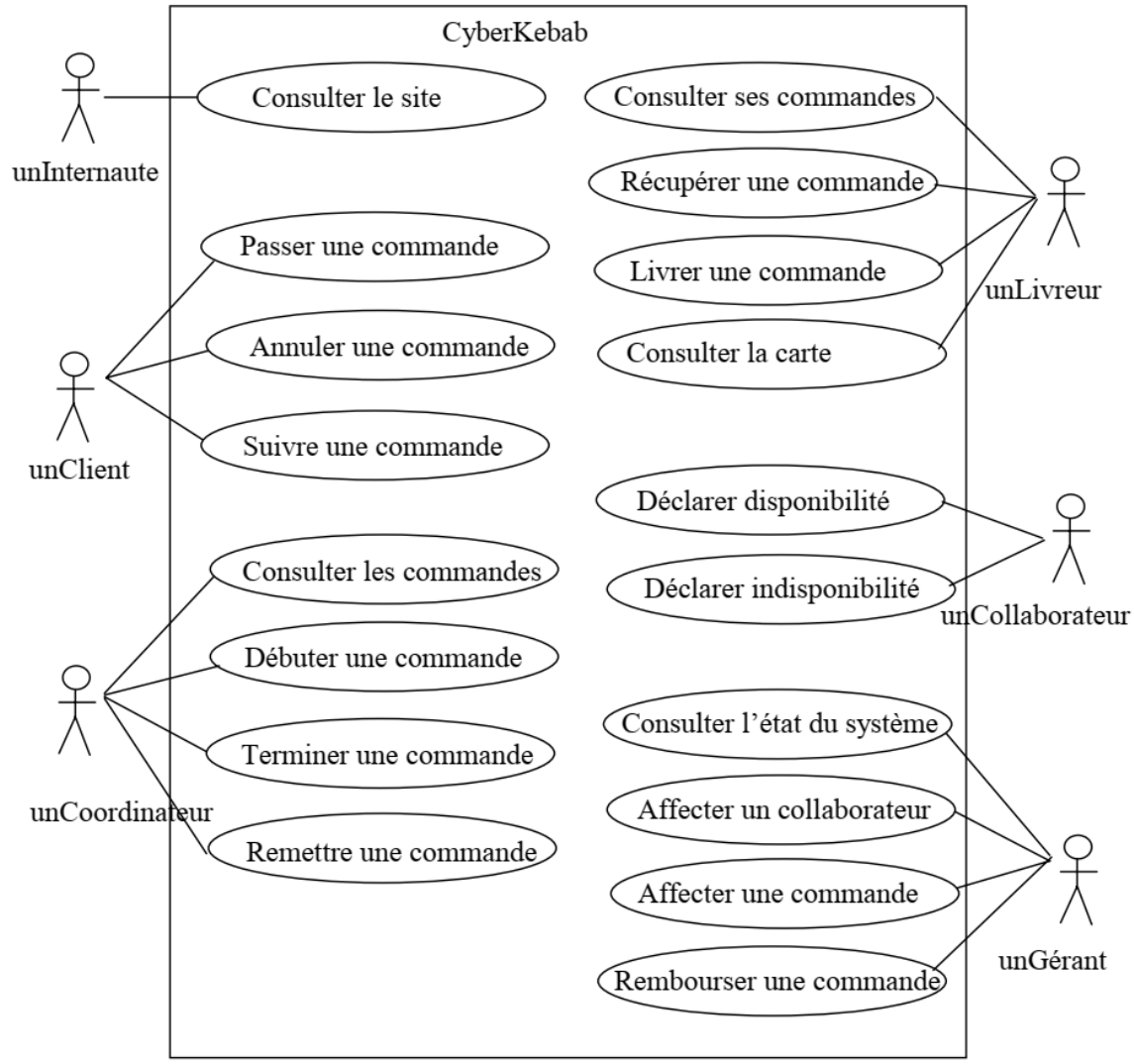
Lieu est un type permettant de situer dans l'espace.

DateHeure est un type permettant de situer dans le temps.

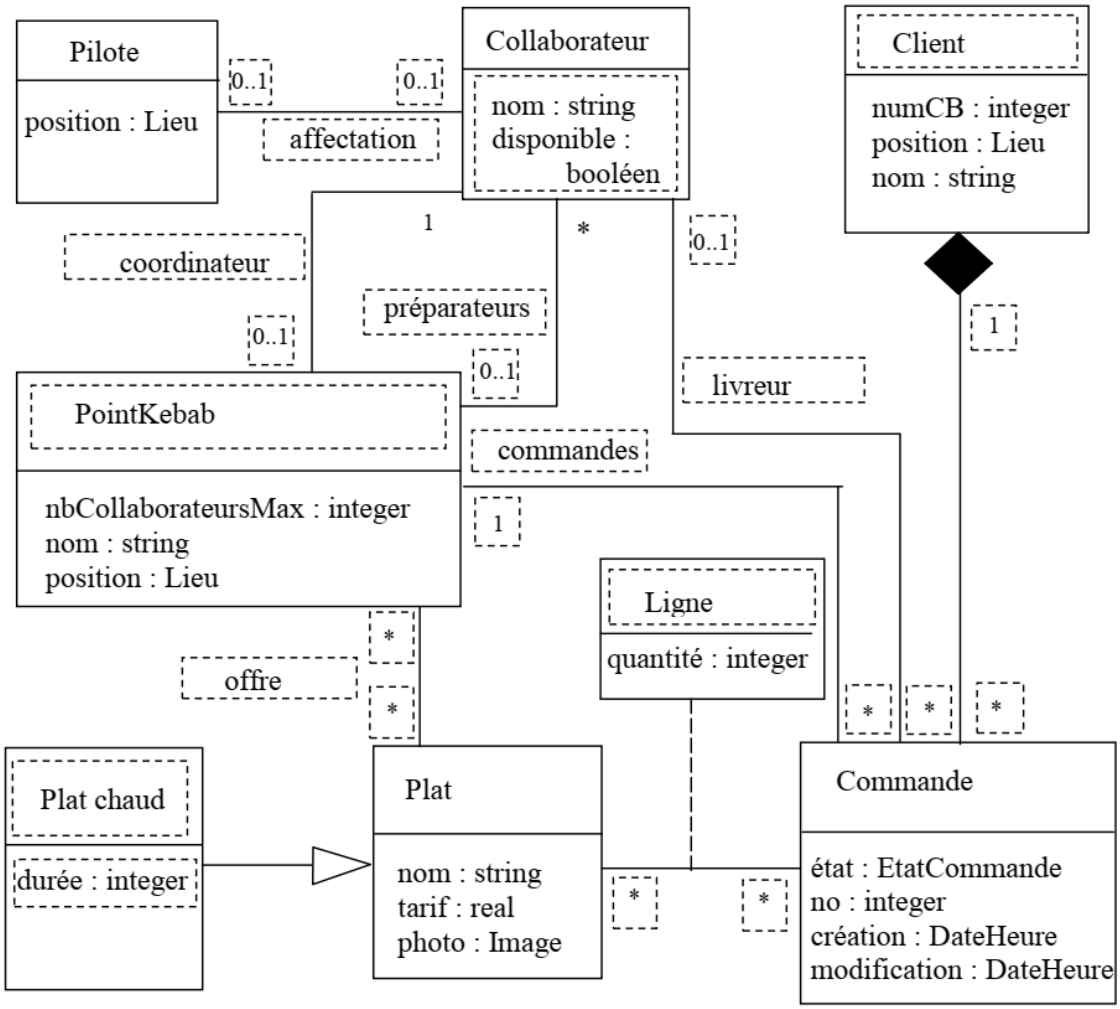
EtatCommande est un type énuméré prenant les valeurs suivantes :

12. Corrigé de l'étude de cas

a)



b)



Lieu est un type permettant de situer dans l'espace.

DateHeure est un type permettant de situer dans le temps.

EtatCommande est un type énuméré prenant les valeurs suivantes :

enAttenteAffectation, enAttentePréparation, enPréparation, enAttenteLivraison, enCoursLivraison, livrée, remboursée, annulée

Conclusion générale

À travers ce support de cours, on a présenté l'ensemble de connaissances de base qu'un étudiant informaticien doit savoir sur le domaine de génie logiciel (GL). Ce document permet à son lecteur de comprendre tout d'abord les motivations et les raisons de l'apparition et de l'adoption du GL dans le monde de l'informatique. Cette adoption est due à la crise logicielle caractérisant la fin des années 60. Le GL a permis de changer la vision des informaticiens vis-à-vis de la production de logiciels. Actuellement, le GL permet de produire des logiciels de qualité avec maîtrise des coûts et délais.

La conception est un processus créatif qui consiste à représenter les diverses fonctions du système permettant d'obtenir rapidement un ou plusieurs programmes réalisant ses fonctions. Une bonne conception se définit en termes de la satisfaction des besoins et des spécifications. Elle participe largement à la production d'un logiciel qui répond aux facteurs de qualité. La technologie orientée objet guide la conception par un ensemble de concepts. Elle reflète plus finement les objets du monde réel. Le langage UML a une relation directe avec l'analyse et la conception orientée objet.

L'UML est traité dans le troisième chapitre de ce document avec suffisamment de détails permettant au lecteur d'apprendre ses principes et de découvrir quelques-uns de ses diagrammes les plus importants dans les phases d'analyse et de conception.

Ce document peut être utilisé par les étudiants informaticiens novices, les étudiants de fin de cycle préparant un projet logiciel et aussi les enseignants voulant découvrir ou enseigner le module GL.

Bibliographie :

[1] D. Gustafson, Génie Logiciel, Dunod, Paris, 2003

[2] M. Lemoine, Précis de génie logiciel, Masson, Paris, 1996

[3] P. Roques, UML 2 par la pratique - Etudes de cas et exercices corrigés, éditions Eyrolles, 2006.

[4] J. Gabay, D.Gabay , UML 2 Analyse et conception, Mise en œuvre guidée avec études de cas, Dunod, 2008.

[5] B. Charroux, A. Osmani, Y. Thierry-Mieg, UML 2, pratique de la modélisation, éditions synthex, 2009.

[6] N. Abdat, L. Mahdaoui, UML Outil du génie logiciel, pages blues, 2007.

[7] P-A. Muller, N.Gaertner, Modélisation objet avec UML, éditions Eyrolles, 2003.

[8] L. Audibert, Cours UML 2.0, site http://www.developpez.com.