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 : 3*ème* année licence

Etabli par Dr. Mohamed MOHAMMEDI

2020/2021

Préface

Le *génie logiciel* a été développé en réponse aux problèmes de construction de grands systèmes logiciels personnalisés pour les applications de défense, gouvernementales et industrielles. Il existe deux approches de conception pour l'étude du g*énie logiciel* : l'approche traditionnelle orientée fonction (approche fonctionnelle) et l'approche contemporaine Orientée Objet. De nombreux ouvrages sur le *génie logiciel* suivent l'une ou l'autre des deux approches, alors que les apprenants doivent connaître les deux approches et être clairs sur les similitudes et les différences entre les deux. Bien que, l'approche fonctionnelle n'est pas adaptée au développement d'applications qui évoluent sans cesse et dont la complexité croit continuellement, ce support de cours couvre entièrement la méthodologie orientée objet des systèmes d'information, ce qui en fait un cours plus intéressent sur le sujet. L'objectif de l'enseignement de *génie logiciel* est d'appliquer une méthodologie d'analyse et de conception pour le développement des logiciels. En particulier, apprendre la modélisation objet avec le langage universel UML.

Ce support de cours s'adresse bien aux étudiants de 3ème année licence en Informatique. Ce cours comporte huit chapitres principaux, où sont exposées des notions sur la naissance et l'importance de *génie logiciel*, le contexte objet dans lequel est né UML, les principaux diagrammes d'UML, mécanismes d'extension : langage OCL + les profils, et une introduction aux méthodes de développement : (RUP, XP). Enfin, chaque chapitre s'achève par quelques exercices corrigés permettant de contrôler l'acquisition des concepts essentiels introduits.

Table des matières

Chapitre 7 : Introduction aux méthodes de développement : (RUP, XP)

Chapitre 8 : Patrons de conception et leur place au sein du processus de développement

Table des figures

Liste des tableaux

Liste des abréviations

com : communication diagram

COO : Conception Orientée Objet

FSK : Freiwillige Selbstkontrolle der Filmwirtschaft

GAB : Guichet Automatique Bancaire

IBM : International Business Machines Corporation

MOA : Maîtrise d'ouvrage

MOE : Maîtrise d'œuvre

OCL : Object Constraint Language

OMG : Object Management Group

OMT : Object Modeling Technique

OOSE : Object Oriented Software Engineering

RUP : Rational Unified Process

sd : sequence diagram

UML : Unified Modeling Language

UP : Unified Process

XP : eXtreme Programming

Introduction générale

Le *Génie Logiciel*, inventé en 1968, est une science de génie industriel qui étudie les méthodes de travail et les bonnes pratiques pour le développement des produits logiciels. Le génie logiciel comprend l'exigence et l'application des principes d'ingénierie dans la production des logiciels. En d'autres termes, il s'intéresse particulièrement aux instructions méthodiques permettant de réaliser que des logiciels complexes répondant aux attentes du client. Ainsi, les logiciel développés doivent être fiables et de bonnes performances produits dans un délai raisonnable et avec les coûts de production et de maintenance les plus bas possibles.

L'objectif de ce cours est d'offrir aux apprenants le savoir nécessaire pour comprendre les fondements de conception de base des systèmes informatiques reposant sur une compréhension technique approfondie de la reproduction des produits logiciels du qualité, produit dans des délais raisonnables et avec des coûts les plus réduits possible. Dans un premier temps, ce cours permet d'initier les apprenants, à la conception des applications informatiques de façon méthodique et reproductible ; en les incitant à rechercher et établir les fonctionnalités d'une application, et à les modéliser sous forme de cas d'utilisation et de scénarios ainsi que rechercher les classes et les acteurs nécessaires à la conception de l'application. Dans un second temps, d'une façon spécifique, à l'issue de ce cours, l'apprenant sera capable de : (i) acquérir les bonnes pratiques de conception, (ii) maîtriser des techniques de génie logiciel, en se focalisant sur les approches par objets et par composants ; (iii) présenter un aperçu sur les principaux courants de pensées en matière de développement logiciel, et (iv) exposer un ensemble de pratiques pragmatiques qui permettent de survivre à un projet logiciel tout au long son cycle de développement.

Ce support de cours s'articule autour de huit chapitres indépendants. Le contenu de ce cours peut-être synthétisé dans ce qui suit :
Le premier chapitre s'intitule « **Introduction au Génie Logiciel** », il a pour objectifs de dévoiler les raisons de l'apparition du domaine de Génie Logiciel. En effet, nous décrivons l'ensemble des activités qui mènent d'un besoin à la livraison du produit logiciel. De plus, ce chapitre couvre les principaux cycles de vie d'un logiciel.

Le deuxième chapitre intitulé « **Modélisation avec UML (Unified Modeling Language)** ». Dans ce chapitre, pour mieux comprendre le langage UML, une définition de la modélisation, modèle et de la modélisation orientée objet est d'abord donnée. Ensuite, le langage UML est présenté dans un petit détail avec une brève explication détaillée des mécanismes généraux qui assurent l'intégrité conceptuelle de la notation. Le but final de ce chapitre est de fournir au lecteur les prérequis nécessaires pour pouvoir comprendre les chapitres suivants.

Le troisième chapitre nommé « **Diagramme UML de cas d'utilisation : vue fonctionnelle** ». Dans ce chapitre l'apprenant va découvrir le diagramme UML utilisé pour une représentation du comportement fonctionnel d'un système logiciel.

Le quatrième chapitre intitulé « **Diagrammes UML de classes et d'objets : vue statique** ». Ce chapitre est consacré aux diagrammes de classes qui permettent de représenter les structures des classes composant le système et les relations entre les classes. Le diagramme d'objets qui permet de montrer un contexte avant ou après une interaction ou pour faciliter le choix des multiplicités, pour faciliter la compréhension de structures de données complexes, est également présenté dans ce chapitre.

Le cinquième chapitre intitulé « **Diagrammes UML : vue dynamique** ». Ce chapitre porte sur les diagrammes d'interaction permettant d'établir un lien entre les diagrammes de cas d'utilisation et les diagrammes de classes, tout en montrant comment des objets communiquent pour réaliser une certaine fonctionnalité.

Le seizième chapitre intitulé « **Autres notions et diagrammes UML (Unified Modeling Language)** ». Ce chapitre traite d'une part d'autres diagrammes UML : diagramme de composants, diagramme de déploiement et diagramme de structure composite. D'autre part, il traite d'autres notions de mécanismes d'extension : langage OCL + profils.

Le septième chapitre intitulé « **Introduction aux méthodes de développement : (RUP, XP)** ». Ce chapitre est une introduction à trois méthodes de développement d'un logiciel à savoir la méthode UP (Unified Process), la méthode RUP (Rational Unified Process), et la méthode XP (eXtreme Programming).

Le huitième chapitre intitulé « **Patrons de conception et leur place au sein du processus de développement** ». Ce chapitre introduit les patrons de conception qui sont l'une des techniques de conception hypermédia. Cette technique de génie logiciel permet une définition ou une réutilisation de patrons de conception qui offrent des caractéristiques déterminantes à savoir : la réutilisabilité, l'uniformité, la facilité d'utilisation et la fiabilité.

Enfin, chaque chapitre s'achève par une série d'exercices d'application et de réflexion pour observer la compréhension et tester les connaissances acquises par les apprenants durant chaque chapitre.

Chapitre 1 : Introduction au Génie Logiciel

Objectif

À l'issue de ce chapitre, l'apprenant sera capable de :

- Connaître la définition et les objectifs du domaine du génie logiciel ;
- Connaître les critères qu'un logiciel doit remplir pour pouvoir dire qu'il est de qualité ;
- Comprendre ce qu'est un cycle de vie logiciel ;
- Connaître les différents modèles de cycle de vie d'un logiciel.

1. Introduction

A la fin des années 60, éclate la « crise du logiciel », dénotant la situation générale qui caractérise les difficultés rencontrées dans le développement de logiciels. Dans son essence, le phénomène renvoie à la difficulté de produire des logiciels de qualité à bon prix et dans des délais raisonnables. Les diverses causes à l'origine de cette crise sont les suivants :

- La qualité du produit logiciel livré était imparfaite et ne répondait pas aux besoins des utilisateurs. De plus, le logiciel produit consommait plus de ressources que prévu et était à l'origine de pannes.

- Les performances du logiciel telles que le temps de réponse, la disponibilité, le coût de stockage, etc., étaient très souvent médiocres.

- Les délais fixés pour la production des produits logiciels sont généralement dépassés et ne répondent pas aux spécifications ;

- La prédiction des coûts de développement de logiciels était presque infaisable et habituellement excessif.

- L'invisibilité du logiciel, ce qui signifie qu'il a souvent été constaté que le logiciel développé ne répondait pas à la demande au moment de la livraison. En d'autres termes, il n'y avait pas de transparence pendant le développement.

- La maintenance du produit logiciel était difficile, coûteuse et souvent source de nouvelles erreurs.

2. Définition

Le génie logiciel est une étude détaillée de l'ingénierie pour la conception, le développement et la maintenance de logiciels. Le domaine du génie logiciel a été introduit pour résoudre les problèmes de projets logiciels de mauvaise qualité. Et, les problèmes qui surviennent lorsque le logiciel dépasse généralement les délais, les budgets et les niveaux de qualité réduits.

Par conséquent, le génie logiciel fait référence à toutes les méthodes, techniques et outils contribuant à la production de logiciels de qualité avec contrôle des coûts et des délais.

3. Objectifs

Le génie logiciel est une discipline qui étudie l'utilisation des méthodes d'ingénierie pour créer et maintenir des logiciels efficaces, pratiques et de haute qualité. Le génie logiciel est une ingénierie qui applique les principes de l'informatique, des mathématiques et des sciences de gestion pour développer des logiciels. Par conséquent, le génie logiciel est une discipline qui utilise des connaissances scientifiques et des principes techniques pour définir, développer et maintenir des logiciels. Il applique des concepts, principes, techniques et méthodes d'ingénierie, ainsi que des techniques de développement scientifique et des méthodes de gestion pour développer des logiciels. Le génie logiciel étudie comment appliquer les théories scientifiques et les techniques d'ingénierie pour guider le développement et la maintenance des logiciels informatiques. Son objectif fondamental est de développer un ensemble de méthodes d'ingénierie scientifique, de concevoir un ensemble de systèmes d'outils pratiques et d'améliorer l'efficacité et la qualité du développement de logiciels.

4. Qualités exigées d'un logiciel

La qualité du logiciel est définie par son aptitude à satisfaire les besoins des utilisateurs. Un logiciel est dit de qualité s'il remplit au moins les critères suivants :

- **Robustesse *:*** Accomplir sans défaillance l'ensemble des fonctionnalités spécifiées, dans un environnement opérationnel de référence et pour une durée d'utilisation donnée.

- **Maintenabilité *:*** C'est la facilité avec laquelle la maintenance d'un produit logiciel peut être effectuée.

- **Efficacité *:*** est la capacité d'un système logiciel à utiliser le minimum de ressources matérielles, que ce soit le temps machine, l'espace occupé en mémoire externe et interne, ou la bande passante des moyens de communication, etc.

- **Facilité d'utilisation *:*** Est la facilité avec laquelle des personnes présentant des formations et des compétences différentes peuvent apprendre à utiliser les produits logiciels et s'en servir pour résoudre des problèmes. Elle recouvre également la facilité d'installation, d'opération et de contrôle.

5. 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 d'un logiciel 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.

6. Étapes de développement d'un logiciel

Un produit logiciel ou un système logiciel doit passer par des étapes de gestation, de naissance, de croissance, de maturité et de déclin, que l'on appelle généralement le cycle de vie du logiciel.

L'ensemble du cycle de vie du logiciel est divisé en plusieurs étapes, de sorte que chaque étape a une tâche claire, de sorte que le développement de logiciels à grande échelle, une structure complexe et une gestion complexe devient plus facile à contrôler et à gérer. Quel que soit le modèle de cycle de vie choisi pour concevoir et réaliser un logiciel, il existe un certain nombre d'étapes de base à suivre :

a. **Faisabilité (Phase de planification et d'étude de faisabilité du logiciel**)

Cette étape est une discussion conjointe entre le développeur du logiciel et le demandeur, principalement pour déterminer les objectifs de développement et la faisabilité du logiciel. Les questions qui devraient se poser lors de la phase de planification et d'étude de faisabilité du logiciel sont les suivantes :

- Pourquoi développer le logiciel ?
- Y a-t-il de meilleures alternatives ?
- Comment procéder pour faire ce développement ?
- Y a-t-il un marché pour le logiciel ?
- Quels moyens faut-il mettre en œuvre ? A-t-on le budget, le personnel, la matériel nécessaires ?

La description de l'activité de faisabilité des besoins (quel est le problème à traiter), ces entrées et ces sorties sont résumés dans la table suivante :

Activité	Etude préalable
Description	Etudier la faisabilité du projet, ses contraintes techniques (coût, temps, qualité) et les alternatives possibles.
Entrées	Problème à résoudre, objectifs à atteindre.
Sorties	**Oui** (la décision est prise de réaliser le logiciel) ou **Non** (le projet est abandonné).

b. Analyse des besoins

Après avoir confirmé que le développement du logiciel est faisable, nous effectuons une analyse détaillée des différentes fonctions que le logiciel doit implémenter. L'étape d'analyse des besoins, qui appelée aussi spécification des besoins, est une étape très importante, et c'est aussi une étape en constante évolution et en profondeur dans l'ensemble du processus de développement logiciel, qui peut jeter une bonne base pour la réussite de l'ensemble du projet de développement logiciel. Cette étape doit contenir la mise du logiciel dans son contexte (type de produit, nouveau/altéré) et l'étude de l'existant. L'étude de l'existant désigne l'étude des produits similaires dans le marché (veille concurrentielle) et l'étude du processus ou des logiciels similaires à l'entreprise. Les besoins de l'entreprise peuvent contenir des besoins fonctionnels (des fonctionnalités que le produit doit automatiser) et des besoins non fonctionnels (disponibilité, rapidité de calcul, etc.).

La description de l'activité de spécification des besoins (quel est le problème à traiter), ces entrées et ces sorties sont résumés dans la table suivante.

Activité	Spécifier
Description	Décrire ce que doit faire le logiciel (comportement en boite-noire). Décrire comment vérifier en boite-noire que le logiciel fait bien ce qui est exigé.
Entrées	Client qui à une idée de ce qu'il veut. Exigences, désir, besoins… concernant le système permettant de résoudre le problème.
Sorties	Cahier des charges du logiciel (ou spécification du logiciel). Des procédures de validation. Version provisoire des manuels d'utilisation et d'exploitation du logiciel.

c. Conception (architecturale et détaillée)

La conception utilise les spécifications pour décider des solutions proposées. Elle peut contenir la description des fonctionnalités de l'application. Il s'agit des fonctionnalités précisées lors de la spécification des besoins. La conception peut contenir la conception des interfaces, la conception des données et la conception de l'architecture matérielle.

La description de l'activité de conception (quel est le problème à traiter), ces entrées et ces sorties sont résumés dans la table suivant :

Activité	Concevoir
Description	Organiser le logiciel afin qu'il puisse satisfaire les exigences de la spécification. Faire les principaux choix techniques pour satisfaire les exigences de la spécification.
Entrées	Une spécification.
Sorties	Une description des décisions de conception. Des procédures de tests qui permettent de vérifier que les décisions de conception sont correctement implémentées en code source et qu'elles contribuent à satisfaire les exigences de la spécification.

d. Codage

Il s'agit de convertir le résultat de la conception du logiciel en un code opérationnel pouvant être exécuté par ordinateur. Afin d'assurer la lisibilité du programme, faciliter la maintenance et améliorer l'efficacité opérationnelle du programme, les techniques de codage qui dépendent du langage doivent être bien conformes à la conception.

La description de l'activité de codage (quel est le problème à traiter), ces entrées et ces sorties sont résumés dans la table suivante :

Activité	Coder et tester
Description	Ecrire le code source du logiciel. Tester le comportement du code source afin de vérifier qu'il réalise les responsabilités qui lui sont allouées.
Entrées	Spécification, conception.
Sorties	Code source. Tests unitaire. Documentation.

e. Intégration

Cette étape consiste à assembler tout ou partie des composants d'un logiciel pour obtenir un système exécutable. Elle utilise la gestion de configuration pour assembler des versions cohérentes de chaque composant.

La description de l'activité d'intégration (quel est le problème à traiter), ces entrées et ces sorties sont résumés dans la table suivante :

Activité	Intégrer
Description	Assembler le code source du logiciel (éventuellement partiellement) et dérouler les tests d'intégration.
Entrées	Conception, code source, tests d'intégration (tests).
Sorties	Un rapport de tests d'intégration.

f. Validation

Une fois la conception du logiciel terminée, il doit subir des tests rigoureux pour trouver et corriger les problèmes du logiciel tout au long du processus de conception. En d'autres termes, les tests permettent de déterminer les bugs techniques, les bugs fonctionnels et la qualité du logiciel, en utilisant des logiciels de test, des techniques et des benchmarks.

La description de l'activité d'intégration (quel est le problème à traiter), ces entrées et ces sorties sont résumés dans la table suivante :

Activité	Valider
Description	Construire le logiciel complet exécutable. Dérouler les tests de validation sur le logiciel complet exécutable.
Entrées	Logiciel complet exécutable à valider. Tests de validation.
Sorties	Rapport de tests de validation.

g. Maintenance

Cette activité permet d'ajuster l'application après la livraison du produit au client. Elle a pour but de corriger les erreurs et les anomalies du système et modifier le système pour y ajouter des fonctionnalités.

Activités :

a. Maintenance corrective : correction des bugs.
b. Maintenance adaptative : ajuster le logiciel.
c. Maintenance perfective, d'extension : augmenter / améliorer les possibilités du logiciel.

Productions :

a. Logiciel corrigé ;
b. Mises à jour ;
c. Documents corrigés.

7. Modèles de cycle de vie d'un logiciel

Il existe différents types de cycles de développement entrant dans la réalisation d'un logiciel. Ces cycles prennent en compte toutes les étapes de la conception d'un logiciel. Parmi les modèles de cycles de vie on peut citer :

7.1 Modèle du cycle de vie en Cascade

Le modèle de cycle de vie en Cascade est créé par Winston W. Royce en 1970. Le principe de ce modèle est d'avoir un certain nombre d'étapes chacune se terminant à une date précise par la production de certains documents ou logiciels, on ne passe à l'étape suivante que si l'étape en cours est terminée. Si une erreur critique est rencontrée, il est possible de revenir à la première étape.

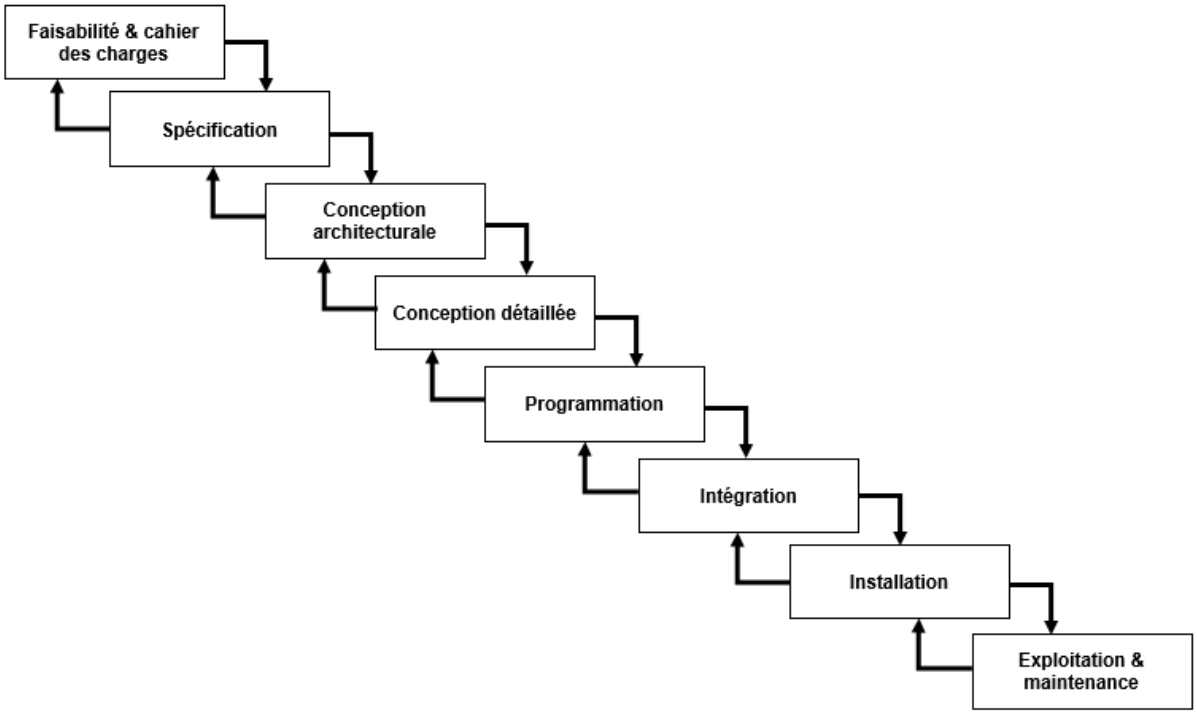


Figure 1.1 : Modèle du cycle de vie en cascade.

Remarque

Le modèle original de la cascade ne comportait pas de possibilité de retour en arrière. Les versions actuelles du modèle en cascade font apparaître la validation-vérification à chaque étape.

En pratique, plusieurs versions du modèle en cascade sont utilisées. Les versions actuelles de ce modèle montrent la validation-vérification à chaque activité. En conséquence, nous trouvons 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 la programmation, du test unitaire ;
- avec l'intégration, du test d'intégration, puis du test d'acceptation ;

- avec l'installation, du test système.

Avantages

- Une structure simple grâce à des phases de projet clairement délimitées.
- Une bonne documentation du processus de développement produite par des étapes clairement définies.
- Le planning est établi à l'avance dès le début du projet et le chef de projet sait précisément ce qui lui sera livré et quand il pourra en prendre livraison.

Inconvénients

- Il y en a pas mal, mais le principal inconvénient est la très faible tolérance à l'erreur (les anomalies sont détectées tardivement) qui entraîne automatiquement un coût élevé en cas d'anomalie.
- Difficulté d'avoir toutes les spécifications du client.
- Pas transparent au client lors du développement.

Remarque

Modèle du cycle de vie en cascade est mieux adapté aux petits projets ou à ceux dont les spécifications sont bien connues et fixes.

7.2 Modèle du cycle de vie en V

Le modèle de cycle de vie en V est un modèle utilisé dans différents processus de développement, particulièrement dans le développement de logiciels. Ce modèle a été développé en 1990 dans sa forme originale, il se perfectionne au fil des années et s'adapte aux méthodes de développement actuelles. L'idée originale, cependant, remonte aux années 1970 et a été envisagée comme une sorte d'extension du modèle de cycle de vie de la cascade. La particularité de ce modèle par rapport au modèle en cascade est qu'il existe un accès avancé aux tests. Outre les différentes phases de développement d'un projet, le modèle en V définit également les procédures associées à mettre en place en termes d'assurance qualité et détaille comment les différentes phases doivent interagir entre elles. En d'autres termes, la caractéristique principale de ce modèle est que chaque étape de conception fonctionne en tandem avec une phase de test (validation). Ensuite, plus on avance dans l'étude, plus le niveau de détails est important, par ailleurs à partir de la « *Programmation* », plus on avance dans les tests se déroulent dans la globalité du projet (moins de détails). Le cycle en V est le cycle qui a été normalisé, il est largement utilisé.

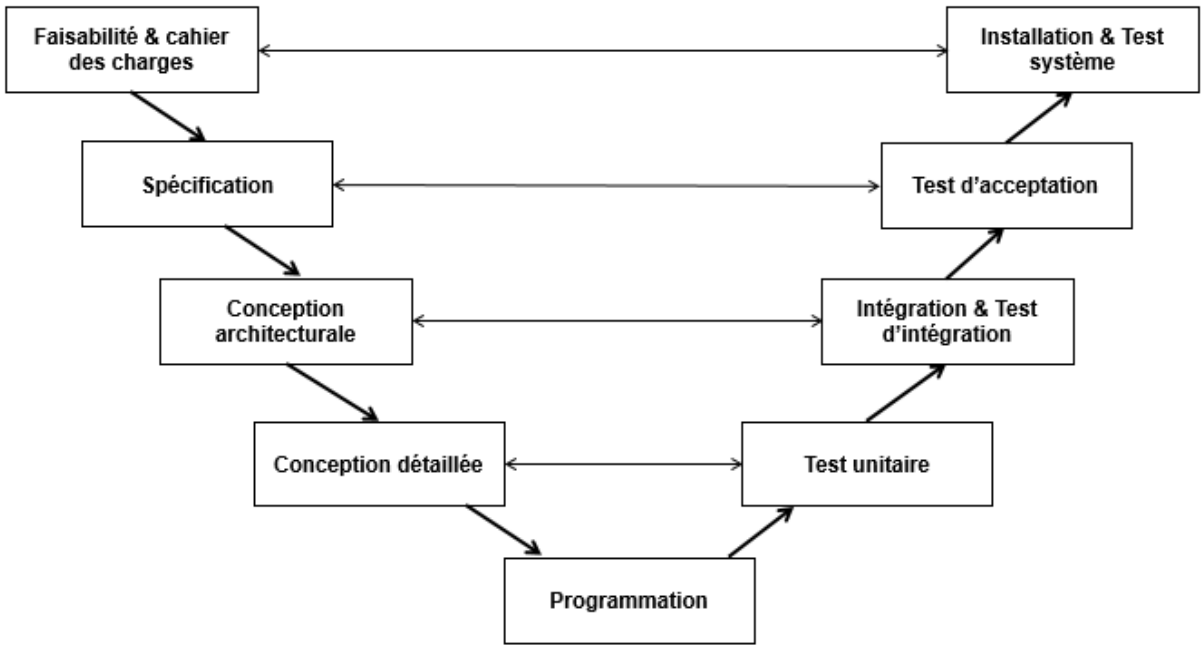


Figure 1.2 : Modèle du cycle de vie en V.

Avantages

- Contient des tests de bas niveau (tests unitaires) et des tests de haut niveau (tests de système) ;
- Identifier clairement les différentes étapes de développement et de test ;
- Affinement progressif de haut en bas, avec une division claire du travail à chaque étape, ce qui facilite le contrôle global du projet.
- Les éventuelles erreurs peuvent être détectées plus tôt.

Inconvénients

- La séquence descendante permet de tester le travail qui ne peut pas être modifié à temps après le codage ;
- Dans le travail réel, les exigences changent souvent, ce qui entraîne une exécution répétée des étapes du modèle V, une grande quantité de retouches et une faible flexibilité.

Remarque

Modèle du cycle de vie en V est adapté aux projets de taille et de complexité moyenne.

7.3 Modèle du cycle de vie en spirale

Le modèle de cycle de vie en spirale a été proposé par Barry BOEHM en 1988, à l'époque il était considéré comme un modèle de processus évolué par rapport aux modèles classiques tels que : le modèle en cascade et le modèle en V. De plus, ce modèle est plus complet que les autres et peut les inclure. Il met fortement l'accent sur l'activité d'analyse des risques : chaque cycle de la spirale se déroule en quatre phases qui se répètent de manière itérative :

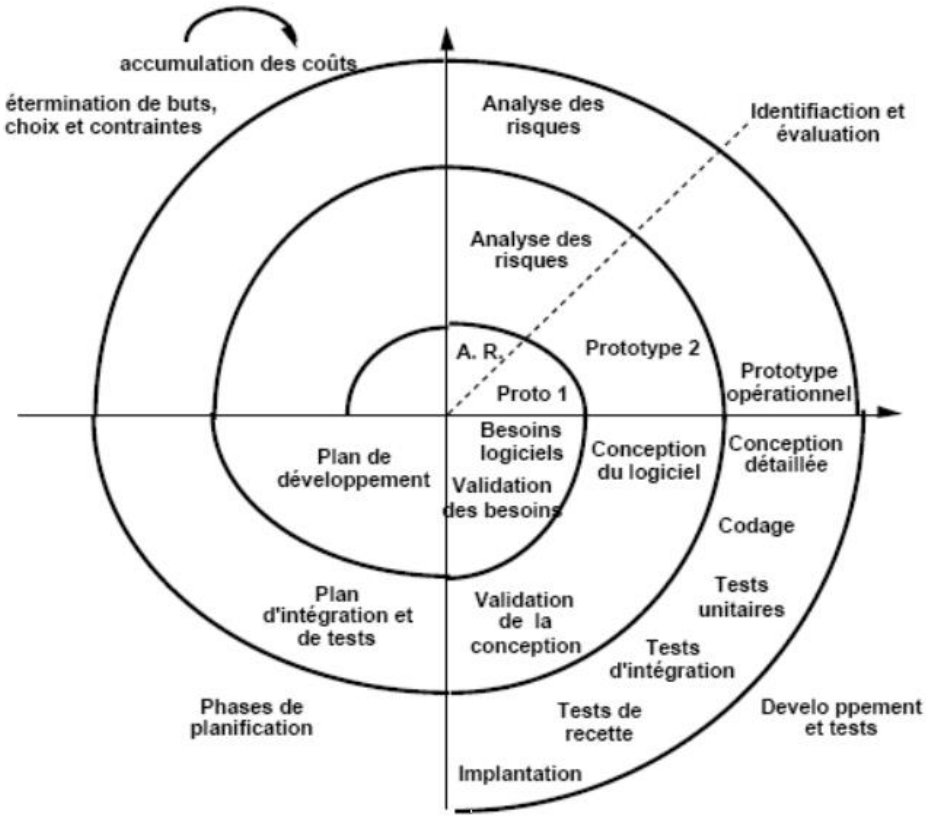


Figure 1.3 : Modèle du cycle de vie en spirale.

- **Phase 1 :** Détermination, à partir des résultats des cycles précédents, ou de l'analyse préliminaire des besoins, des objectifs du cycle, des alternatives pour les atteindre et des contraintes ;
- **Phase 2 :** Analyse des risques, évaluation des alternatives, éventuellement maquettage (prototypage).
- **Phase 3 :** Développement et vérification de la solution retenue, un modèle « classique » (cascade ou en V) peut être utilisé ici ;
- **Phase 4 :** Revue des résultats et planification du cycle suivant.

L'analyse préliminaire est affinée au cours des premiers cycles. Le modèle utilise des maquettes exploratoires pour guider la phase de conception du cycle suivant. Le dernier cycle se termine par un processus de développement classique.

Avantages

- Flexibilité de conception, qui peut être modifiée à toutes les étapes du projet.
- Construction d'un système à grande échelle avec de petits segments pour rendre le calcul des coûts simple et facile.
- Le client participe toujours à chaque étape de développement, en s'assurant que le projet ne s'écarte pas de la bonne direction et de la contrôlabilité du projet.

- Au fur et à mesure de l'avancement du projet, le client dispose toujours des dernières informations sur le projet afin de pouvoir interagir efficacement avec la direction.
- Le client reconnaît la bonne communication et les produits de haute qualité apportés par la méthode de développement interne de l'entreprise.

Inconvénients

Le modèle en spirale n'est pas absolument excellent par rapport aux autres modèles, en fait, ce modèle a aussi ses propres défauts, comme suit :

- L'utilisation du modèle en spirale nécessite une expérience et une expertise considérables en matière d'évaluation des risques : dans le développement de projets risqués, si le risque n'est pas identifié à temps, il entraînera inévitablement des pertes importantes.
- Trop d'itérations augmenteront les coûts de développement et retarderont le temps de soumission.

Remarque

Modèle du cycle de vie en spirale est utilisé pour des projets innovants, à risques, et dont les enjeux sont importants.

8. Conclusion

Il est efficace d'organiser le travail de développement logiciel selon des principes et des méthodes d'ingénierie, et c'est aussi un moyen essentiel de se débarrasser de la crise logicielle. L'idée principale du ce domaine est de souligner l'importance de l'application des principes d'ingénierie dans le processus de développement logiciel. Le chapitre suivant portera sur la modélisation avec le langage UML.

9. Exercices

Exercice 1 :

1. Dresser la liste des principaux acteurs impliqués dans le développement de logiciels et expliquez brièvement le rôle de chacun.
2. Expliquer la différence entre les tests unitaires et les tests d'intégration ?
3. Expliquer brièvement la différence entre les tests de validation et de vérification ?
4. Expliquer pourquoi la maintenance du logiciel est nécessaire ?
5. Quelles sont les qualités requises dans un cahier de charges ?
6. Expliquer pourquoi le modèle de cascade ne reflète pas exactement les activités de développement du logiciel.
7. Expliquer pourquoi il est utile de faire la distinction entre définition des besoins et spécification des besoins.
8. Comment peut-on réduire l'écart entre les besoins réels d'un client et les besoins exprimés dans le cahier de charges ?
9. Quelles sont les phases impliquées dans le modèle de cycle de vie en V ?

Exercice 2 :

Une université voudrait s'équiper d'un système intégré de gestion des étudiants et qui prendrait en compte tous les détails concernant les étudiants y compris les informations personnelles, les cours suivis et les notes obtenues aux examens.
Les trois approches possibles sont :

1. Acheter un système de gestion de bases de données et développer son propre système basé sur cet outil.
2. Acheter un système comparable à celui d'une autre université et le modifier pour ses propres besoins.
3. Se joindre à un groupe d'autres universités, établir un cahier des charges commun, contacter une société de logiciels qui développera un seul système pour tous.

Travail demandé :

1. Identifier deux risques possibles pour chacune de ces stratégies et proposez des techniques de résolution de risque qui permettraient de décider quelle approche adopter ?
2. Présenter la liste de fonctionnalités que pourrait contenir cette application.

10. Corrigés des exercices

Corrigé de l'exercice 1

1. Les principaux acteurs intervenants à la réalisation d'un projet logiciel sont comme suit :

- ***L'utilisateur final*** exprime et définit précisément ses besoins d'information, puis met en œuvre les applications. Le rôle de l'utilisateur final tout au long d'un projet est crucial pour le succès de ce dernier. En effet, tout projet vise à atteindre les objectifs fixés par l'utilisateur final et à obtenir sa satisfaction. De plus, la première condition de réussite d'un projet réside dans la compréhension de ce qui est attendu par la maîtrise d'ouvrage.

- ***La maîtrise d'ouvrage (MOA), propriétaire*** aussi dénommée ***maître d'ouvrage*** est la personne pour qui est réalisé le projet. Il est le commanditaire du projet qui définit le cahier des charges et par conséquent les besoins, le budget consacré à ce projet, le calendrier prévisionnel ainsi que les objectifs à atteindre.

- Le ***Chef de Projet Maîtrise d'Œuvre (MOE)*** ou ***développeur*** a pour rôle d'analyser, de concevoir et de réaliser des logiciels (applications informatiques) répondant à des besoins spécifiques. En conséquence, il assure le bon déroulement du projet du début à la fin en faisant appel à des ressources internes et externes à l'entreprise.

2. Expliquer la différence entre les tests unitaires et les tests d'intégration ?
Les tests unitaires permettent de valider chaque module à part en faisant abstraction à leurs interactions avec les autres modules. Les tests d'intégration permettent de valider la cohésion des modules pour valider l'architecture globale.

3. Expliquer la différence entre les tests de validation et de vérification ?
Les tests de validation sont menés par le propriétaire (MOA) afin de vérifier les fonctionnalités développées par rapport au cahier des charges. La vérification est un ensemble de tests menés par l'utilisateur.

Explication :

La validation vérifie que la conception d'un produit correspond à l'usage auquel il est destiné (si le bon objet a été construit) ; la vérification évalue le logiciel qui déterminera si le produit logiciel développé satisfait aux conditions mises en avant (si l'objet construit a été fait correctement).

4. La maintenance du logiciel (ou maintenance logicielle) est menée par le développeur (MOE) afin de modifier un produit logiciel après la livraison, pour en corriger les défauts (cas d'une maintenance corrective), pour améliorer la performance ou d'autres attributs, ou pour adopter le produit à un environnement modifié (cas d'une maintenance évolutive).

5. Les qualités requises dans un cahier de charges sont les suivantes :
   - Bon niveau de généralité ;
   - Formulation adéquate des besoins ? Problème bien décrit ;
   - Être précis, non ambigu malgré l'usage d'un langage naturel (≠ mathématique) ;
   - Être complet (pas d'omission involontaire) ;
   - Être cohérent (pas d'inférence de fonctionnalités) ;
   - Être vérifiable : critères de validation définis. Evaluer la faisabilité des besoins ? faire éventuellement une maquette, une simulation ;
   - Être modifiable : facilité à exprimer un changement ou ajout de besoins.

6. Expliquer pourquoi le modèle de cascade ne reflète pas exactement les activités de développement du logiciel.

En théorie, le modèle en cascade doit créer les conditions pour une réalisation rapide et peu coûteuse des projets par une planification minutieusement élaborée au préalable. Toutefois, les avantages du modèle en cascade sont sujets à controverse dans la pratique. D'une part, les phases du projet sont rarement délimitées clairement dans le développement logiciel. Pour les projets logiciels complexes notamment, les développeurs sont souvent confrontés au fait que les différents composants d'une application se trouvent dans différentes phases de développement au même moment. D'autre part, le déroulement linéaire du modèle en cascade ne correspond souvent pas aux conditions réelles.

7. Expliquer pourquoi il est utile de faire la distinction entre définition des besoins et spécification des besoins.

Les exigences sont ce que votre programme doit faire, les spécifications sont la façon dont vous envisagez de le faire. Une autre façon de voir les choses est que les exigences représentent l'application du point de vue de l'utilisateur ou de l'entreprise dans son ensemble. Donc, la différence clé entre exigence et spécification en génie logiciel est qu'une exigence est un besoin

d'un intervenant auquel le logiciel doit répondre tandis qu'une spécification est un document technique avec les exigences analysées qui décrit les caractéristiques et le comportement d'un logiciel.

8. Comment peut-on réduire l'écart entre les besoins réels d'un client et les besoins exprimés dans le cahier de charges ?

   La réduction de l'écart entre les besoins réels d'un client et les besoin exprimés dans le cahier des charges se fait en utilisant le prototypage. Ceci a conduit à l'idée d'adopter une approche évolutive de développement des systèmes. On présente alors à l'utilisateur un système complet et on enrichi celui-ci à mesure que les besoins réels deviennent apparente une façon de faire et de produire un prototypage qui l'on transforme progressivement en un produit fini.

9. Les phases impliquées dans le modèle de cycle de vie en V sont récapitulées dans le tableau suivant :

Phases / Documents	Avant-projet	Analyse	Conception générale	Conception détaillée	Implémentation	Tests unitaires	Intégration et test d'intégration	Installation et test de réception
Cahier des charges du projet	■	→						
Spécification		■	→					→
Conception générale			■	→			→	
Conception détaillée				■	→	→		
Listages					■	→		
Tests unitaires					?	■		
Test d'intégration				?			■	
Test de réception			?					■
Manuels d'utilisation et d'exploration			→ ?				■	→ ?

■ Document élaboré entièrement pendant la phase. Document partiellement élaboré pendant la phase.

→ Document en entrée de la phase. ? Document éventuellement complété pendant la phase.

Table 1.1 : Phases intervenant dans le modèle de cycle de vie en V.

10. Corrigés des exercices

Corrigé de l'exercice 1

1. Identifier deux risques possibles pour chacune de ces stratégies et proposez des techniques de résolution de risque qui permettraient de décider quelle approche adopter ?

a. Rares sont les applications web ou de bureau capables de fonctionner sans un système de base de données adapté. Le système de gestion de base de données est en effet tout aussi capital que l'ensemble de données à proprement parler puisque sans lui, la base de données ne serait pas fonctionnelle. L'exploitation de son propre base de données, une sécurisation des données complète est capitale.

b. Les systèmes comparables sont gratuits, mais leurs fonctions sont souvent moins avancées que les logiciels propriétaires, parce qu'en cours d'élaboration ou d'amélioration. Certaines fonctions de systèmes comparables ne sont pas développées simplement parce que les auteurs n'y voient pas d'intérêt pour eux, tandis que les logiciels propriétaires sont développés précisément pour répondre aux attentes et aux demandes des clients. Pour remédier à ce problème on doit définir correctement le produit ou le service que l'on veut acheter, en adéquation avec les besoins de l'université. La rédaction d'un cahier des charges permet de spécifier les attentes vis-à-vis de ce produit ou de ce service. Le cahier des charges doit être donc rédigé soit de façon détaillée, soit de façon fonctionnelle. Dans le premier cas, les fournisseurs se contentent d'y répondre point par point. Dans le second cas, le donneur d'ordres fait appel à leur capacité à proposer des solutions.

c. L'établissement d'un cahier des charges commun a vocation à être exhaustif, sa rédaction est longue et technique. De plus, il est important d'avoir une idée très précise de ce que l'on souhaite obtenir avant de le rédiger. Par ailleurs, il ne permet le choix d'une société de logiciels (prestataire) qui développera un seul système pour tous que tard dans le projet, ce qui empêche ainsi le prestataire de jouer pleinement son rôle de conseil. Il est important d'avoir une idée très précise de ce que l'on souhaite obtenir avant de rédiger un cahier des charges en commun. C'est pourquoi, on doit attendre que le projet soit suffisamment mûr. De ce fait, le cahier des charges commun doit permettre à celui qui le lit de comprendre précisément le travail à effectuer. Il doit donc lever toute ambigüité. Enfin, il doit décrire de manière détaillée tous les écrans accessibles aux utilisateurs et les processus de la solution.

2. La liste de fonctionnalités que pourrait contenir cette application, est comme suit :

Espace administration :

+ Gestion des étudiants (nom, prénom, carteIdentNatio, dateNaissance, adresse, niveau, filière, groupe, etc.)

- Ajout d'un étudiant
- Modification d'un étudiant
- Suppression d'un étudiant
- Consulter la liste des étudiants / Chercher
- Importer une liste des étudiants
- Gestion des filière/parcours (nom, abréviation)
- Gestion niveau (titre, filière)
- Gestion des groupes (nom, niveau)
- Gestion des matières (nom, coefficient, volume horaire, crédit, enseignant)

- Gestion des enseignants (nom, prénom, téléphone, email, grade, spécialité)
- Gestion des grades (titre, charge d'enseignement)
- Gestion des spécialités (titre)

Espace enseignant :

- Authentification
- Mise à jour des informations personnelles
- Consulter les matières / étudiants
- Consulter les étudiants
- Gestion des notes (saisi / Mise à jours)

Espace étudiant :

- Authentification
- Mise à jour des informations personnelles
- Consulter les matières / enseignants
- Consulter les notes

Chapitre 2 : Modélisation avec UML (Unified Modeling Language)

Objectif

À l'issue de ce chapitre, l'apprenant sera capable de :

- Maîtriser les notions de Modélisation, Modèle, Modélisation Orientée Objet, UML en application ;
- Connaître les différentes versions de l'UML survenues entre 1994 et 2017 ;
- Maîtriser les mécanismes généraux.

1. Introduction

UML (Unified Modeling Language) est un langage de modélisation standardisé à usage général dans le domaine de l'ingénierie logicielle orientée objet. Ce langage est constitué d'un ensemble de techniques de notation graphique utilisées pour créer des modèles visuels de systèmes logiciels orientés objet. De plus, UML combine des techniques de modélisation de données, de modélisation d'entreprise, de modélisation d'objets et de modélisation de composants et peut être utilisé tout au long de cycle de vie du développement logiciel et à travers différentes technologies de mise en œuvre.

2. Modélisation

Une modélisation est une représentation simplifiée d'une réalité. Elle permet de capturer des aspects pertinents pour répondre à un objectif défini a priori.

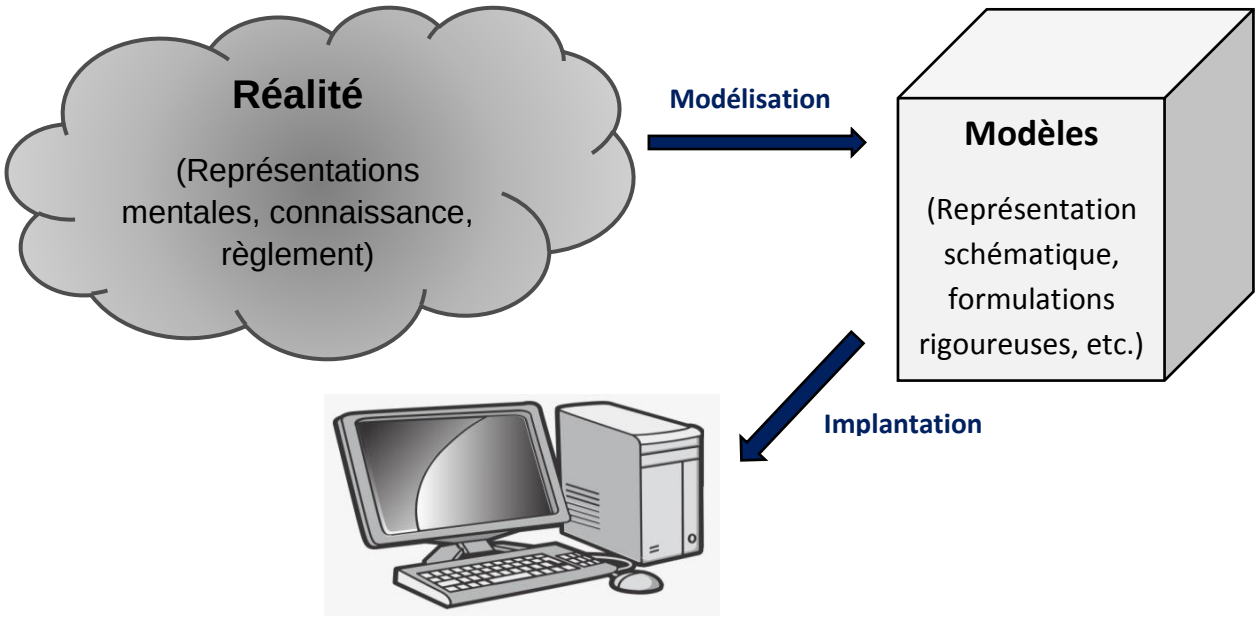


Figure 2.1. Exemple type d'une modélisation.

Exemple :
Un astronaute modélisera la Lune comme un corps céleste ayant une certaine masse et se trouvant à une certaine distance de la Terre, alors qu'un poète la modélisera comme une dame avec laquelle il peut avoir une conversation.

3. Modèle

Un modèle est une représentation abstraite de la réalité qui exclut certains détails du monde réel. Il permet de réduire la complexité d'un phénomène en éliminant les détails qui n'influencent pas son comportement de manière significative. De plus, il reflète ce que le concepteur croit important pour la compréhension et la prédiction du phénomène modélisé, les limites du phénomène modélisé dépendent des objectifs du modèle.

Le modèle s'exprime sous une forme simple et pratique pour le travail. Quand le modèle devient compliqué, il est souhaitable de le décomposer en plusieurs modèles simples et manipulables. L'expression d'un modèle se fait dans un langage compatible avec le système modélisé et les objectifs attendus. Ainsi, le physicien qui modélise la lune utilisera les mathématiques comme langage de modélisation.

Remarque

Dans le cas du logiciel, l'un des langages utilisés pour la modélisation est le langage UML (Unified Modeling Language). Il possède une sémantique propre et une syntaxe composée de graphique et de texte et peut prendre plusieurs formes (diagrammes).

4. Modélisation Orientée Objet

La Conception Orientée Objet (COO) est la méthode qui conduit à des architectures logicielles fondées sur les objets du système, plutôt que sur une décomposition fonctionnelle. La Conception Orientée Objet est caractérisée par :

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

5. UML en application

Au milieu des années quatre-vingt-dix, les auteurs de Booch, OOSE (Object Oriented Software Engineering) et OMT (Object Modeling Technique) ont décidé de créer un langage de modélisation unifié avec pour objectifs :

- Modéliser un système des concepts à l'exécutable, en utilisant les techniques orientée objet ;
- Réduire la complexité de la modélisation ;
- Utilisable par l'homme comme la machine :
    - Représentations graphiques mais, disposant de qualités semi-formelles suffisantes pour être traduites automatiquement en code source ;

- Ces représentations ne disposent cependant pas de qualités formelles suffisantes pour justifier d'aussi bonnes propriétés mathématiques que des langages de spécification formelle (Z, VDM, etc.).

– Officiellement UML est né en 1994.

Remarque

Il existe des langages à différents niveaux de formalisation, à savoir : (i) langages formels (ex. Z, B, VDM) : le plus souvent mathématiques, au grand pouvoir d'expression et permettant des preuves formelles sur les spécifications ; (ii) langages semi-formels (ex. MERISE, UML, Orienté but, etc.) : le plus souvent graphiques, au pouvoir d'expression moindre mais plus faciles d'emploi.

6. Historique d'UML

Les grandes étapes de la diffusion d'UML peuvent se résumer comme suit :
1994 - 1996 : rapprochement des méthodes OMT, BOOCH et OOSE et naissance de la première version d'UML.
23 novembre 1997 : version 1.1 d'UML adoptée par l'OMG.
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.
⋮
Décembre 2017 : sortie de la version 2.5.1.

7. Mécanismes généraux

7.1 Les étiquettes

Une étiquette est une paire (nom, valeur) qui ajoute une nouvelle propriété à un élément de modélisation. Une étiquette ou valeur marquée (tagged value) permet la création de nouvelles informations dans la spécification d'un élément.

7.2 Les stéréotypes

Un stéréotype est une annotation s'appliquant sur un élément de modèle. Il n'a pas de définition formelle, mais permet de mieux caractériser des variétés d'un même concept. Il permet donc d'adapter le langage à des situations particulières.

Remarque

Le stéréotype est représenté par une chaînes de caractères entre guillemets « nom du stéréotype » dans, ou à proximité du symbole de l'élément de modèle de base.

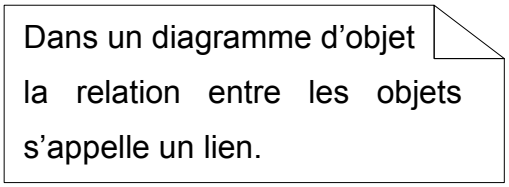
7.3 Les notes

Une note est un commentaire explicatif attaché à un ou plusieurs éléments de modélisation. Graphiquement, une note 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.

Notation graphique

Exemple :



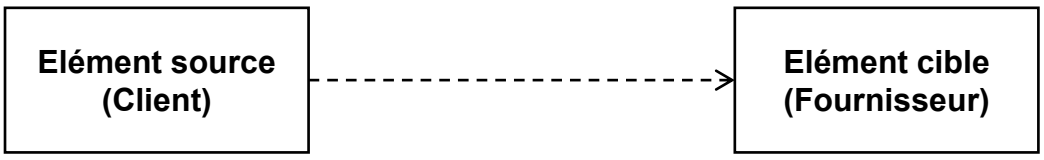
7.4 Les contraintes

Une contrainte constitue une condition ou une restriction sémantique exprimée sous forme d'instruction dans un langage textuel qui peut être naturel ou formel. En général, une contrainte peut être attachée à n'importe quel élément de modèle ou liste d'éléments de modèle. On représente une contrainte sous la forme d'une chaîne de texte placée entre accolades { }. La chaîne constitue le corps écrit dans un langage de contrainte qui peut être :

- Naturel ;
- Dédié, comme OCL (Object Constraint Language) ;
- Ou encore directement issu d'un langage de programmation.

7.5. La relation de dépendance

La relation de dépendance, notée par une flèche pointillée, définit une relation d'utilisation unidirectionnelle entre deux éléments de modélisation, appelés respectivement source (exemple : client) et cible (exemple : fournisseur) de la relation.



Remarque

Une relation de dépendance peut également être dotée d'une note et/ou d'une contrainte.

7.6. Dichotomies (type, instance) et (type, classe)

De nombreux éléments de modélisation présentent une dichotomie (type, instance), dans laquelle le type dénote l'essence de l'élément, et l'instance avec ses valeurs correspond à une

manifestation de ce type. De même, la dichotomie (type, classe) correspond à la séparation entre la spécification d'un élément qui est énoncé par le type et la réalisation de cette spécification qui est fournie par la classe.

8. Les diagrammes UML

UML comporte plusieurs types de diagrammes représentant autant de vues distinctes pour représenter des concepts particuliers du système d'information. Il existe trois axes de modélisation : fonctionnel, statique, dynamique.

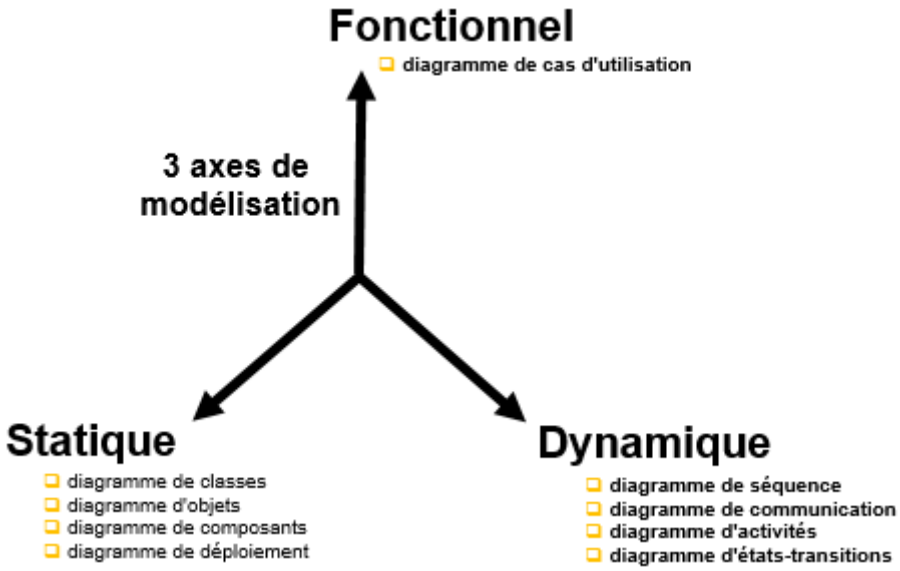


Figure 2.2 : Diagrammes d'UML par axes de modélisation.

Ces diagrammes, d'une utilité variable selon les cas, ne sont pas nécessairement tous produits à l'occasion d'une modélisation. Les plus utiles pour la maîtrise d'ouvrage sont les diagrammes de cas d'utilisation, de classes, d'objets, de séquence, d'activités et d'états-transitions. Les diagrammes de composants, et de déploiement sont surtout utiles pour la maîtrise d'œuvre à qu'ils permettent de formaliser les contraintes de la réalisation et la solution technique.

9. Paquetages

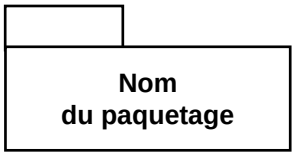
Un paquetage est un regroupement d'éléments de modèle et de diagrammes. Il permet ainsi d'organiser des éléments de modélisation en groupes. Il peut contenir tout type d'élément de modèle : des classes, des cas d'utilisation, des interfaces, des diagrammes, etc., et même des paquetages imbriqués.

Les éléments contenus dans un paquetage doivent représenter un ensemble fortement cohérent et sont généralement de même nature et de même niveau sémantique. Généralement, il existe une seule racine qui détient la totalité du modèle d'un système.

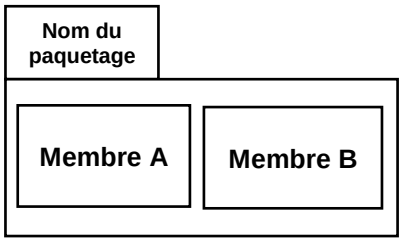
Notation graphique

Le formalisme général d'un paquetage et les trois manières de présenter un paquetage sont les suivantes :

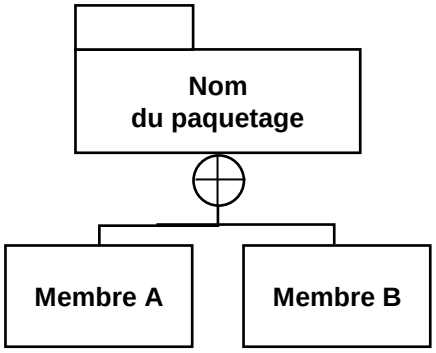
- **Représentation globale :** Le nom du paquetage se trouve à l'intérieur du grand rectangle.



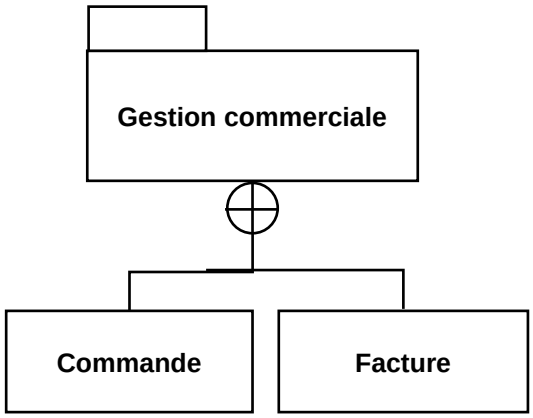
- **Représentation détaillée :** Les membres du paquetage sont représentés et le nom du paquetage d'ensemble s'inscrit dans le petit rectangle.



- **Représentation éclatée :** Les membres du paquetage sont reliés par un lien connecté au paquetage par le symbole ⊕.



Exemple :



10.Conclusion

Bien que la modélisation UML soit une arme utile pour la modélisation logicielle, elle doit être utilisée selon certaines règles, sinon elle ne pourra pas faire jouer pleinement sa valeur, et ce sera la moitié de l'effort. Comprenez les prérequis pour utiliser le langage de modélisation UML et implémentez-les soigneusement selon ces méthodes, je pense qu'il y aura des résultats idéaux. Ce chapitre étant introductif et afin de ne pas privilégier les aspects théoriques sur les aspects pratiques, ce chapitre ne comporte donc pas d'exercices. Le chapitre suivant portera sur le diagramme de cas d’utilisation (vue fonctionnelle).

Chapitre 3 : Diagramme de cas d'utilisation (Vue fonctionnelle)

Objectif

À l'issue de ce chapitre, l'apprenant sera capable de :

- Maîtriser les concepts de base des diagrammes de cas d'utilisation ;
- Apprendre à établir le diagramme de cas d'utilisation ;
- Décrire textuellement des cas d'utilisation.

1. Introduction

Dans les méthodes de développement de logiciels traditionnelles et les premières méthodes de développement orientées objet, le langage naturel est utilisé pour décrire les exigences fonctionnelles du système. Une telle approche n'a pas de format unifié, manque de formalisation de la description et est relativement aléatoire, ce qui est sujet à la confusion et à l'inexactitude dans la compréhension. Lorsque l'auteur d'UML a proposé le modèle de diagramme de cas d'utilisation, ces problèmes ont été bien résolus. La fonction principale du ce diagramme : (1) utilisé pour décrire les exigences fonctionnelles et les scénarios d'utilisation du système du système à développer, (2) en tant que base du processus de développement, pilotez le travail de développement à toutes les étapes, et (3) utilisé pour vérifier et confirmer la configuration système requise.

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

Un diagramme de cas d'utilisation est l'un des quatorze (14) types de diagramme du langage de modélisation UML, un langage utilisé pour modéliser la structure et le comportement des logiciels et d'autres systèmes. Il décrit les cas d'utilisation et les acteurs avec leurs dépendances et relations respectives. Le diagramme de cas d'utilisation est un diagramme de comportement. Il représente le comportement attendu d'un système et est donc utilisé pour spécifier les exigences d'un système.

3. Concepts de base

Le diagramme de cas d'utilisation est destiné à représenter les besoins des utilisateurs par rapport au système. Le diagramme de cas d'utilisation comprend les éléments suivants : l'acteur, le cas d'utilisation et l'interaction entre l'acteur et le cas d'utilisation.

3.1. Acteur

Un acteur représente un rôle d'une entité externe qui interagit avec le système étudié. L'entité externe peut être un utilisateur humain, une organisation, une machine ou un autre système externe.

Il se représente symboliquement par un petit bonhomme avec son nom (i.e. son rôle) inscrit dessous.

Nom de l'acteur

Il est également possible de représenter un acteur par rectangle stéréotypé « acteur » avec son nom (i.e. son rôle) inscrit dessous.

« acteur »
Nom de l'acteur

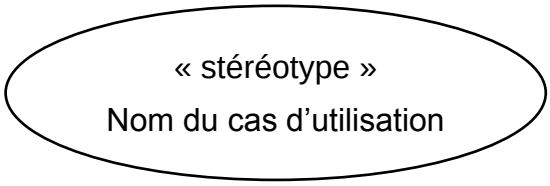
Remarque

Plusieurs utilisateurs peuvent avoir le même rôle, et donc correspondre à un même acteur, et une même personne physique peut jouer des rôles différents vis-à-vis du système, et donc correspondre à plusieurs acteurs.

3.2. Cas d'utilisation

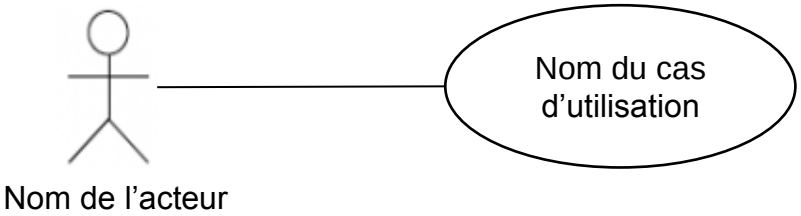
Un cas d'utilisation décrit à un certain nombre d'actions qu'un système devra exécuter pour atteindre l'objectif d'un acteur. En effet, un cas d'utilisation doit renvoyer un résultat observable qui est utile pour l'acteur du système.

Un cas d'utilisation se représente graphiquement par un ovale contenant l'intituler du cas (un verbe à l'infinitif), et optionnellement, au-dessus du l'intitulé, un stéréotype.



3.3. Interaction

La relation entre un acteur et un cas d'utilisation représentée par une ligne de connexion/interaction, appelée association. Une ligne continue entre un acteur et un cas d'utilisation indique clairement qu'une communication est établie.

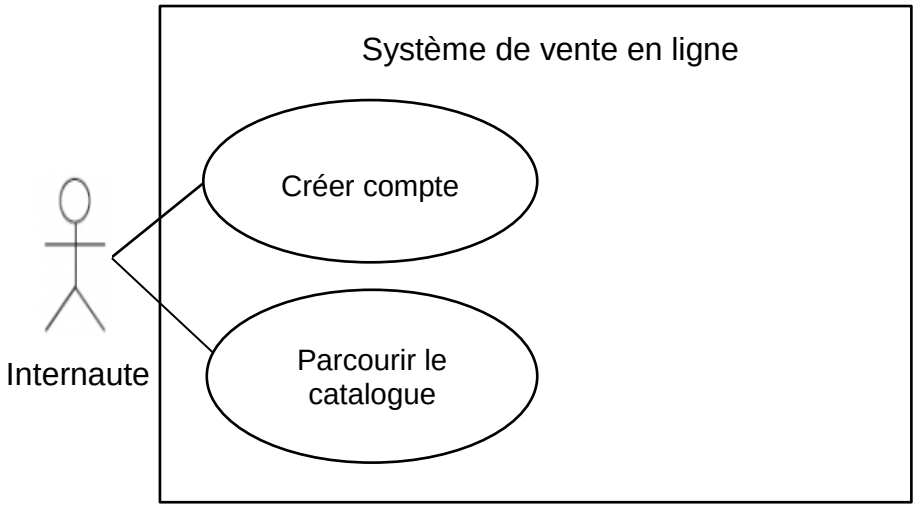


3.4. Limite du système

Tous les cas d'utilisation peuvent être placés dans les limites du système, indiquant qu'ils appartiennent à un système. Le nom du système figure à l'intérieur du cadre, et les participants

(acteurs) sont placés en dehors des limites du système, indiquant que les participants n'appartiennent pas au système. Mais, le participant est responsable de piloter directement (ou indirectement) l'exécution du cas d'utilisation qui lui est associé.

Exemple :



Remarque

La limite d'un système est un rectangle que vous pouvez dessiner dans un diagramme de cas d'utilisation, pour séparer les cas d'utilisation appartenant au système et les acteurs extérieurs. Cette limite est une aide visuelle facultative pour le diagramme : elle n'apporte aucune valeur sémantique au modèle.

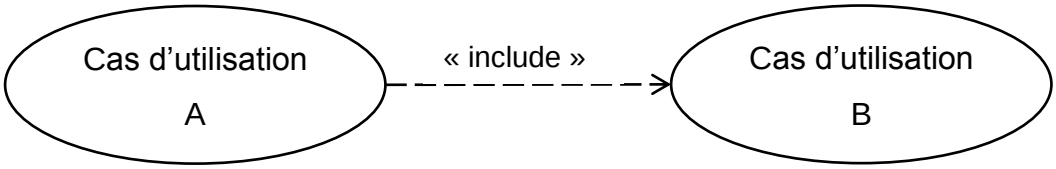
Un acteur est qualifié de principal pour un cas d'utilisation lorsque l'acteur initie les échanges nécessaires à la réalisation du cas d'utilisation. En effet, un acteur principal obtient un résultat observable du système. Cependant, un acteur secondaire n'est sollicité que pour des informations supplémentaires. Par conséquent, l'acteur principal sollicite le cas d'utilisation tandis que l'acteur secondaire est sollicité par le cas d'utilisation. Par convention et dans la mesure du possible les acteurs principaux sont représentés à gauche du système alors que les acteurs secondaires sont représentés à droite du système.

4. Relations entre cas d'utilisation

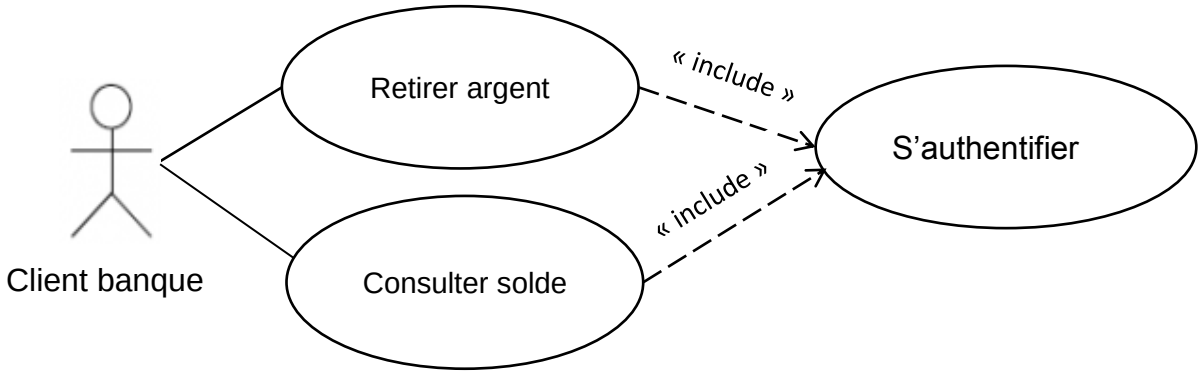
Le but des relations entre les cas d'utilisation est de décomposer le système en fonctionnalités plus fines. Trois types de décompositions ou de relation standard entre cas d'utilisation sont proposés par UML : relation d'inclusion, relation d'extension et relation de généralisation.

4.1. Relation d'inclusion

Une relation d'inclusion est une relation dans laquelle un cas d'utilisation source (cas d'utilisation A) inclut les fonctionnalités de cas d'utilisation destination (cas d'utilisation B). La relation d'inclusion prend en charge aussi la réutilisation des fonctionnalités dans un modèle de cas d'utilisation. En d'autres termes, elle permet de décomposer des comportements et de définir des comportements partageables entre plusieurs cas d'utilisation. Elle est symbolisée avec une dépendance étiquetée par le stéréotype « include » (ou « inclut » en français).

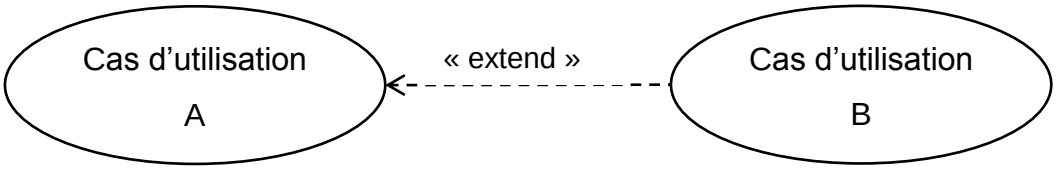


Exemple :
Dans cet exemple, le retrait d'argent et la consultation du solde sont des transactions sécurisées et nécessitent par conséquent une authentification.

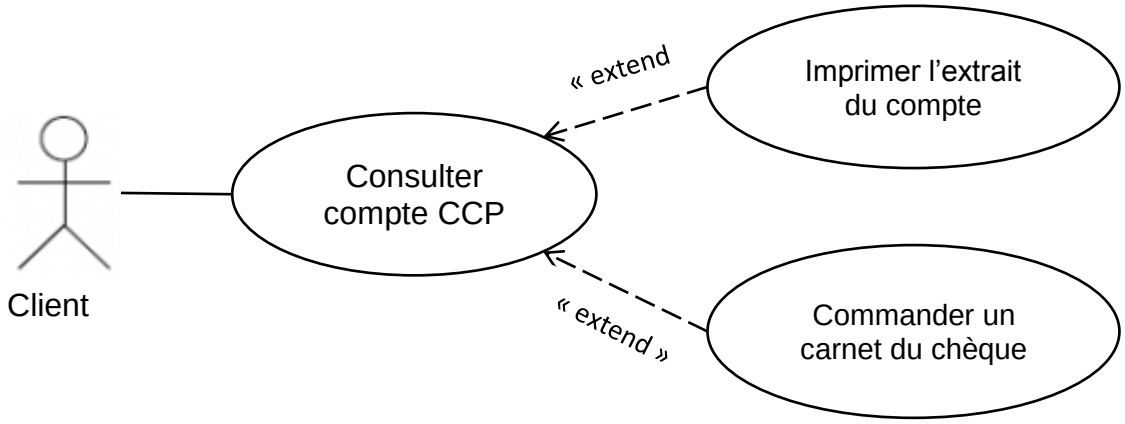


4.2. Relation d'extension

Une relation d'extension peut être utilisée pour spécifier qu'un cas d'utilisation d'extension (cas d'utilisation B) étend le comportement d'un autre cas d'utilisation destination de base (cas d'utilisation A). La relation d'extension est symbolisée par une dépendance étiquetée avec le stéréotype «extend» (ou « étend » en français).



Exemple :

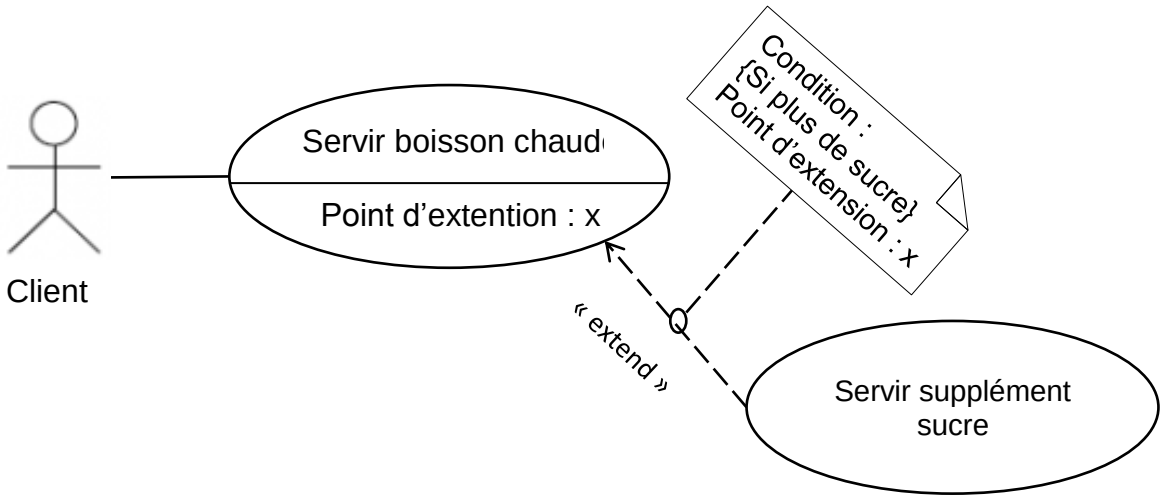


Un point d'extension identifie le point du cas d'utilisation de base où le comportement d'un cas d'utilisation d'extension peut être inséré. Ce point est spécifié pour un cas d'utilisation de base et est référencé par une relation d'extension entre le cas d'utilisation de base et le cas d'utilisation

d'extension. Un cas d'utilisation de base peut avoir plusieurs points d'extension. Chaque point d'extension porte un nom unique qui identifie un ou plusieurs emplacements dans la séquence de comportement du cas d'utilisation de base. Le nom de l'extension est indiqué dans le compartiment Points d'extension du cas d'utilisation. De plus, le point d'extension est éventuellement associé à une contrainte indiquant le moment où l'extension intervient. Une extension est souvent soumise à condition.

La condition de la relation d'extension ainsi que les références aux points d'extension sont facultativement affichées dans une note jointe à la relation d'extension correspondante.

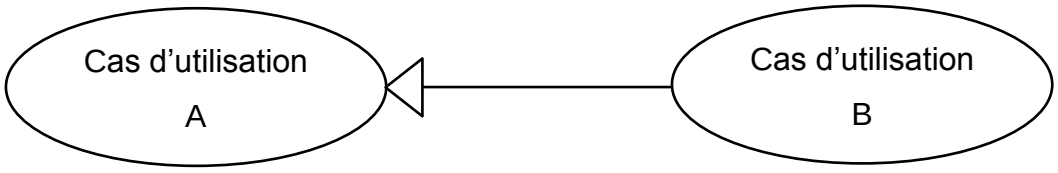
Exemple :



4.3. Relation de généralisation

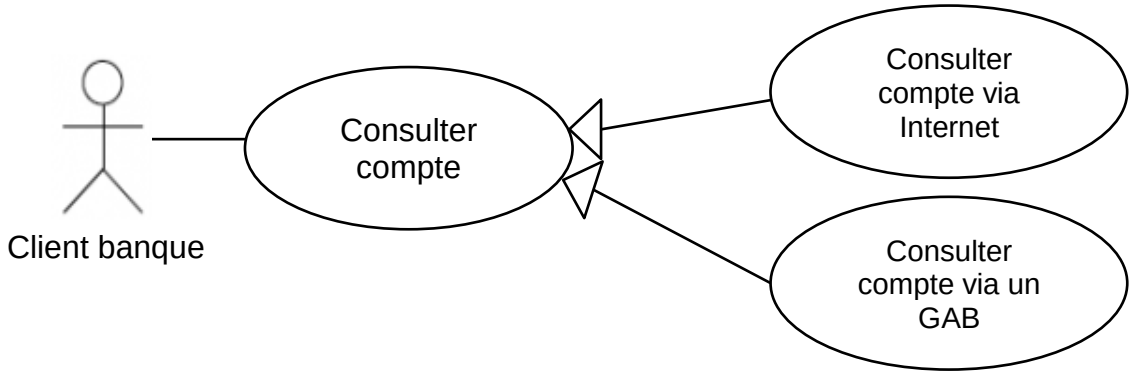
La relation de généralisation/spécialisation est présente dans la plupart des diagrammes UML et se traduit par le concept d'héritage dans les langages orientés objet. Le cas d'utilisation A est une généralisation de B, si B est un cas particulier de A. B est un cas particulier de A ne serait-ce que si A peut être remplacé par B.

La relation de généralisation entre cas d'utilisation est représentée symboliquement par une ligne continue avec une flèche triangulaire creuse indiquant le cas d'utilisation le plus général.



Exemple :

Dans cet exemple, on représente le fait qu'un client peut consulter le compte soit via Internet soit via un Guichet Automatique Bancaire (GAB).



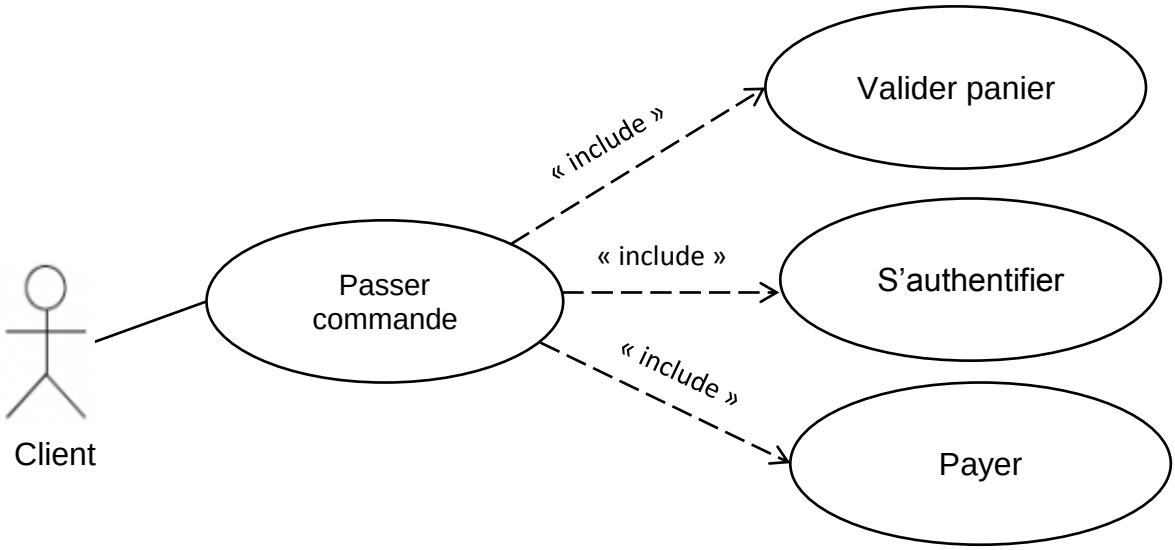
Dans cet exemple, la consultation d'un compte via Internet est un cas particulier de la consultation.

Remarque

Quand un cas n'est pas directement relié à un acteur, il est qualifié de cas d'utilisation interne.

Lorsqu'un cas d'utilisation est trop complexe ou fait intervenir un trop grand nombre d'actions, on peut procéder à sa décomposition en cas d'utilisation plus simples. En effet, les inclusions permettent essentiellement de factoriser une partie de la description d'un cas d'utilisation qui serait commune à d'autres cas d'utilisation. En d'autres termes, les inclusions permettent de décomposer un cas complexe en sous cas plus simples.

Exemple : Relations entre cas pour décomposer un cas complexe



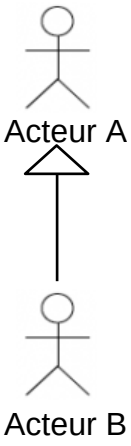
Remarque

1. Il ne faut surtout pas abuser de ce type de décomposition : il faut éviter de réaliser du découpage fonctionnel d'un cas d'utilisation en plusieurs sous cas d'utilisation pour ne pas retomber dans le travers de la décomposition fonctionnelle.
2. Les cas d'utilisation ne s'enchaînent pas, car il n'y a aucune représentation temporelle dans un diagramme de cas d'utilisation.

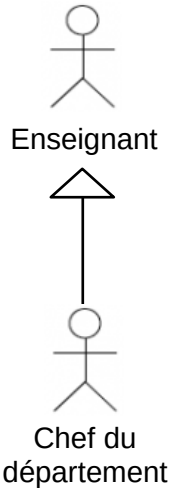
5. Relations entre acteurs

La relation de généralisation est le seul type de relation possible entre deux acteurs. Cette relation exprime le fait que l'acteur qui est du côté opposé à la pointe de la flèche (acteur B) est spécialisé dans le sens où il peut réaliser tout ce que l'acteur le plus général (acteur A) peut réaliser, plus d'autres fonctionnalités.

La relation de généralisation entre acteurs est symbolisée par un trait plein avec une flèche triangulaire creuse indiquant l'acteur le plus général.



Exemple :



Le chef du département est un enseignant avec un pouvoir supplémentaire : en plus de pouvoir enseigner, guider et conseiller les étudiants, il est responsable du fonctionnement pédagogique et administratif du département et il exerce l'autorité hiérarchique sur le personnel placé sous sa responsabilité. Par contre, l'enseignant ne peut pas faire les tâches confiées au chef du département.

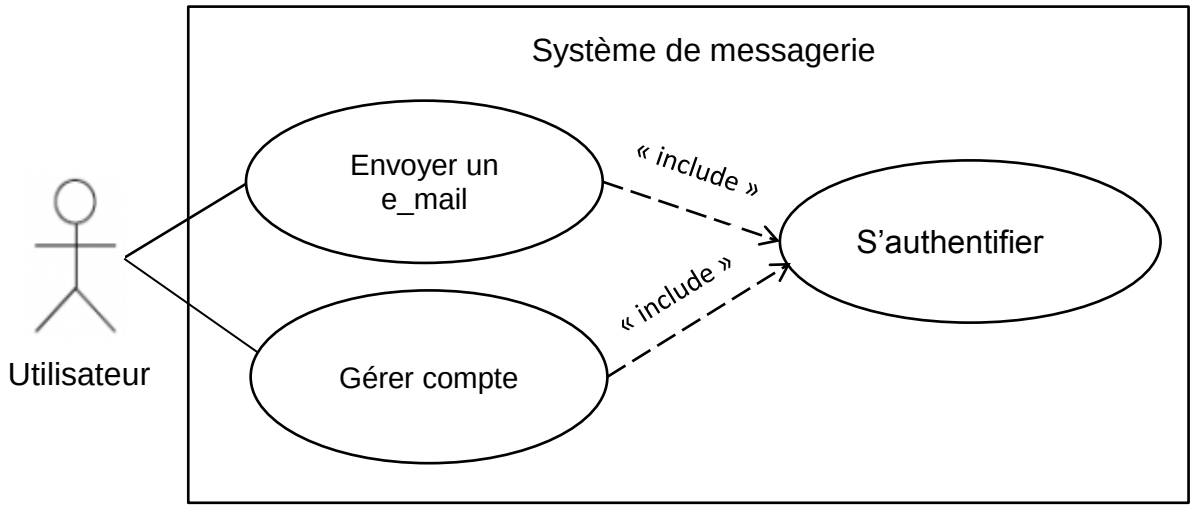
6. Description textuelle des cas d'utilisation

Une fois les cas d'utilisation identifiés, il faut les décrire. Cette description repose sur la notion de scénario. Un scénario représente une succession particulière d'enchaînements s'exécutant du début à la fin du cas d'utilisation. Les scénarios doivent décrire l'interaction entre l'acteur et le système. Ils n'ont pas pour but de décrire comment le système réalise les échanges.
UML ne propose pas de présentation type de cette description textuelle. Cependant, les travaux menés par Alistair Cockburn sur ce sujet constituent une référence en la matière et tout naturellement je reprends ici l'essentiel de la présentation récapitulée dans la table 3.1.

Sommaire d'identification	
Titre	Nom du cas d'utilisation
Résumé	But du cas d'utilisation
Acteurs	Acteurs participants aux cas d'utilisation (principaux et secondaires)
Description des scénarios	
Préconditions	Conditions décrivant dans quel état doit être le système avant que le cas d'utilisation puisse être déclenché
Scénario nominal	Séquences d'actions associées au cas d'utilisation se déroulant lorsqu'il n'y a pas d'erreurs
Enchaînements alternatifs	C'est un embranchement dans un scénario nominal mais y revient toujours.
Enchainement d'erreurs (*d'exception*)	Séquences d'action conduisant à un échec. En d'autres termes, un scénario d'exception intervient quand une erreur se produit, le séquencement nominal s'interrompt, sans retour à l'enchaînement nominal.
Postconditions	Conditions décrivant l'état du système à l'issue des différents scénarios

Table 3.1 : Présentation type de description textuelle des cas d'utilisation.

Exemple :
Soit le diagramme de cas d'utilisation d'un système de messagerie suivant :



La description textuelle récapitulée dans la table 3.2 montre une façon de décrire les interactions pour s'authentifier.

Sommaire d'identification	
Titre	Authentification
Résumé	L'authentification permet à l'utilisateur l'accès à son compte et l'affichage de la liste des emails
Acteurs	Utilisateur
Description des scénarios	
Préconditions	Application accessible
Scénario nominal	1. L'utilisateur introduit son nom de compte et son mot de passe 2. Le système vérifie l'existence du compte et du mot de passe 3. Le système collecte les emails de l'utilisateur 4. Le système affiche les emails à l'utilisateur
Enchainement d'erreurs (*d'exception*)	2a. Aucun utilisateur ne correspond au compte et mot de passe introduits. Dans ce cas le système renvoie un message d'erreur.
Postconditions	L'utilisateur accède à ses emails

Table 3.2 : Description textuelle de cas d'utilisation « **s'authentifier** ».

7. Conclusion

Les diagrammes de cas d'utilisation sont utilisés pour capturer la nature dynamique du système. Il est utilisé par le contexte des participants et leurs interrelations. Un diagramme de cas d'utilisation de conception de haut niveau est utilisé pour capturer les exigences du système. Cela représente donc la fonction et le flux du système. Bien que l'ingénierie directe et inverse des diagrammes de cas d'utilisation ne soit pas un bon choix, ils le simulent toujours d'une manière légèrement différente. Le chapitre suivant sera consacré aux diagrammes de classes et d'objets qui représentent une vue graphique de la structure statique d'un système, exprimée en termes de classes (respectivement objets) et de relations entre ces classes (respectivement objets).

8. Exercices

Exercice 1 :

Dans un magasin, le processus de vente est le suivant : le client entre, passe dans les rayons, demande éventuellement des renseignements ou procède à des essais, prend des articles (si le stock est suffisant), passe à la caisse où il règle ses achats (avec tout moyen de paiement accepté). À noter que les moyens de paiement réalisés au sein d'un commerce physique le sont en espèces, par carte bancaire ou par chèque. Enfin, le client peut éventuellement bénéficier d'une réduction.

Travail demandé :

Modéliser cette situation par un diagramme de cas d'utilisation.

Exercice 2 :

Un nouveau Système de Gestion de Stock (SGS) doit être mis en place. Celui-ci sera utilisé par le service stock et doit couvrir les activités de gestion des stocks et de réapprovisionnement. Dans le service stock, se côtoient les chefs d'équipes et les magasiniers. Les magasiniers sont responsables par la réception des marchandises et leur expédition aux autres magasins de la société. Les chefs d'équipe gèrent le réapprovisionnement : ils soumettent aux fournisseurs les demandes de devis pour les marchandises et élaborent les commandes. Par contre, tous les employés, qu'ils soient chef d'équipe ou magasinier, sont responsables de la gestion du catalogue (inclusion des nouveaux produits, mise à jour des produits, etc.). Toutes ces tâches seront gérées par le système de gestion SGS. Pendant le réapprovisionnement les magasins transmettent au service de stock leurs besoins. Le chef d'équipe vérifie si les marchandises demandées sont disponibles. Si c'est le cas, un ordre d'expédition est transmis aux magasiniers, qui vont expédier les produits au magasin qui les a demandés.

Travail demandé :

Modéliser cette situation par un diagramme de cas d'utilisation.

Exercice 3 :

Les clients d'une vidéothèque en ligne ont une première idée à quoi ça ressemble la gestion des utilisateurs et des fonctionnalités qui devraient être disponibles. Un développeur prend les notes suivantes dans une conversation :

a. Pour utiliser les fonctions de la vidéothèque en ligne, les clients doivent d'abord se connecter. Les clients non connectés ne voient même pas les fonctions de la vidéothèque.
b. Lors de la connexion, il vérifie s'il y a des nouvelles pour le client (par exemple, à propos de nouveaux films qui correspondent à leurs locations précédentes). Si c'est le cas, un message sera affiché.
c. La connexion réussie mène automatiquement à l'affichage de la page principale de la vidéothèque, à partir de laquelle toutes les fonctions de la vidéothèque seront accessibles : remplir des crédits, afficher le catalogue de films, et emprunter des films.
d. Un film à emprunter doit d'abord être sélectionné par l'utilisateur. La note FSK (vignette FSK indiquant que le film est recommandé à partir de 12 ans) du film est comparée à l'âge du client et, le cas échéant, le prêt du film sera refusé.
e. Par la suite, le crédit du client est vérifié et un crédit trop faible est également refusé. Sinon, le solde créditeur du client sera réduit du prix du film, un lien de téléchargement spécifique à ce prêt sera généré et envoyé au client.
f. Les clients ne peuvent recharger leur solde qu'avec un paiement en ligne sécurisé. Pour un paiement en ligne sécurisé, le système utilise le fournisseur de services externe "SuperPay".
g. Un administrateur connecté doit pouvoir exécuter toutes les fonctions qu'un client connecté peut effectuer. En outre, il devrait pouvoir augmenter directement le solde

créditeur d'un client (pour des raisons de clientèle, rédiger un bonus pour la réalisation d'un chiffre d'affaires spécifique, etc.).

Travail demandé :

Modéliser cette situation à l'aide d'un diagramme de cas d'utilisation.

Exercice 4 :

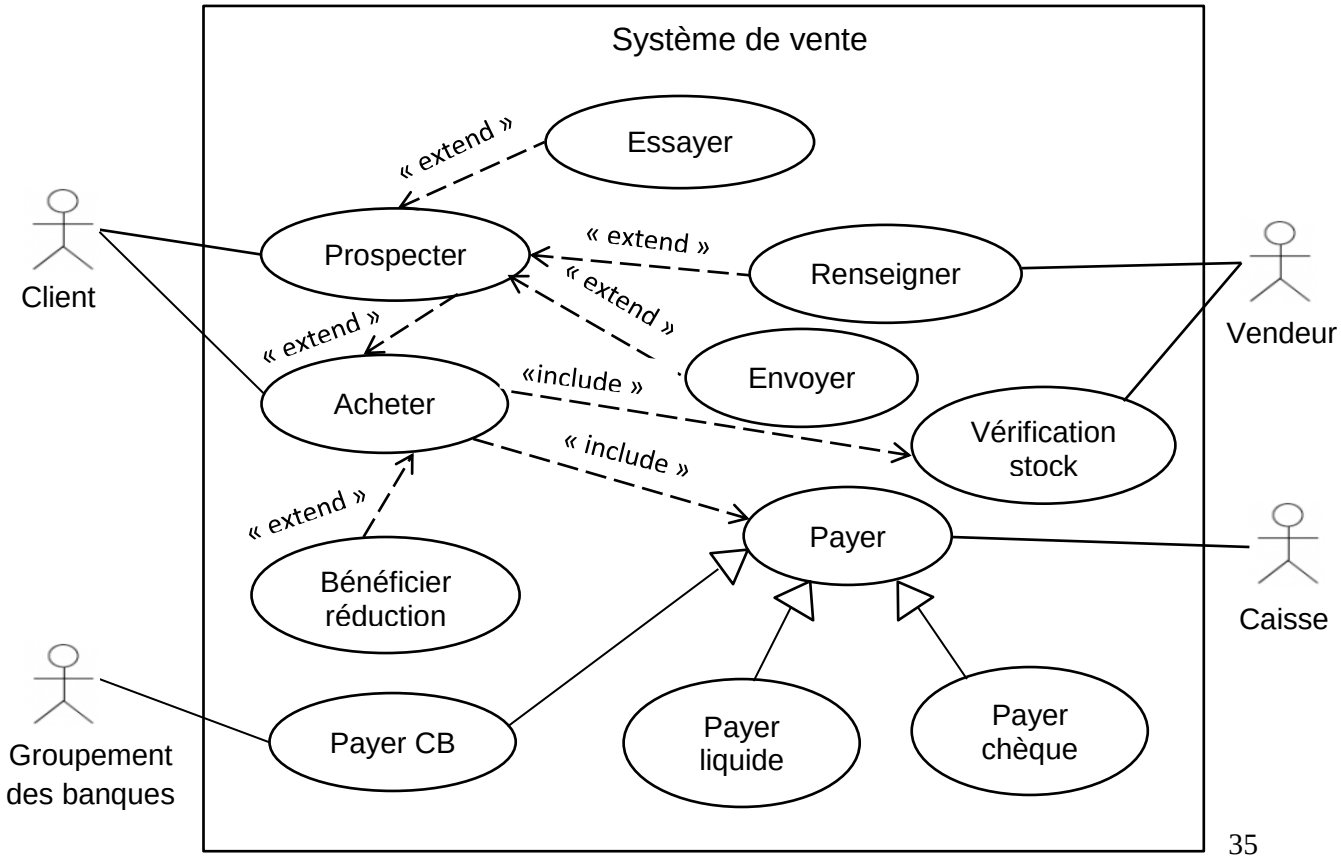
Une entreprise souhaite modéliser avec UML le processus de formation de ses employés afin d'informatiser certaines tâches. Le processus de formation est initialisé quand le responsable de formation reçoit une demande de formation d'un employé. Cet employé peut éventuellement consulter le catalogue des formations offertes par les organismes agréés par l'entreprise. Cette demande est instruite par le responsable qui transmet son accord ou son refus à l'employé. En cas d'accord, le responsable cherche la formation adéquate dans les catalogues des formations agréées qu'il tient à jour. Il informe l'employé du contenu de la formation et lui soumet la liste des prochaines sessions prévues. Lorsque l'employé à fait son choix, le responsable de formation l'inscrit à la session retenue auprès de l'organisme de formation concerné. En cas d'empêchement l'employé doit avertir au plus vite le responsable de formation pour que celui-ci demande l'annulation de l'inscription. À la fin de la formation l'employé transmet une appréciation sur le stage suivi et un document attestant sa présence. Le responsable de formation contrôle la facture envoyée par l'organisme de formation.

Travail demandé :

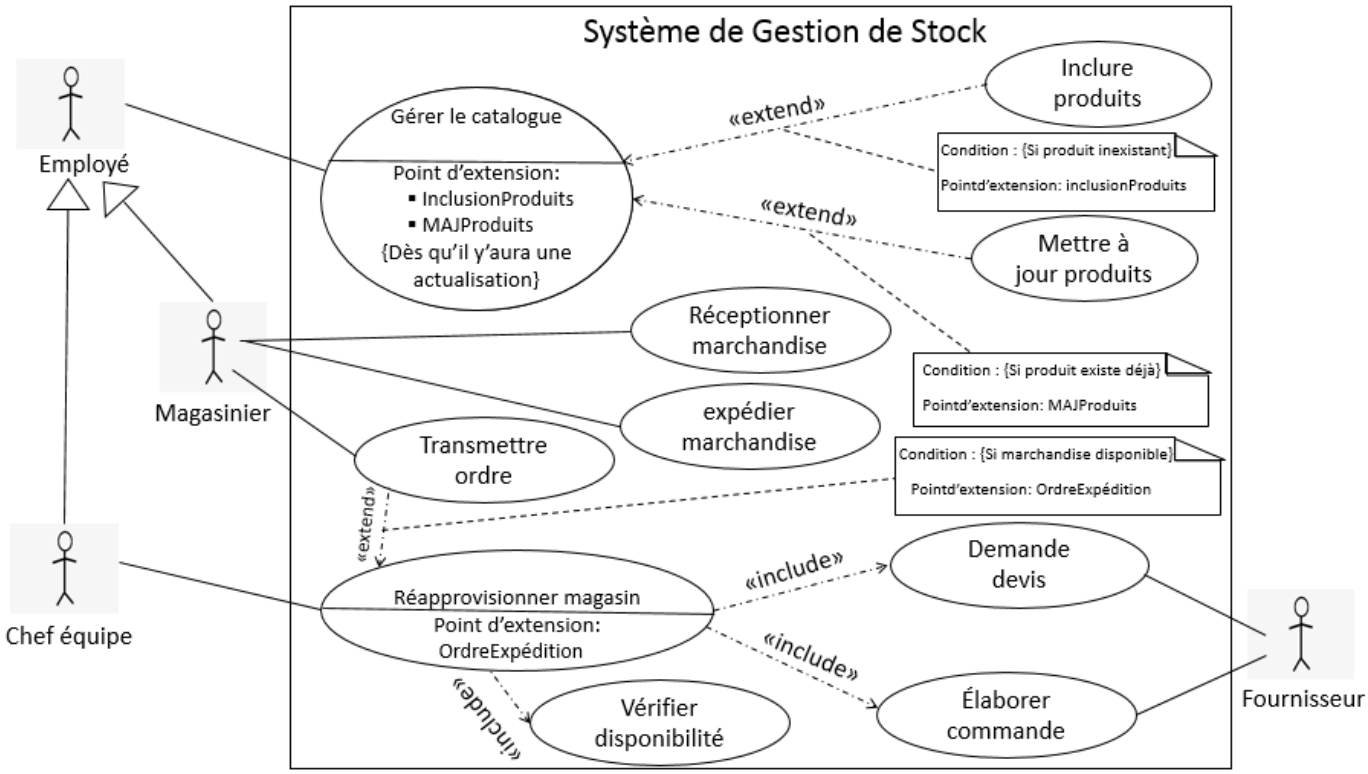
Modéliser cette situation à l'aide d'un diagramme de cas d'utilisation.

9. Corrigés des exercices

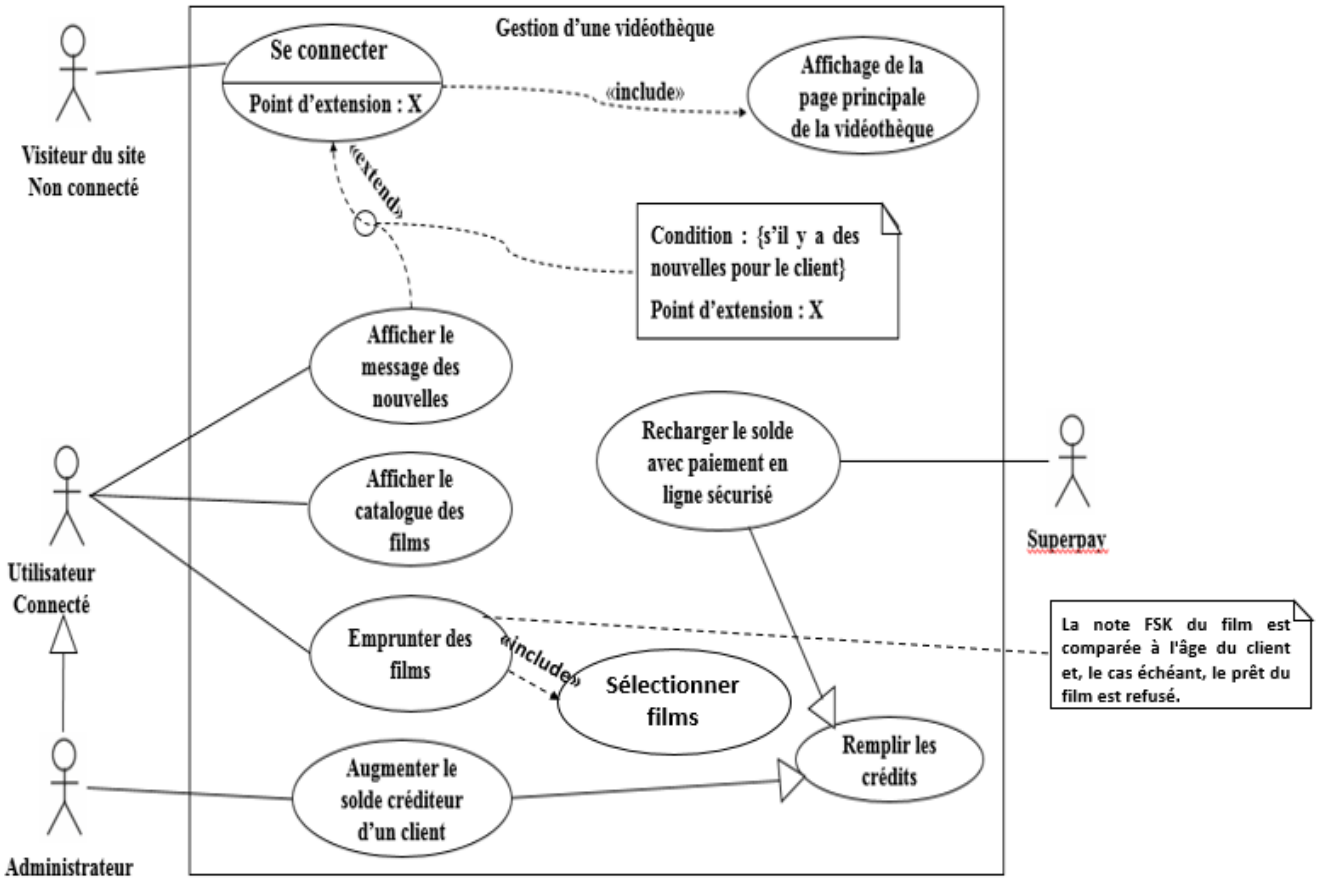
Corrigé de l'exercice 1



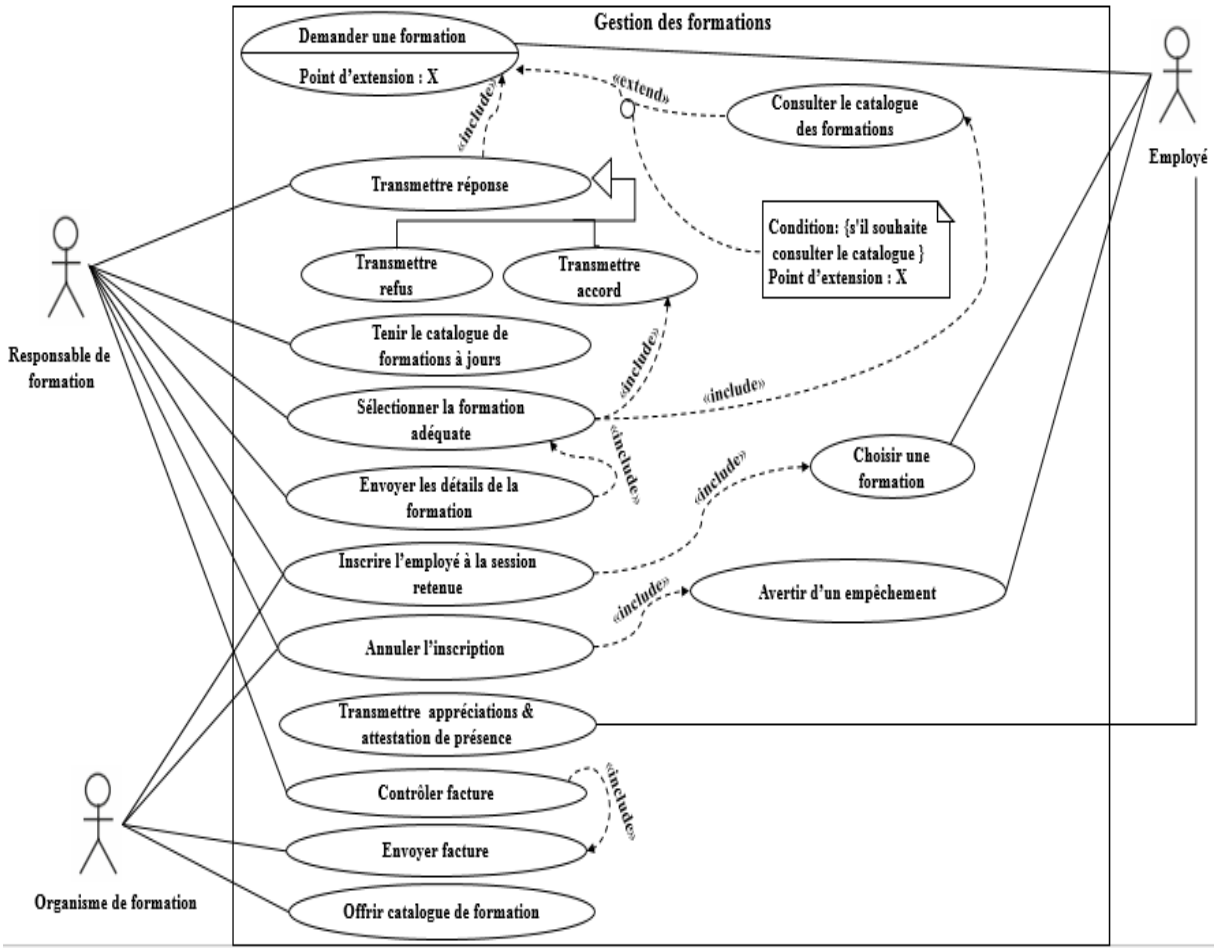
Corrigé de l'exercice 2



Corrigé de l'exercice 3



Corrigé de l'exercice 4



Chapitre 4 : Diagrammes UML de classes et d'objets (vue statique)

Objectif

À l'issue de ce chapitre, l'apprenant sera capable de :

- Connaître les principaux éléments de cette vue statique qui sont : les classes et leurs relations : association, multiplicité, rôles, agrégation, composition, héritage, etc ;
- Modéliser les classes du système et leurs relations indépendamment d'un langage de programmation particulier ;
- Comprendre comment représenter les instances des classes (objets).

1. Introduction

Le composant le plus important du développement logiciel orienté objet est la classe ou le modèle d'objet (vue statique, vue structurelle, vue statique, vue structurelle), qui est représenté graphiquement sous forme de diagramme de classes. Ce diagramme montre les classes et leurs relations entre elles. Dans la conception de bases de données, des parties essentielles de cette technologie ont été utilisées avec grand succès depuis 1976 (Peter Pin-Shan Chen) utilisé comme diagramme de relation d'entité. Le comportement d'un objet est décrit par des opérations de dénomination sans donner de détails sur le comportement dynamique.

2. Diagramme de classes

2.1. Définition

Le diagramme de classes est le plus important des diagrammes UML, c'est le seul qui soit obligatoire lors de la modélisation objet d'un système. Un diagramme de classes fournit une vue globale d'un système. De plus, il permet de modéliser les classes du système et leurs relations.

Les diagrammes de classes sont statiques : ils affichent ce qui interagit mais pas ce qui se passe pendant l'interaction.

Remarque

Les diagrammes de classes sont les diagrammes les plus répandus dans les systèmes de modélisation orientés objets.

2.2 Représentations des classes

Une classe est représentée par un rectangle divisé en trois (3) compartiments.

1. Le premier compartiment contient le nom de la classe qui :

- Débute par une lettre majuscule ;
- Il est centré dans le compartiment supérieur de la classe ;
- Il est écrit en caractère gras.

2. Le deuxième compartiment contient les attributs ;

3. Le troisième compartiment contient les méthodes.

La syntaxe d'un attribut d'une classe est comme suit :

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

- ***Visibilité*** : les symboles + (public), # (protégé), - (privé) et ~ (paquetage) sont indiqués devant chaque attribut pour signifier le type de visibilité.

- ***Nom d'attribut*** : nom donné à l'attribut.

- ***Type*** : type de l'attribut.

- ***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.

Seul le nom d'attribut est requis, sinon, ce qui est entre crochets désigne des clauses optionnelles.

Exemple :
- Nom : string [1] = « sansnom »
Dans cet exemple, nom est un attribut privé de type chaîne de caractères devant avoir une et une seule valeur. C'est-à-dire qu'un objet ne peut avoir qu'un seul nom. À la création d'un objet, si on ne spécifie pas de valeur pour nom, on lui attribue la valeur par défaut « sansnom ».
La syntaxe d'une opération d'une classe est la suivante :

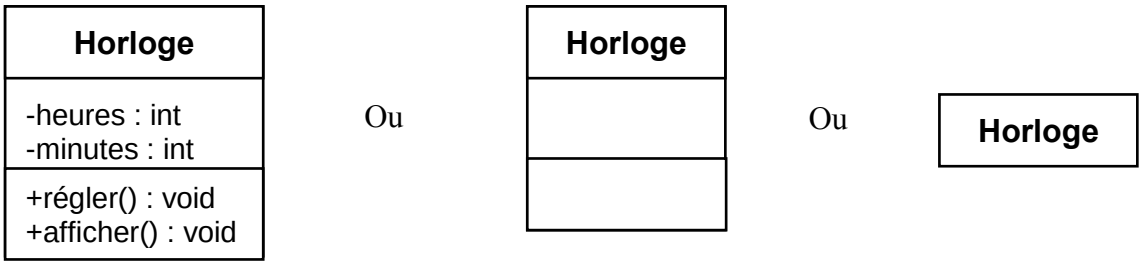
[Visibilité] nom ([liste de paramètres]) [: type]

Où,

- **Visibilité** : les symboles + (public), # (protégé), - (privé) et ~ (paquetage) sont indiqués devant chaque opération pour signifier le type de visibilité.
- **Nom d'opération** : 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.

Seul le nom de l'opération et () sont requis, sinon, ce qui est entre crochets désigne des clauses optionnelles.

Exemple :



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

Remarque

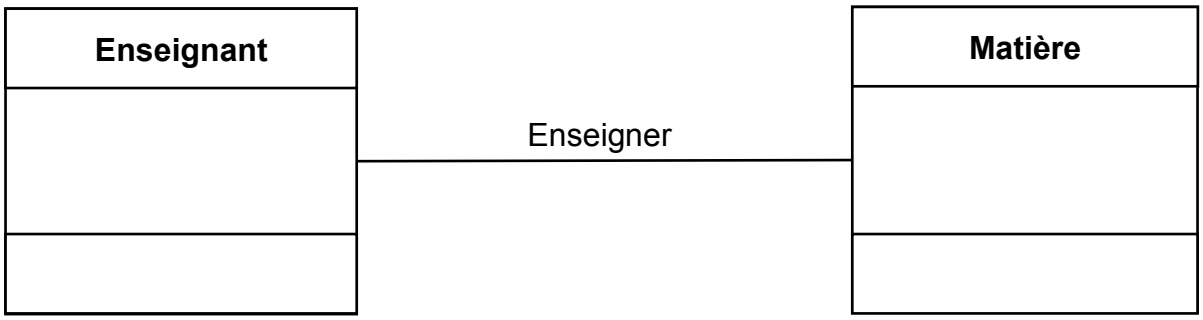
Une instance est un objet créé à partir d'une classe. La création d'un objet (instance) à partir d'une classe s'appelle l'instanciation.

2.3 Association entre classes

Association

L'association représente une relation entre plusieurs classes. Elle correspond à l'abstraction 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 sur le lien d'association.

Exemple :



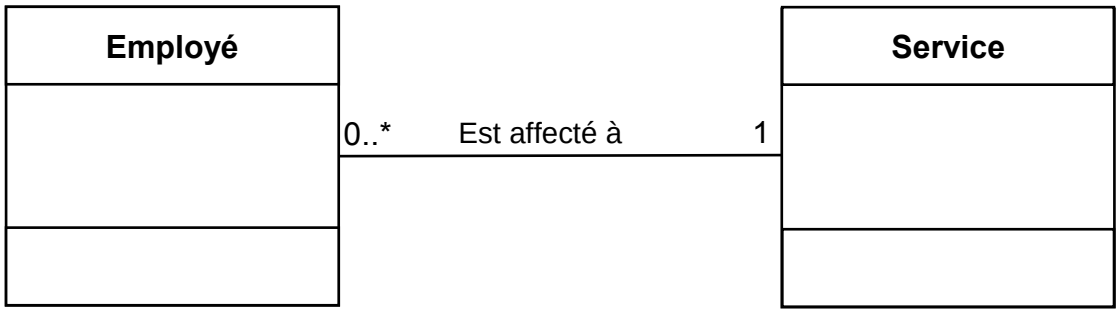
Multiplicité

La multiplicité est définie par un ensemble non vide d'entiers positifs à l'exclusion d'un ensemble ne contenant que zéro. Elle apparaît à chaque extrémité d'une relation et indique le nombre d'objets de la classe apparaissant à cette extrémité pouvant s'associer à un seul et unique objet de la classe apparaissant à l'autre extrémité.

Multiplicités	Signification
1	Une et une seule instance
0..1	Zéro ou une instance
*	De zéro à un nombre indéterminé d'instances
0..*	De zéro à un nombre indéterminé d'instances
1..*	De un à un nombre indéterminé d'instances
x	Exactement x instances
x..y	De x instances à y instances

Table 4.1 : Exemples de principales multiplicités définies dans UML.

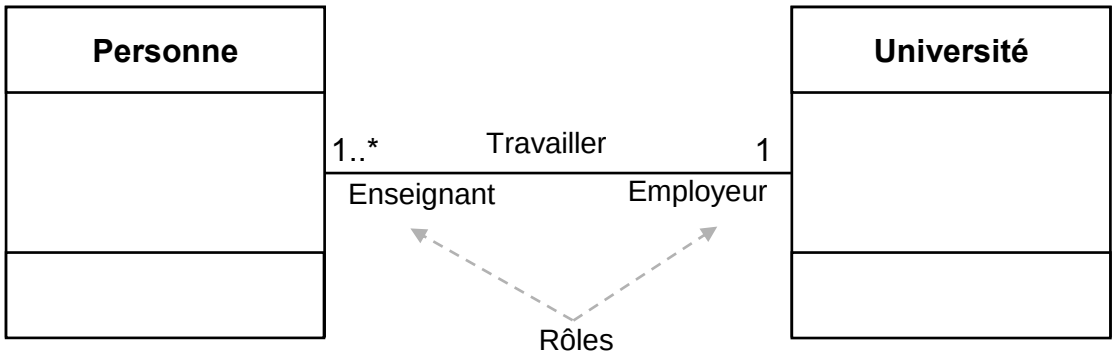
Exemple :



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.

Les rôles

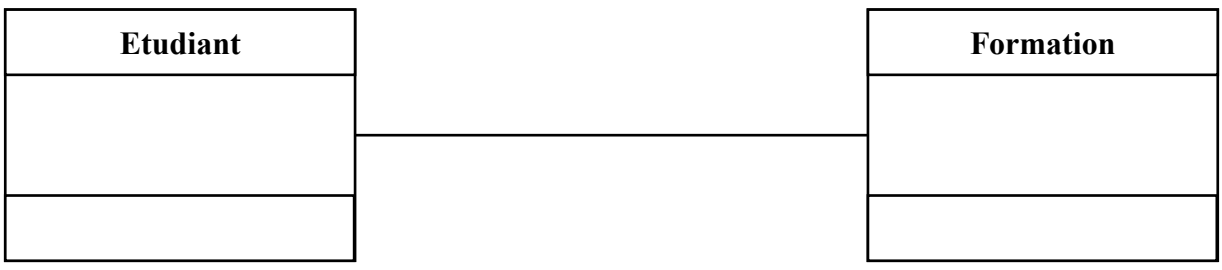
Il est possible de préciser le rôle d'une classe au sein d'une association. Le rôle est placé à une extrémité du lien d'association, il se distingue ainsi du nom de l'association situé au centre du lien. Le rôle spécifie la fonction d'une classe pour une association donnée.



2.3.1. Association binaire

L'association binaire : est l'association qui relie deux classes. Elle est donc de dimension de deux (2).

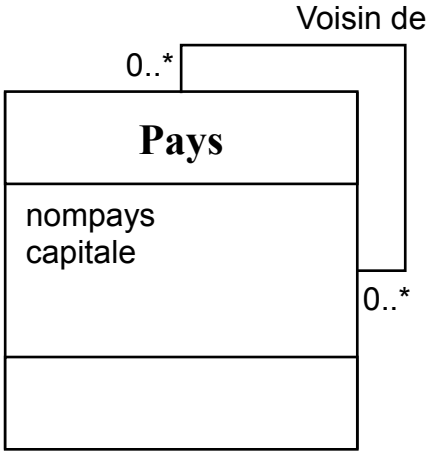
Exemple :



2.3.2 Association réflexive

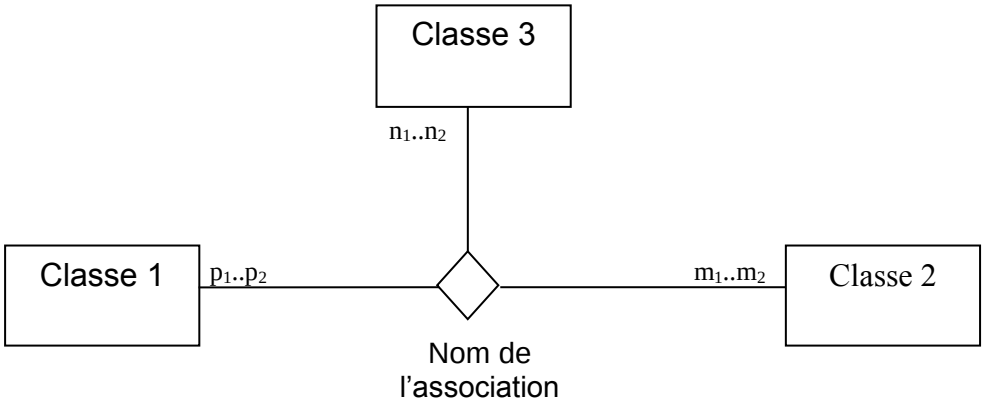
Lorsque le lien existe entre des objets de la même classe on parle d'association réflexive (unaire). Autrement dit, une association qui lie une classe avec elle-même est une association réflexive.

Exemple :



2.3.3 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.

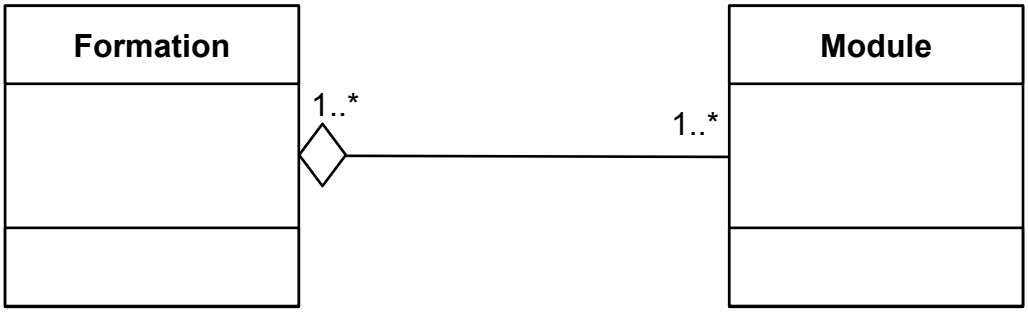


Pour une association ternaire, les multiplicités se lisent de la façon suivante : Pour un couple d'instances de la classe 1 et de la classe 2, il y a au minimum n_1 instances de la classe 3 et au maximum n_2.

2.4 Associations particulières

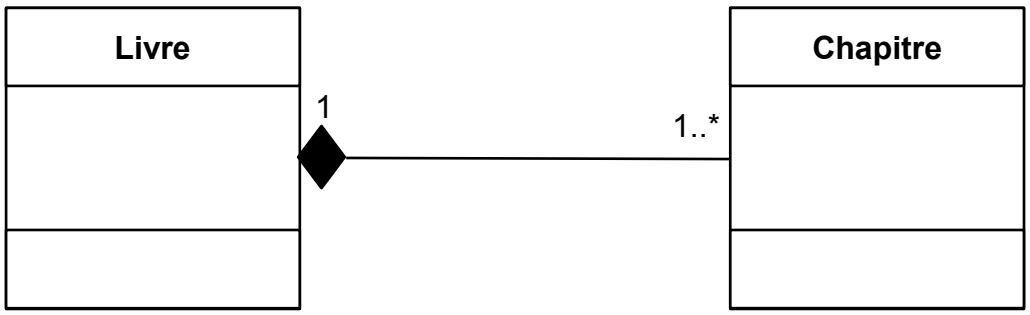
2.4.1. Agrégation

L'agrégation est une association qui permet de représenter un lien de type «ensemble» comprenant des «éléments». Il s'agit d'une relation entre une classe représentant le niveau «ensemble» et 1 à n classes de niveau «éléments». L'agrégation représente un lien structurel entre une classe et une ou plusieurs autres classes. L'agrégation se note par un trait entre deux classes orné d'un losange blanc du côté de la classe agrégat.



2.4.2. Composition

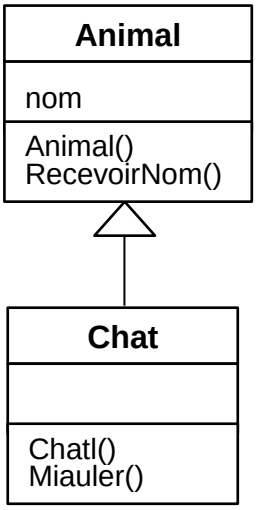
La composition est une relation d'agrégation dans laquelle il existe une contrainte de durée de vie entre la classe «composé» et la ou les classes «composant». Autrement dit, la suppression de la classe «composé» implique la suppression de la ou des classes «composant». La composition se représente par un trait plein orné d'un losange noir du côté de la classe « composé ».



2.5. Héritage

L'héritage est l'association entre deux classes permettant d'exprimer que l'une est plus générale que l'autre. L'héritage implique une transmission automatique de propriétés, à savoir les attributs et méthodes d'une classe X à une classe X'. La relation d'héritage entre deux classes est symbolisée par une flèche triangulaire creuse.

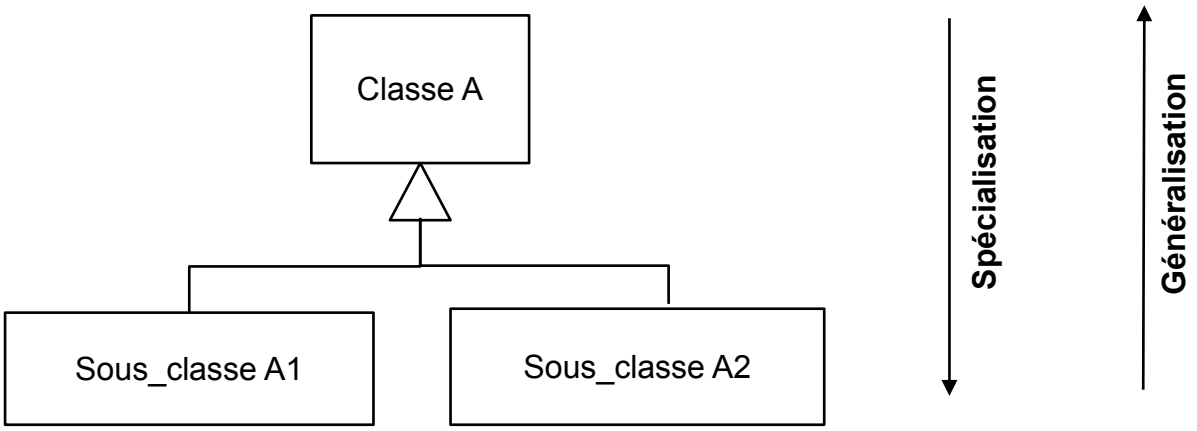
Exemple :



Cet exemple représente la classe Chat qui hérite de la classe Animal.

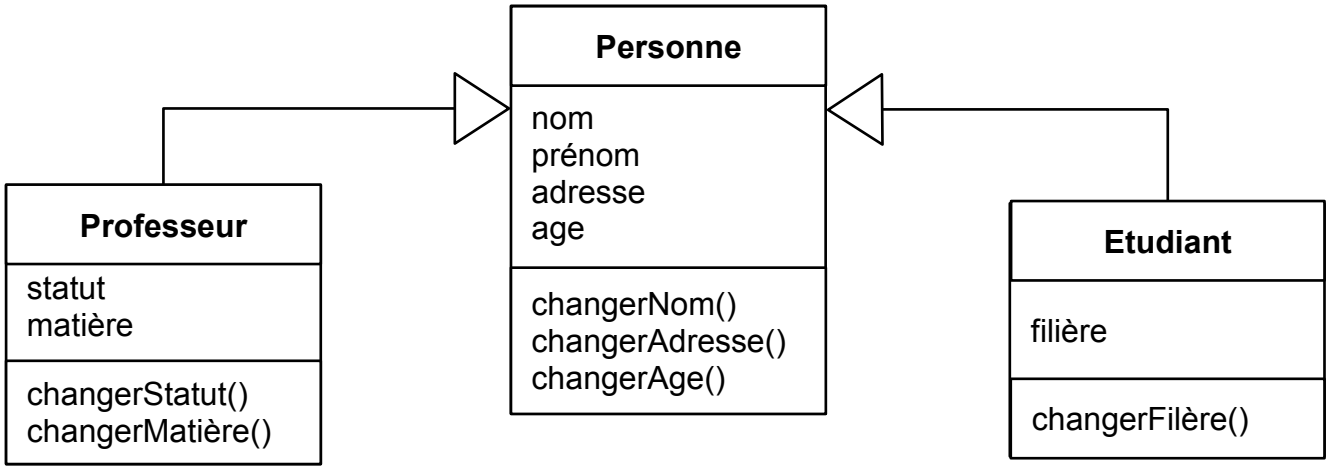
L'héritage est mis en œuvre grâce à deux mécanismes : Généralisation et spécialisation. L'opération qui consiste à créer une super classe à partir de classes s'appelle la généralisation. Inversement la spécialisation consiste à créer des sous classes à partir d'une classe.

Notation graphique



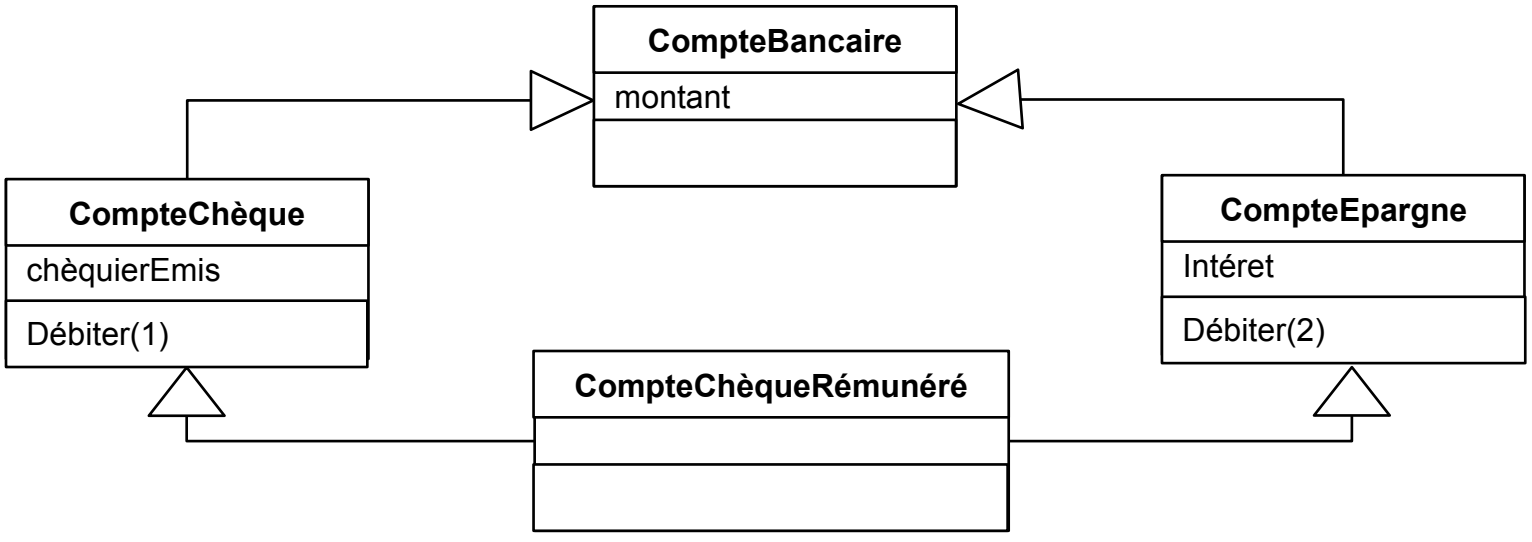
On distingue deux types d'héritage :

Héritage simple : L'héritage est simple lorsqu'une classe hérite d'une seule super_classe.



Les classes «Professeur» et «Etudiant» héritent de la classe «Personne».

Héritage multiple : L'héritage est multiple quand il y a deux ou plusieurs super_classes pour une même sous_classe.



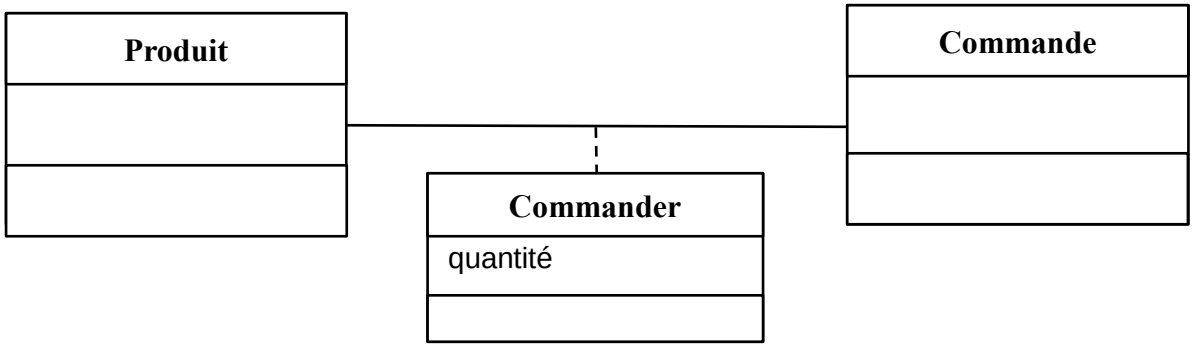
La classe «CompteChèqueRémunéré» hérite à la fois de la classe «CompteChèque» et de la classe «CompteEpargne».

2.6 Concepts avancés

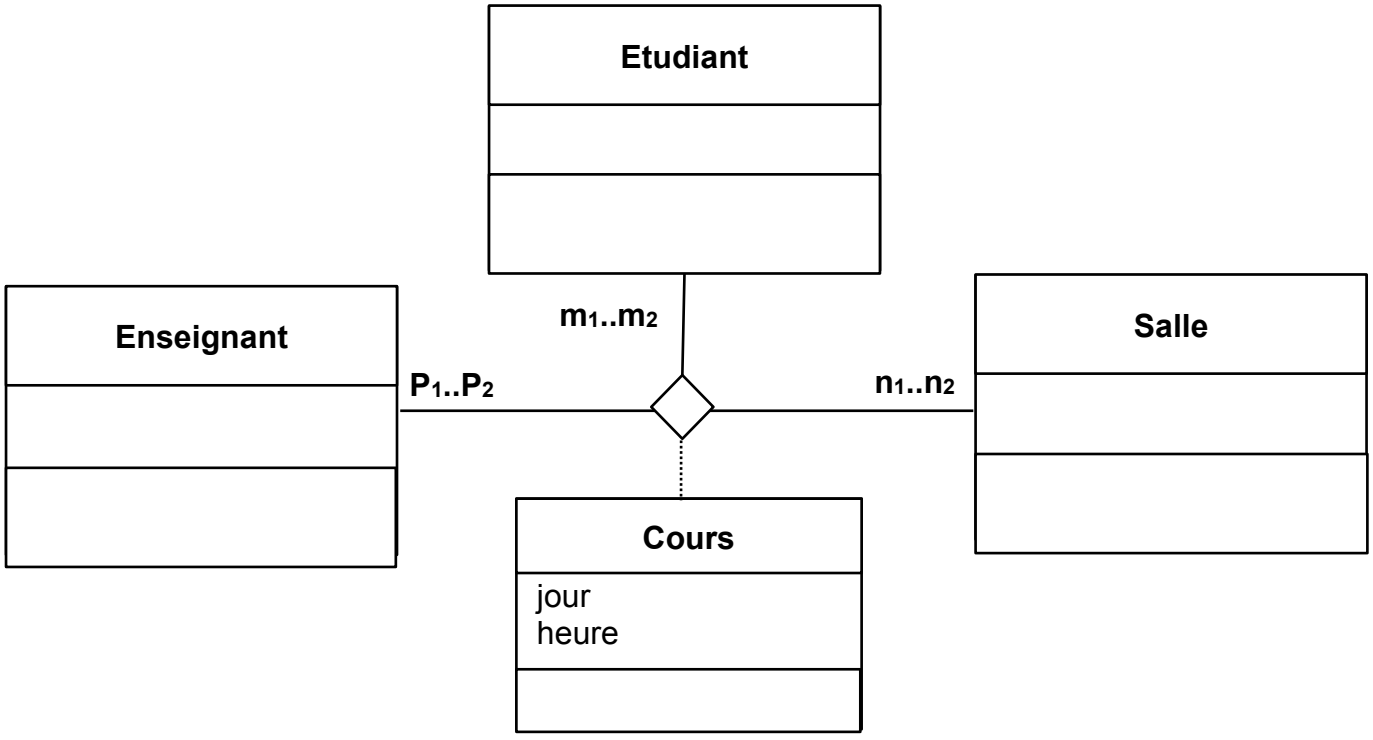
2.6.1 Classe association

Une classe-association permet de décrire soit des attributs soit des opérations propres à l'association. Cette classe-association est elle-même reliée par un trait en pointillé au lien d'association.

Exemple :
Cet exemple représente une classe-association dans le cas association binaire.



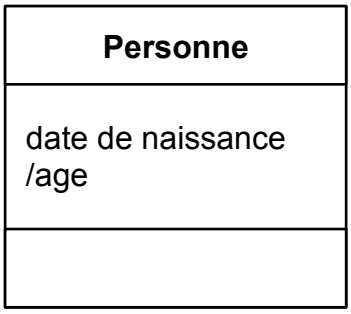
Exemple :
Cet exemple représente une classe-association dans le cas association ternaire.



2.6.2 Associations et attributs dérivés

On parle d'attribut (ou d'association) dérivé(e) lorsque l'attribut ou l'association en question découle (ou dérive) d'autres attributs de la classe ou de ses sous-classes. On les symbolise par le préfixe « / » :

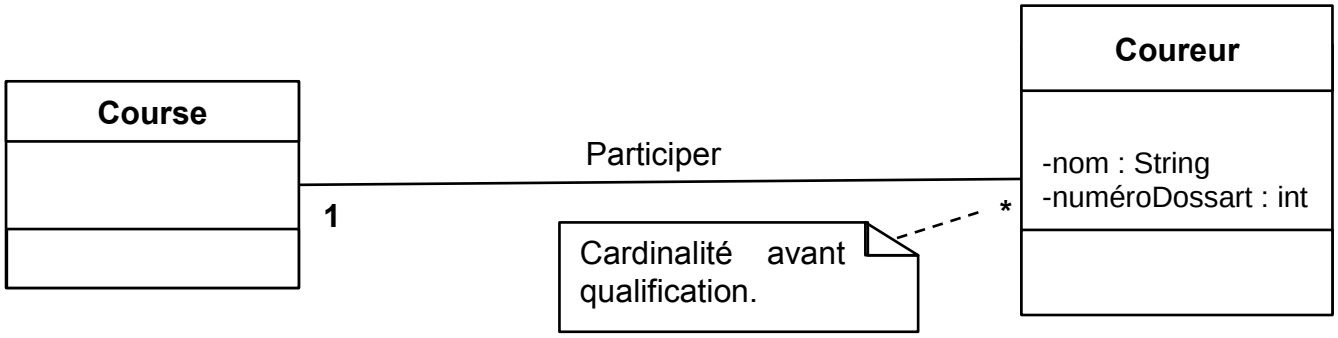
Exemple :



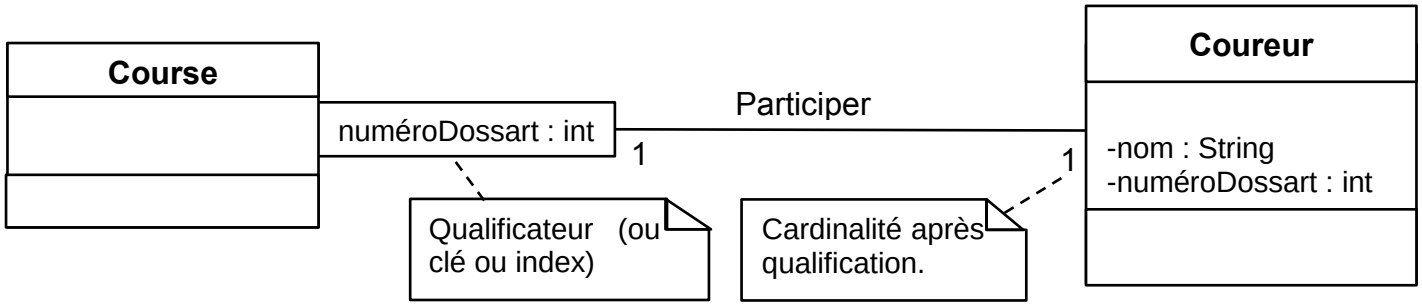
2.6.3. Association qualifiée (Qualification)

Une association qualifiée permet de restreindre la multiplicité d'une association en ajoutant un qualificateur (aussi appelé clé ou index). Ce qualificateur est constitué de un ou plusieurs attributs qui permettent de cibler un ou plusieurs objets en particulier. Le qualificateur est placé dans un rectangle à l'extrémité de l'association (extrémité opposée à la classe dont nous limitons la multiplicité).

Exemple :
Un objet de la classe Course est relié à un nombre indéterminé d'objet de la classe Coureur.



Par contre, la classe Course associée au qualificateur numéroDossart n'est reliée qu'à un seul objet de la classe coureur

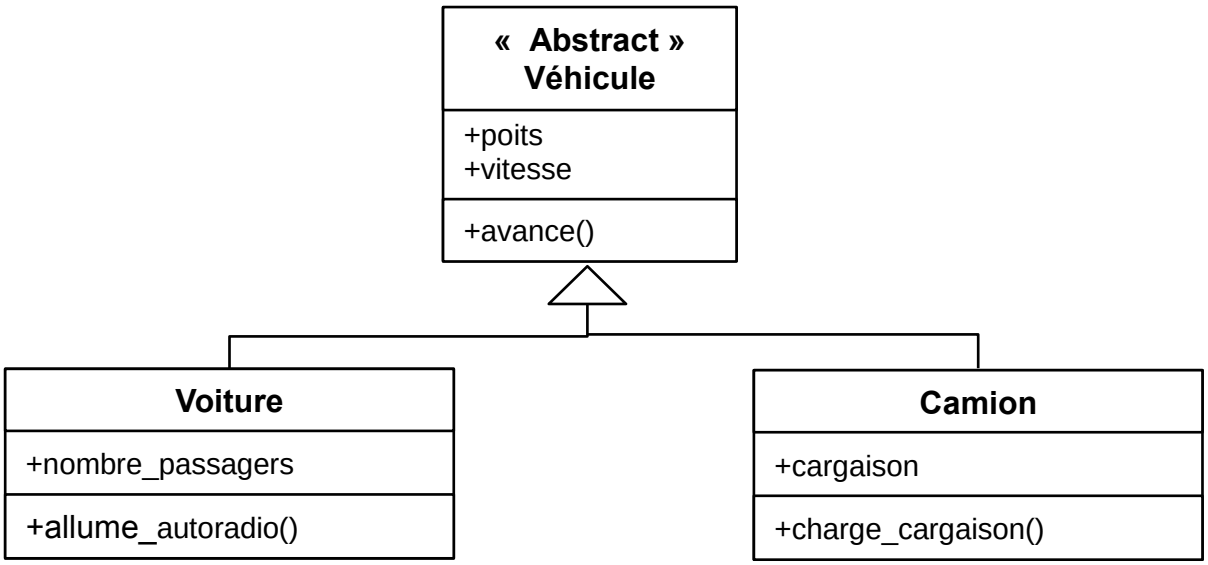


2.6.4. Classe abstraite

Une classe abstraite est une classe qui n'a pas d'instance directe, mais dont les classes descendantes ont des instances. Dans une relation d'héritage, la super-classe est par définition une classe abstraite.

Exemple :

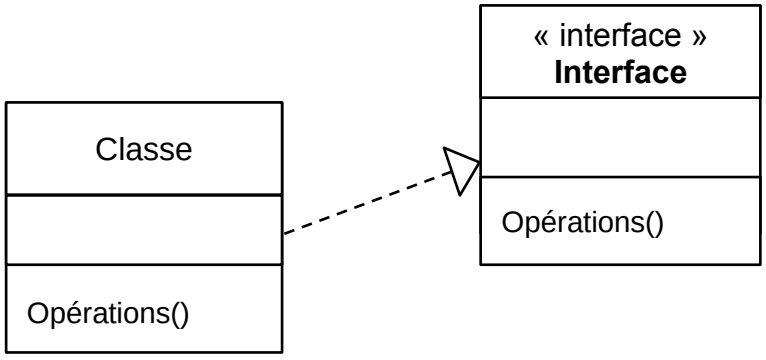
L'autre représentation du nom d'une classe abstraite est le faite de le mettre en italique sans avoir recoure au stéréotype « abstract ».



1.6.5. Interfaces

Le but d'une interface est de regrouper un ensemble d'opérations assurant un service cohérent offert par un classeur et une classe en particulier. Une interface est représentée comme une classe, avec les mêmes compartiments, mais on ajoute le stéréotype " interface " avant le nom de l'interface. Une relation de type réalisation (flèche d'héritage en pointillés) est utilisée entre une interface et une classe qui l'implémente.

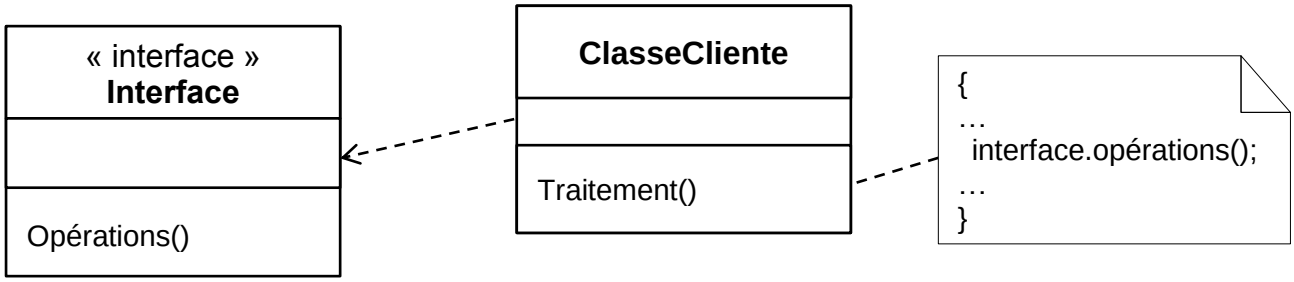
Notation graphique



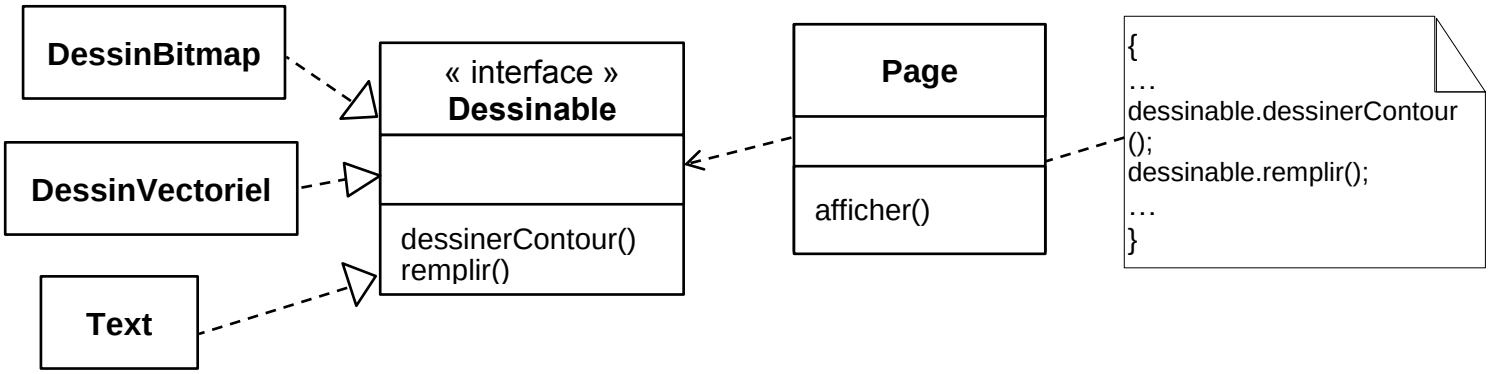
Les classes implémentant une interface doivent implémenter toutes les opérations décrites dans l'interface.

Lorsqu'une classe dépend d'une interface (interface requise) pour réaliser ses opérations, elle est dite " classe cliente de l'interface ". Une classe réalise une interface si elle est capable d'exécuter toutes les opérations de l'interface. Pour exprimer le fait qu'une classe est cliente d'une interface nous utiliserons une relation de dépendance entre la classe cliente et l'interface requise. Toute classe implémentant l'interface pourra être utilisée. Enfin, une interface ne spécifie souvent qu'une partie limitée du comportement de la classe.

Notation graphique



Exemple :

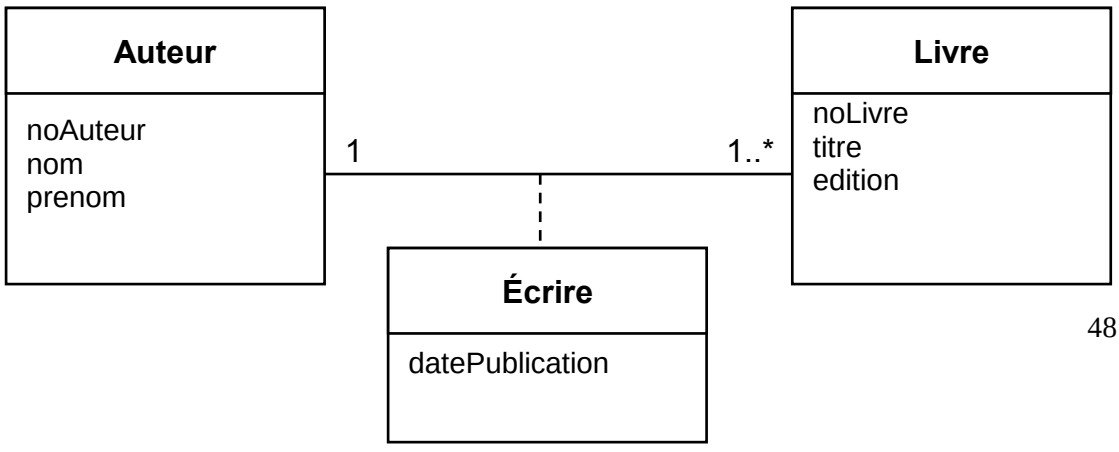


3. Modèle de domaine

En génie logiciel, un modèle de domaine est le modèle conceptuel d'un domaine. En langage UML, on utilise un diagramme de classes pour représenter le modèle de domaine.

Règle1 : présence de la multiplicité « * » sur un côté de l'association

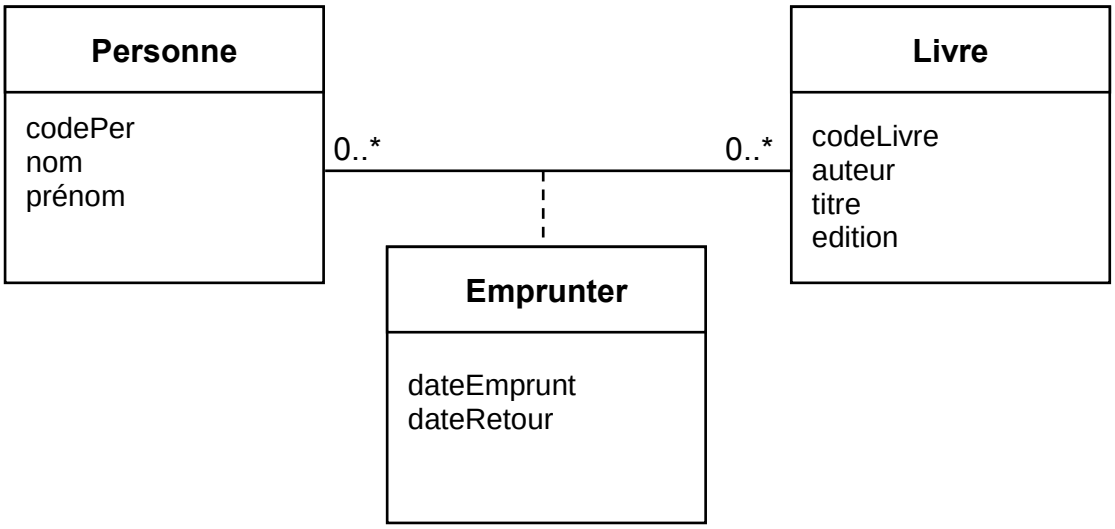
- Chaque classe se transforme en une relation.
- Chaque attribut de classe se transforme en un champ de relation.
- L'identifiant de la classe qui est associée à la multiplicité « 1 » devient une clé étrangère dans l'autre classe.



Auteur (noAuteur, nom, prenom)
Livre (noLivre, titre, edition, #noAuteur, datePublication)

Règle 2 : présence de la multiplicité « * » sur les deux côtés

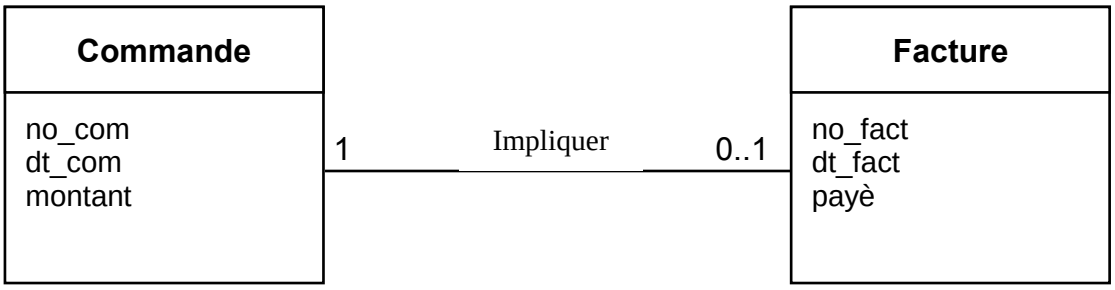
- Chaque classe se transforme en une relation.
- Chaque attribut de classe se transforme en un champ de relation.
- L'association se transforme en une relation qui aura comme champs l'identifiant de chacune des deux classes (plus d'éventuels autres attributs).



Personne (codePer, nom, prénom)
Livre (codeLivre, auteur, titre, edition)
Emprunter (#codePer,#codeLivre, dateEmprunt, dateRetour)

Règle 3 : présence de la multiplicité « 1..1 » « 0..1 » sur les côtés

- Chaque classe se transforme en une relation.
- Chaque attribut de classe se transforme en un champ de relation.
- L'identifiant de la classe qui est associée à la multiplicité « 1..1 » devient clé étrangère dans l'autre classe (celle qui est associée à la multiplicité « 0..1 »).



Commande (no_com, dt_com, montant)
Facture (no_fact, #no_com, dt_fact, payé)

Règle 4 : présence d'une relation de composition

- Chaque classe se transforme en une relation.

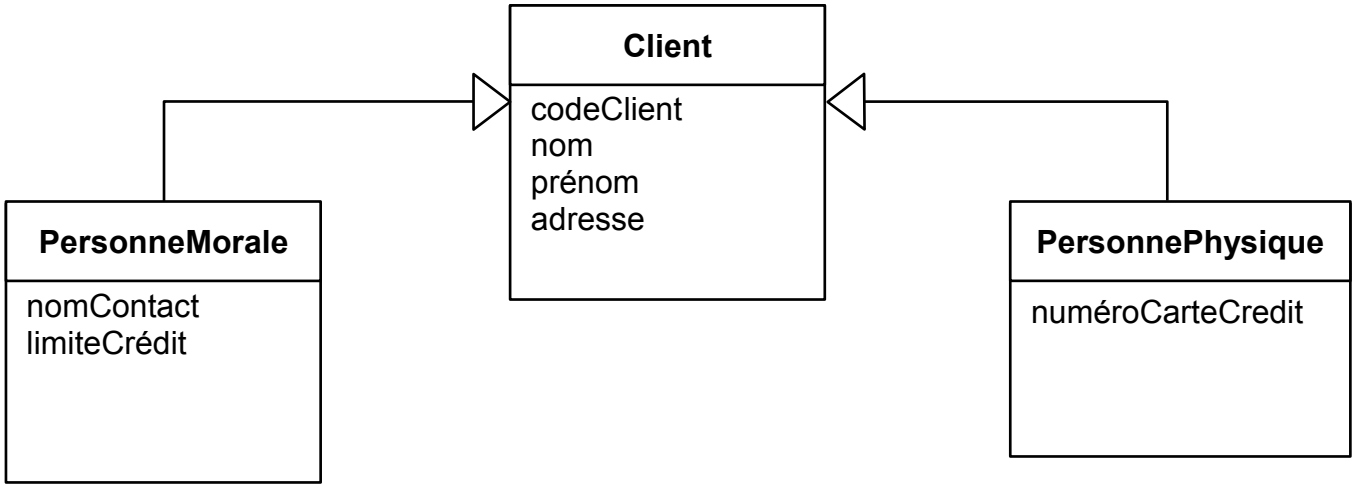
- Chaque attribut de classe se transforme en un champ de relation.
- L'identifiant de la classe qui est associée à la multiplicité « 1 » (classe qui représente le niveau ensemble) devient à la fois une clé étrangère et partie de la clé primaire dans l'autre classe.

Groupe (id_groupe, nom_groupe)
Etudiant (id_étudiant, #Id_groupe, nom, prénom, date_naissance, adresse)

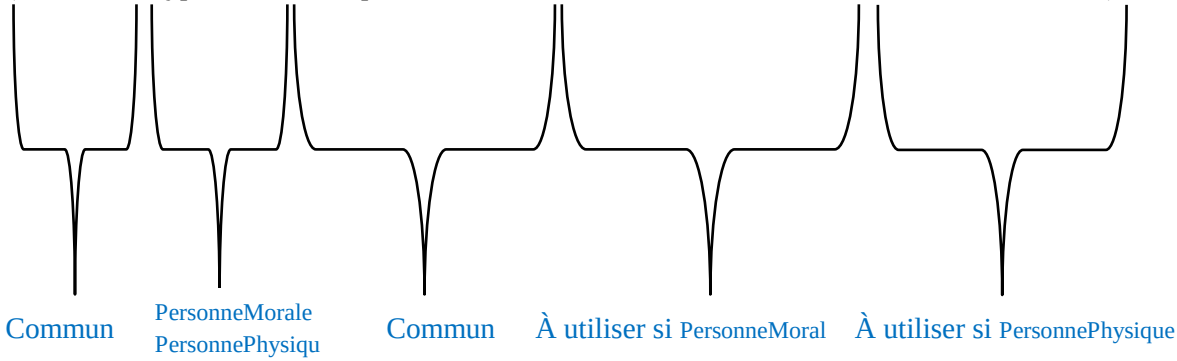
Règle 5 : présence d'une généralisation-spécialisation

Méthode 1 : push-up

- Créer une relation avec tous les attributs des classes.
- Ajouter un attribut pour distinguer les types des objets.

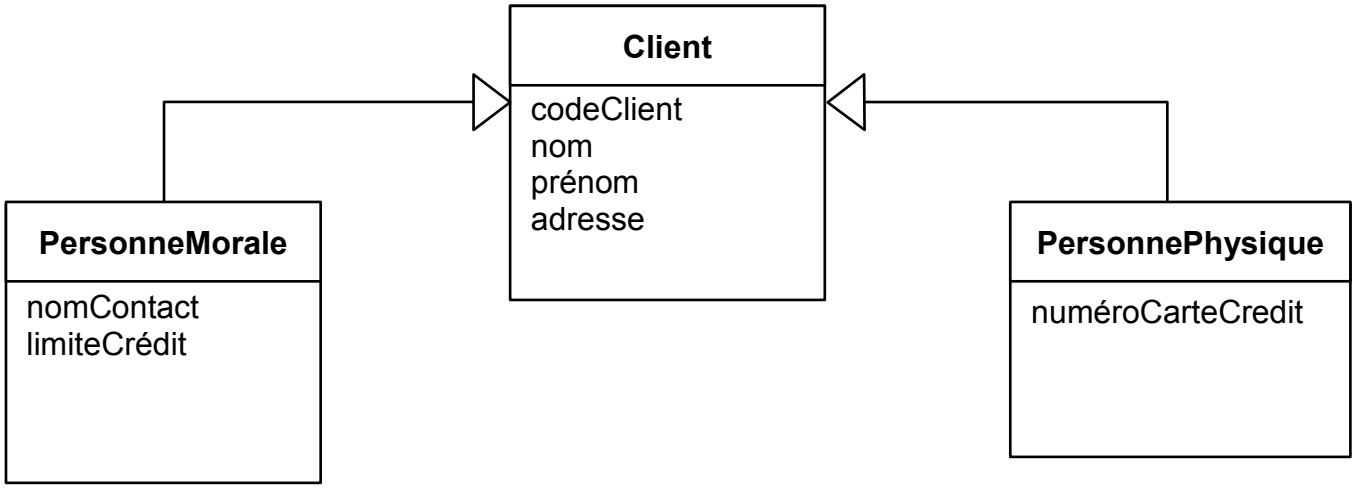


Client (codeClient, typeClient, nom, prénom, adresse, nomContact, limiteCrédit, numéroCarteCrédit)



Méthode 2 : push-down

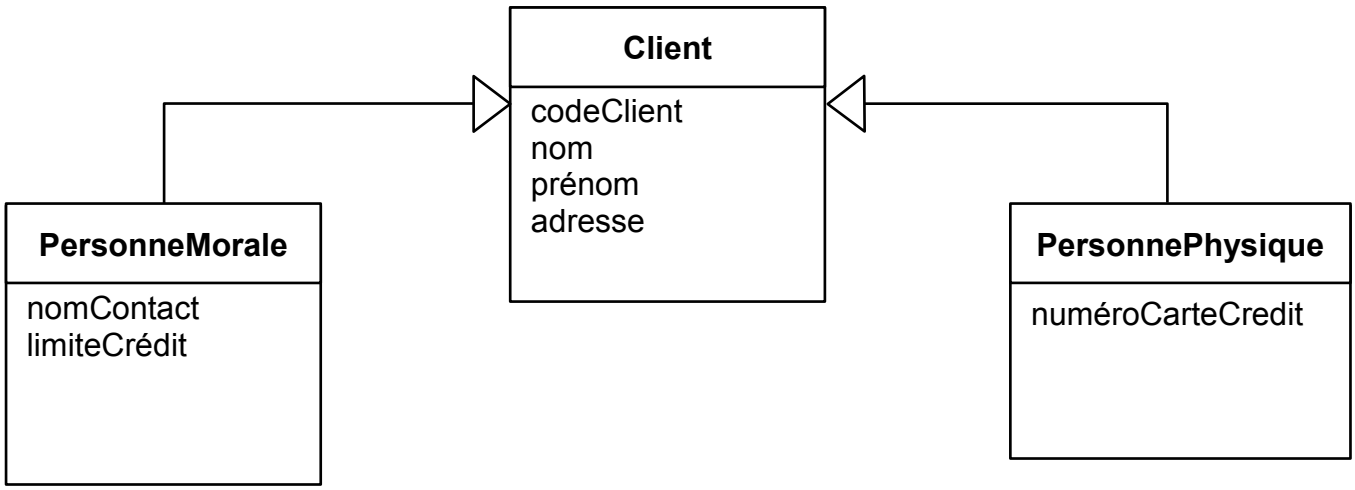
- Créer une relation pour chaque sous type.
- Chaque relation se compose des attributs génériques et des attributs spécifiques.



PersonneMorale (codeClient, nom, prénom, adresse, nomContact, limiteCrédit)
PersonnePhysique (codeClient, nom, prénom, adresse, numéroCarteCredit)

Méthode 3 : distinction

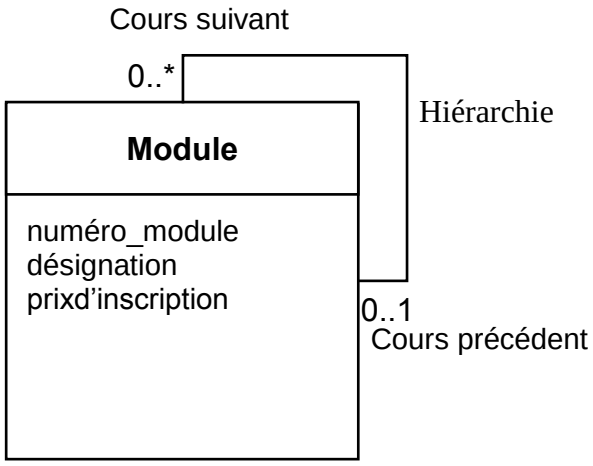
- Créer une relation par classe et les relier par des associations



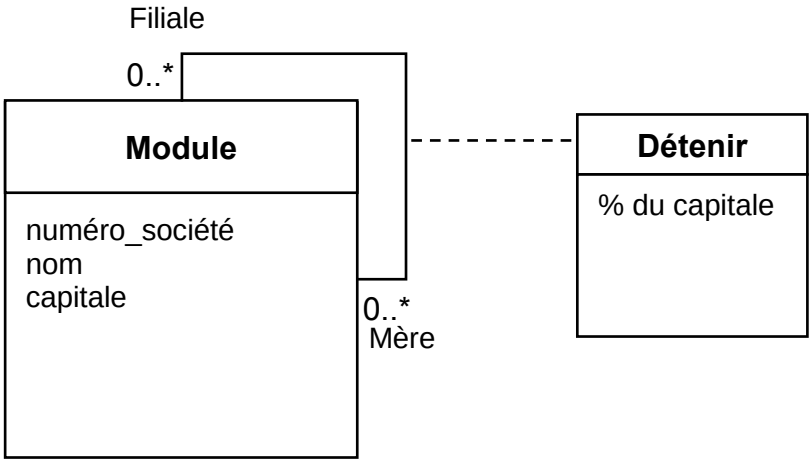
Client (codeClient, nom, prénom, adresse)
PersonneMorale (#codeClient, nomContact, limiteCrédit)
PersonnePhysique (#codeClient, numéroCarteCredit)

Règle 6 : présence d'une relation réflexive

Nous appliquons les règles générales avec la seule différence que la relation est 2 fois reliée à la même entité.



Module (<u>numéro_module</u>, désignation, prixd'inscription, #numéro_module_Cours précédent)



Societé (<u>numérosociété</u>, nom, capitale)

Détenir (<u>#numéro_société, #numéro_société_filiale</u>, % du capitale)

4. Diagramme d'objets

4.1. Définition

Le diagramme d'objets, dans le langage de modélisation UML, permet de représenter les instances des classes (objets) et leurs liens à un instant donné. Le diagramme d'objets fait parties des diagrammes structurels (statique), il donne une vue figée du système à un moment précis. À un diagramme de classe correspond une infinité de diagrammes d'objets.

Remarque

Les diagrammes d'objets s'utilisent pour montrer l'état des instances d'objet avant et après une interaction, autrement dit c'est une photographie à un instant précis des attributs et objet existant. Il est utilisé en phase exploratoire.

4.2. Représentation graphique

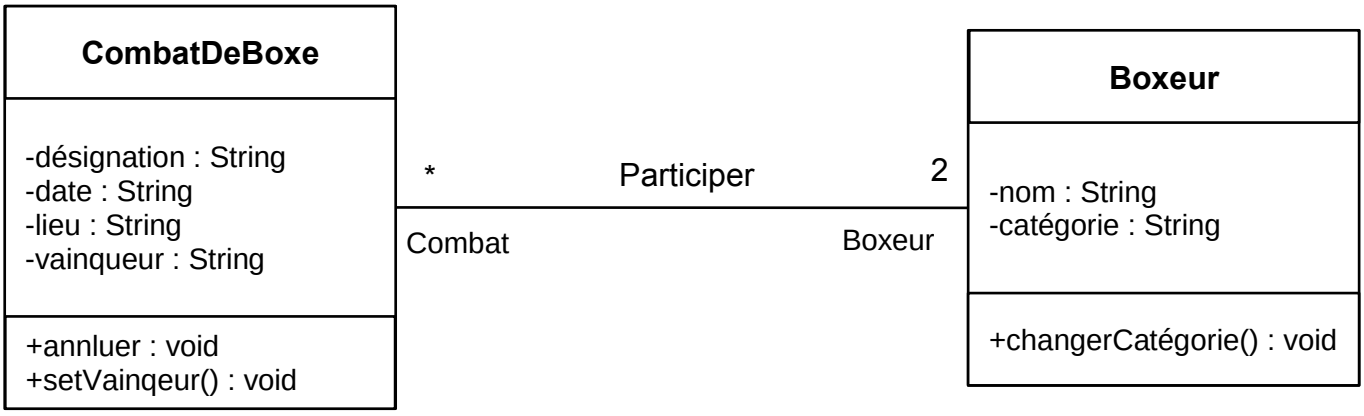
Chaque objet est représenté dans un rectangle dans lequel figure le nom de l'objet (souligné) et éventuellement la valeur d'un ou plusieurs de ses attributs. Comme pour la représentation d'une classe, la représentation d'un objet pourra être plus ou moins détaillée.

Représentation graphique	Signification
: Voiture	Instance anonyme de la classe ***Voiture***
voitureDeMathèo : Voiture	Instance nommée de la classe ***Voiture***
voitureDeMathèo	Instance nommée d'une classe anonyme
voitureDeMathèo : Voiture marque=volkswagen modèle= golf couleur=noire	Spécification des attributs
: Voiture	Collection d'instances anonymes (tableau)

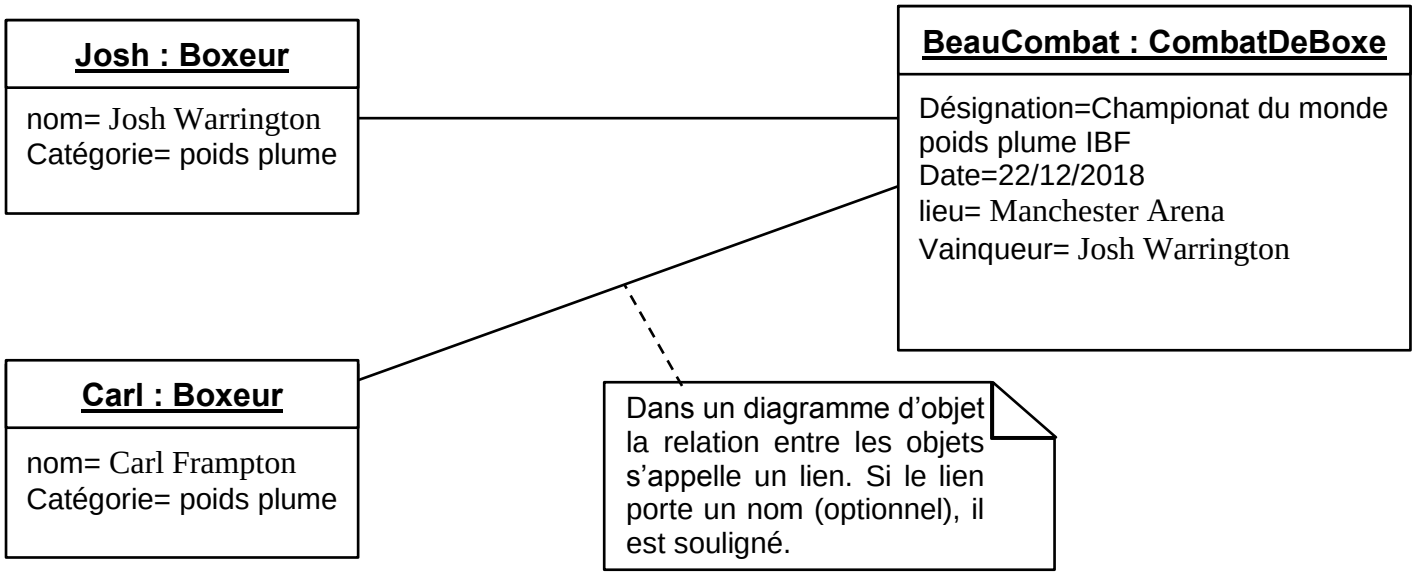
Table 4.2 : Représentation graphique d'un objet.

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

Exemple :



Avec le diagramme d'objet ci-dessous on définit une instance particulière du diagramme de classe. Cette instance est le combat entre Josh Warrington et Carl Frampton qui a eu lieu le samedi 22 décembre 2018 à Manchester Arena (Manchester, Royaume-Uni).



5. Conclusion

Les diagrammes de classes sont les diagrammes UML les plus populaires dans la société orientée objet. Il décrit les objets d'un système et leurs relations. Fondamentalement, le diagramme de classes représente une vue statique du système. Les diagrammes de classes sont les seuls diagrammes UML qui peuvent être directement mappés aux langages orientés objet. Par conséquent, il est largement utilisé dans la communauté des développeurs. Le diagramme d'objets est une instance du diagramme de classes. Par conséquent, les éléments de base d'une classe de graphes sont similaires. Le diagramme d'objets est composé d'objets et de liens. À un moment précis, il capture une instance du système. Nous abordons dans le chapitre suivant les diagrammes dynamiques d'UML, qui montrent une vue dynamique du logiciel.

6. Exercices

Exercice 1 :

Comparer les diagrammes de classes et les diagrammes objets : à quoi servent-ils ? Quel rapport pouvons-nous établir entre eux ?

Exercice 2 :

Il s'agit d'établir le schéma conceptuel d'une base de données pour la gestion des formations d'un institut privé :

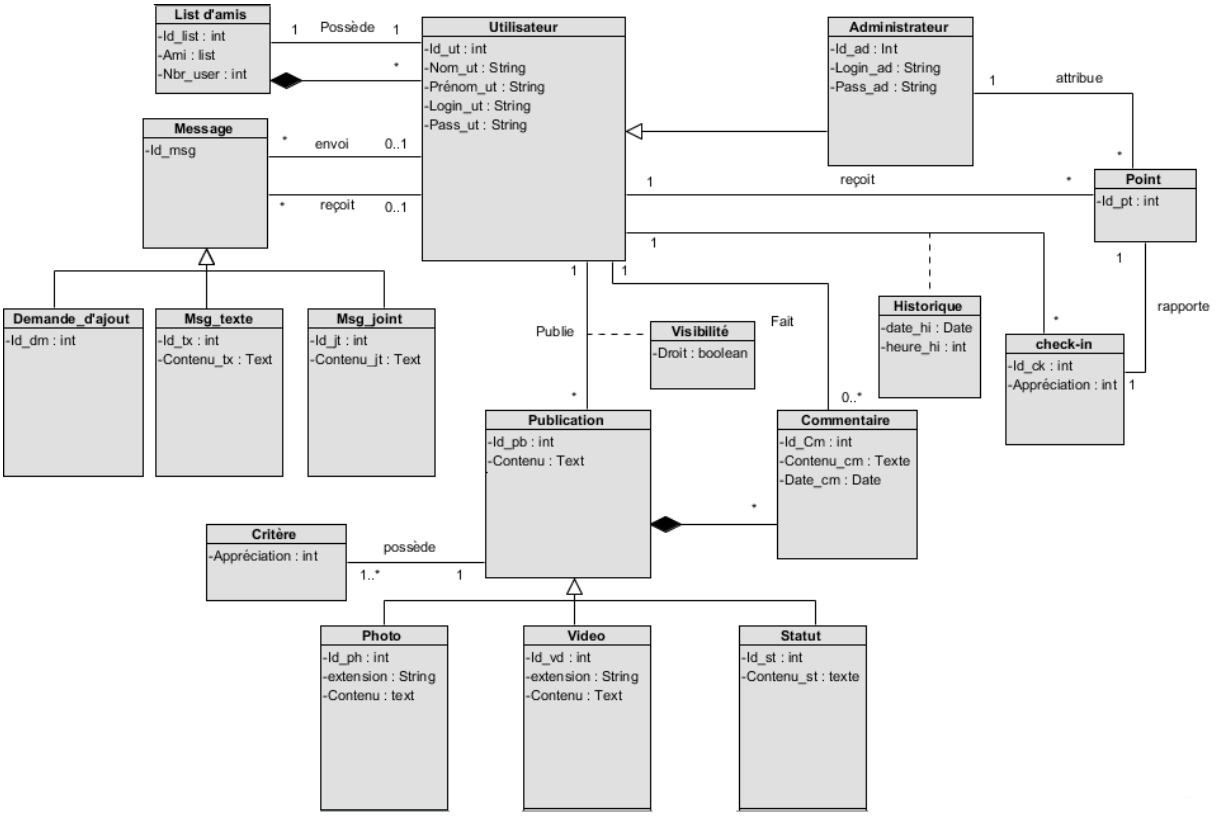
- Un cours est caractérisé par un numéro de cours (noCours), un libellé (libelle), une durée en heures (duree) et un type (type).
- Un cours peut faire l'objet dans l'année de plusieurs sessions identiques.
- Une session est caractérisée par un numéro (noSes), une date de début (date) et un prix (prix).
- Une session est le plus souvent assurée par plusieurs animateurs et est placée sous la responsabilité d'un animateur principal.
- Un animateur peut intervenir dans plusieurs sessions au cours de l'année.

- On désire mémoriser le nombre d'heures (nbh) effectué par un animateur pour chaque session.
- Un animateur est caractérisé par un numéro (noAni), un nom (nomA) et une adresse (adrA).
- Chaque session est suivie par un certain nombre de participants.
- Un participant est une personne indépendante ou un employé d'une entreprise cliente.
- Un participant est caractérisé par un numéro (noPar), un nom (nomP) et une adresse (adrP).
- Dans le cas d'un employé, on enregistre le nom (nomEn) et l'adresse de l'entreprise (adrEn).
- On désire pouvoir gérer d'une manière séparée (pour la facturation notamment) les personnes indépendantes d'une part, et les employés d'autre part. Si nécessaire, on fera les hypothèses sémantiques complémentaires qui pourraient s'imposer.

Travail demandé :
Elaborez le diagramme de classes UML correspondant.

Exercice 3 :

Établir le modèle relationnel correspondant au modèle de domaine présenté ci-dessous qui représente les concepts métiers d'une application de réseau social (utilisez la méthode ''**push-up**'' en cas de généralisation).



Exercice 4 :

Un robot se déplace dans un environnement composé de zones, de murs et de portes. Proposez le diagramme d'objets décrivant la situation suivante : le robot Mars est en mouvement. Il est lié à une instance mondeCourant de la classe Monde décrivant les mondes possibles où peut évoluer le robot. Le robot peut manipuler des objets se trouvant dans le monde dans lequel il évolue.

À l'instant qui nous intéresse, le robot Mars est en mouvement et mondeCourant est lié aux zones z1 et z2. La zone z2 est composée de deux murs (m1 et m2) et d'une porte. La largeur de la porte est de un mètre.

Travail demandé :

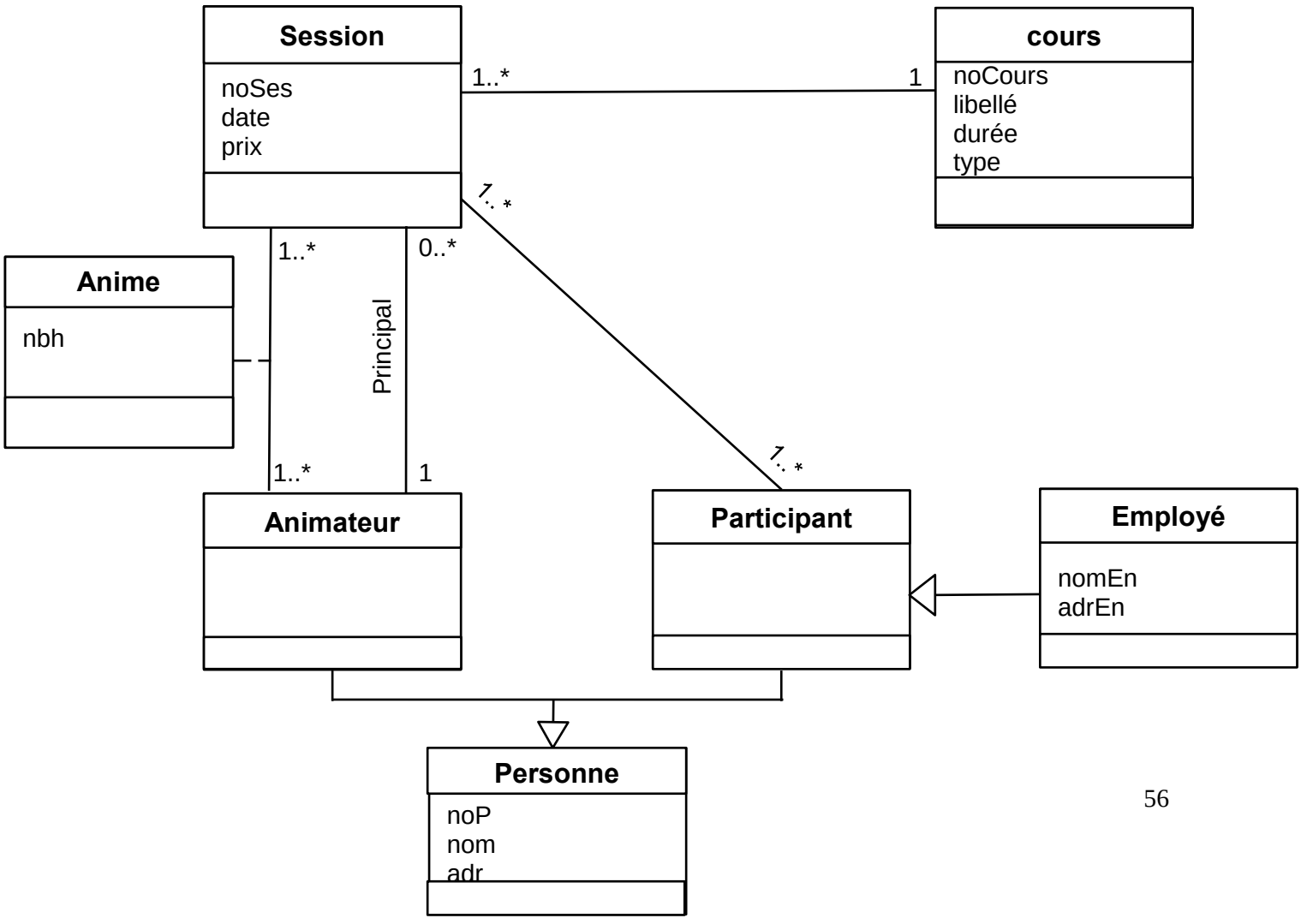
Donnez le diagramme d'objets associé à cette situation.

7. Corrigés des exercices

Corrigé de l'exercice 1

Le diagramme de classes sert à identifier les concepts propres au domaine modélisé. Il présente la structure statique du système modélisé. Le diagramme objets sert à illustrer un diagramme de classes. Il agit comme un snapshop du système en construction. Tous les éléments dans un diagramme objets sont donc des instances (objets, liens,…) des diagrammes de classe.

Corrigé de l'exercice 2

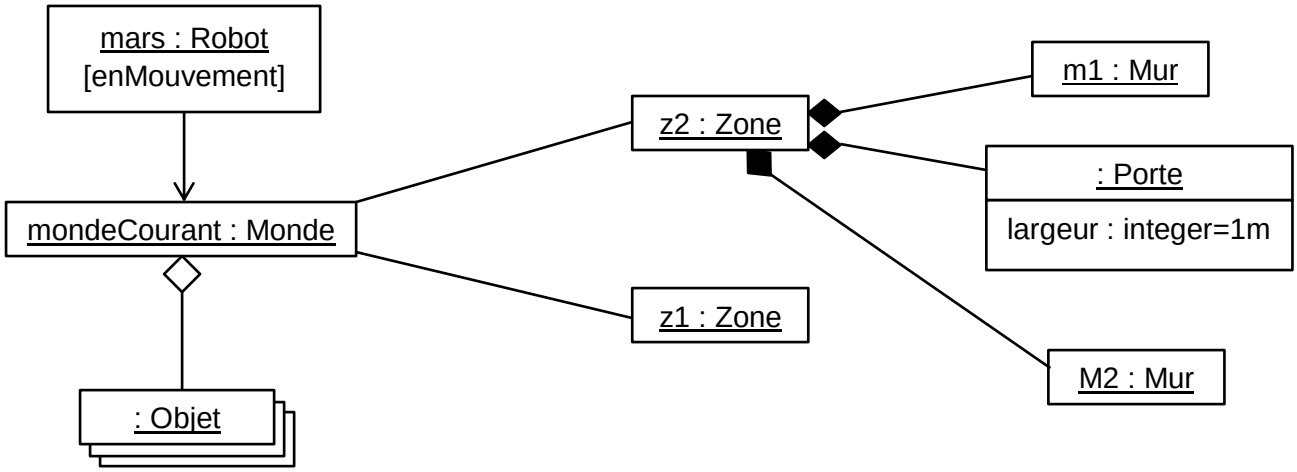


Corrigé de l'exercice 3

Publication (id_pb, type_pb, #id_ut, droit, extension_ph, contenu_ph, id_vd, , extension_vd, contenu_vd, id_st, contenu_st)
Critère (appréciation, #id_pb)
Commentaire (id_cm, #id_pb, contenu_cm, date_cm, #id_ut)
Utilisateur (id_ut, #id_list, type_user, nom_ut, prénom_ut, login_ut, pass_ut, id_ad, login_ad, pass_ad)
List d'amis (id_list, ami, nbr_user)
Message (id_msg, type_msg, id_dm, id_tx, contenu_tx, id_jt, contenu_jt, #id_ut_émetteur, #id_ut_récepteur)
Chek-in (id_ck, appréciation, #id_ut, date_hi, heure_hi)
Point (id_pt, id_ut, #id_ad)

Corrigé de l'exercice 4

Vous avez besoin des classes Robot, Monde, Objet, Zone, Mur et Porte. La classe Robot est active. Le diagramme d'objets présenté à la figure ci-après correspond à un snapshot d'une situation particulière et varie selon la position du robot.



Chapitre 5 : Diagrammes UML (vue dynamique)

Objectif

À l'issue de ce chapitre, l'apprenant sera capable de :

- Maîtriser les concepts de base d'un diagramme de séquence ;
- Apprendre à établir le diagramme de séquence ;
- Reconnaître un diagramme de communication.
- Maîtriser les concepts de base d'un diagramme d'états-transitions ;
- Apprendre à établir le diagramme d'états-transitions ;
- Maîtriser les concepts de base d'un diagramme d'activités ;
- Apprendre à établir le diagramme d'activités ;

1. Introduction

L'interaction d'objet pour implémenter un comportement peut-être décrite de deux façons complémentaires. La première est centrée sur des objets individuels (diagramme d'états-transitions) et la deuxième est centrée sur une collection d'objets qui coopèrent (diagrammes d'interaction). Les diagrammes d'interactions permettent d'établir un lien entre les diagrammes de cas d'utilisation et les diagrammes de classes. En d'autres termes, ils permettent de modéliser comment les objets communiquent entre eux pour réaliser une certaine fonctionnalité. En fait, ceci apporte un aspect dynamique à la modélisation du système. Enfin, l'élaboration d'un diagramme d'interaction se concentre sur un sous-ensemble d'éléments du système et l'étude de la manière dont ils interagissent pour décrire un comportement bien particulier.

2. Diagrammes d'interaction

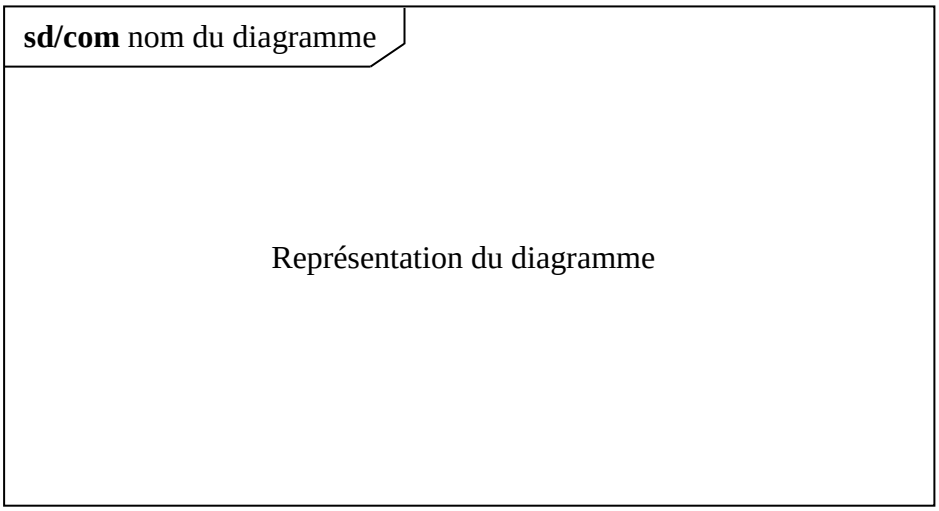
Le diagramme d'interaction décrit la relation de coopération dynamique entre les objets (instances de classes et/ou acteurs) et la séquence d'actions dans le processus de coopération. Les diagrammes d'interaction sont souvent utilisés pour décrire le comportement d'un cas d'utilisation, montrer les objets impliqués dans le cas d'utilisation et le message passant entre ces objets, c'est-à-dire le processus de réalisation d'un cas d'utilisation.

Il existe deux formes de diagrammes d'interaction, les diagrammes de séquence et les diagrammes de collaboration. Le diagramme de séquence décrit l'échange de messages d'objets dans l'ordre chronologique, et le diagramme de communication anciennement appelé diagramme de collaboration se concentre sur la description de la façon dont les composants du système fonctionnent ensemble.
Le diagramme de séquence et le diagramme de collaboration expriment l'interaction et le comportement du système dans le système sous différents angles, et ils peuvent être transformés l'un dans l'autre.

Le diagramme d'interaction est placé dans un grand rectangle qui dispose d'une étiquette en haut à gauche. L'étiquette est sous forme d'un pentagone accompagné du mot-clé (**sd** ou **com**) suivit du nom de l'interaction.

- **sd** lorsqu'il s'agit d'un diagramme de séquence.
- **com** lorsqu'il s'agit d'un diagramme de communication.

Le formalisme général du cadre d'un diagramme d'interaction est le suivant :



2.1 Diagrammes de séquence

Le diagramme de séquence est un diagramme d'interaction mettant l'accent sur la chronologie de l'envoi des messages.
On distingue deux types de diagrammes de séquence :

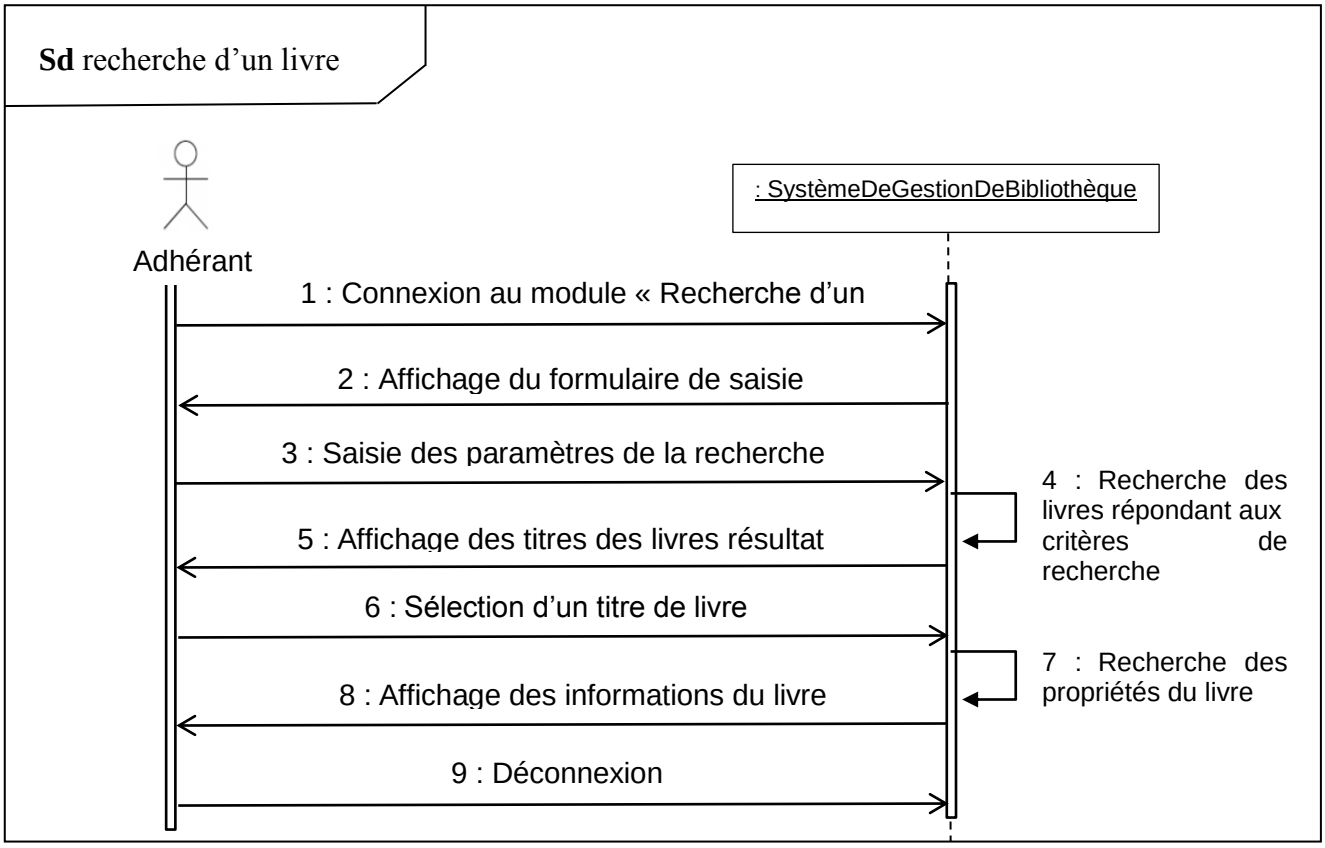
- Les diagrammes de séquence système.
- Les diagrammes de séquence représentant les interactions entre objets.

2.1.1 Diagrammes de séquence système : ils sont utilisés pour la documentation d'un cas d'utilisation. Ils assurent la description des interactions dans des termes proches de l'utilisateur.

Disposition des participants :

- Acteur principal à gauche ;
- Un objet représentant le système en boite noire ;
- Les éventuels acteurs secondaires sollicités durant le scénario à droite du système.

Exemple :



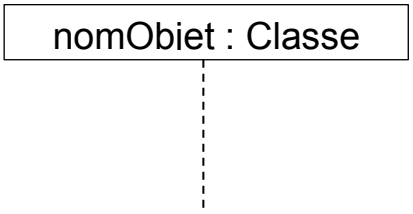
Ce diagramme de séquence système correspond au scénario nominal associé au cas d'utilisation «recherche d'un livre».

2.1.2 Diagrammes de séquence représentant les interactions entre objets : ils mettent en jeu :

- Un ensemble d'objets (instances de classes et/ou acteurs) ;
- La chronologie des échanges entre les objets (messages avec leurs paramètres et leur valeur de retour) ;
- Les contraintes de temps.

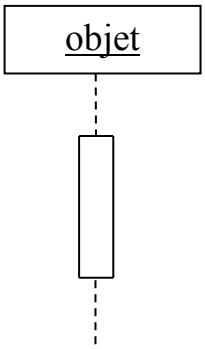
Ligne de vie

Une ligne de vie est utilisée pour indiquer l'existence d'objets dans le diagramme de séquence sur une période de temps. Elle est représentée graphiquement sous la forme d'une ligne en pointillés s'étendant vers le bas à partir de l'icône de l'objet concerné, indiquant la durée pendant laquelle l'objet existe, comme illustré dans la figure suivante :

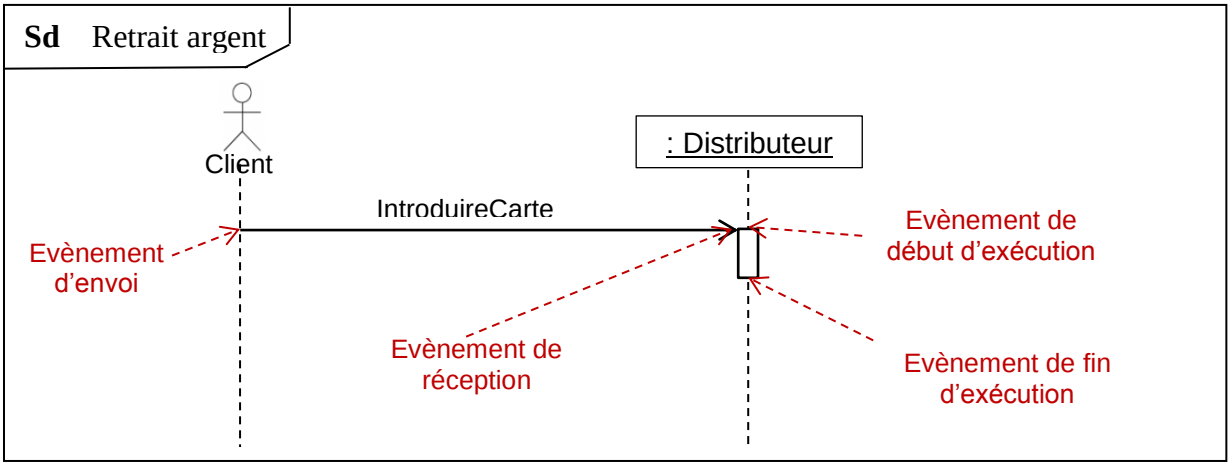


Barre d'activation

Lorsqu'un objet est déjà activé il peut quand même recevoir d'autres messages (appel d'une autre de ses méthodes), cette période d'activité se représente par un dédoublement de la bande d'activation sur la ligne de vie de l'objet. Le début et la fin d'une bande correspondent respectivement au début et à la fin d'une période d'activité.

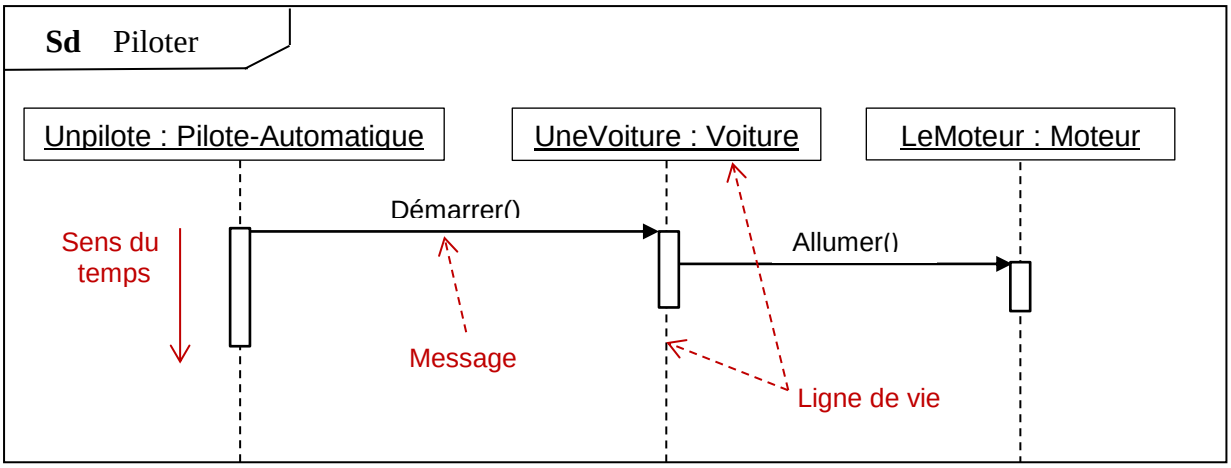


Exemple :



Représentation des interactions

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 étiquetés par le nom de l'opération ou du signal invoqué.



Les messages

Un message définit une communication particulière entre des lignes de vie. Ainsi, un message est une communication d'un objet vers un autre objet. La réception d'un message est considérée par l'objet récepteur comme un événement qu'il faut traiter (ou pas). Plusieurs types de messages existent, les plus communs sont :

- L'envoi d'un signal ;
- L'invocation d'une opération (appel de méthode) ;
- La création ou la destruction d'une instance.

Les messages sont généralement divisés en message synchrone, message asynchrone et message de retour. Comme illustré dans la Figure 5.1.

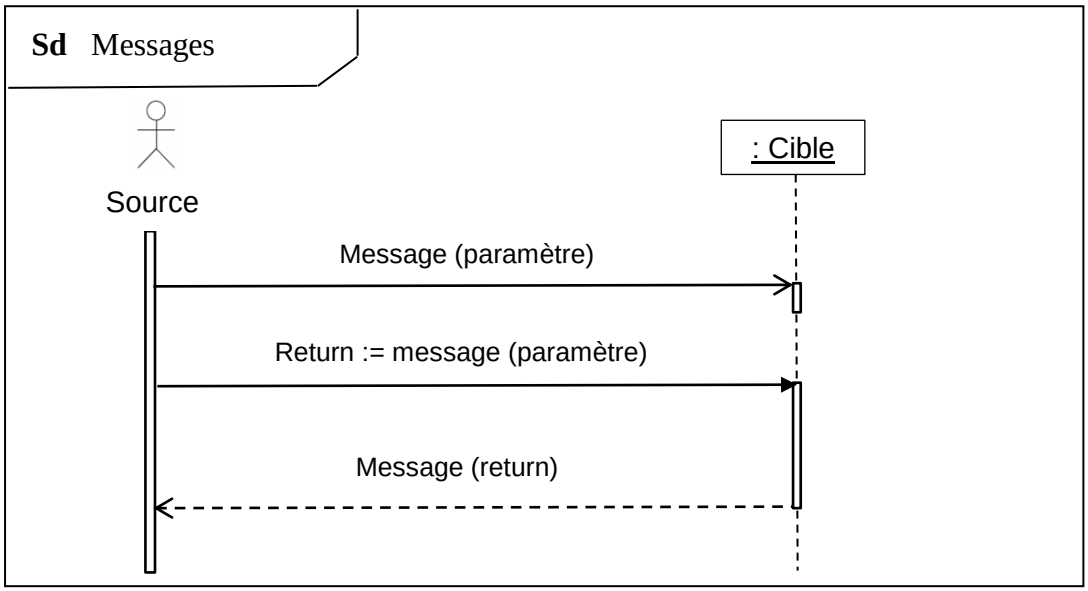


Figure 5.1 : Type de messages.

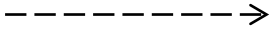
Message synchrone

L'expéditeur du message passe le contrôle au destinataire du message, puis arrête l'activité, en attendant que le destinataire du message abandonne ou reprenne le contrôle. Utilisé pour

exprimer la signification de la synchronisation. La flèche avec extrémité pleine symbolise ce type de message.

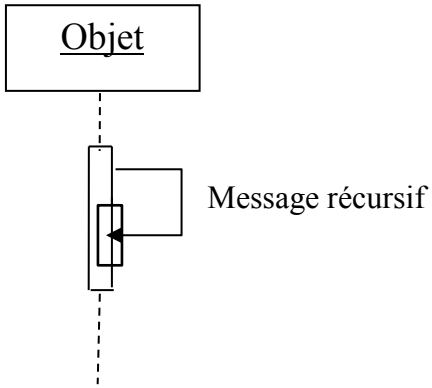
Message asynchrone
L'expéditeur du message transmet le signal au destinataire du message via le message, puis poursuit ses propres activités sans attendre que le destinataire renvoie le message ou le contrôle. Le destinataire et l'expéditeur des messages asynchrones fonctionnent simultanément. La flèche avec une extrémité non pleine qui symbolise ce type de message.

Message de retour
Le message de retour signifie le retour de l'appel de procédure. Une flèche en pointillé avec une extrémité non pleine qui symbolise ce type de message.



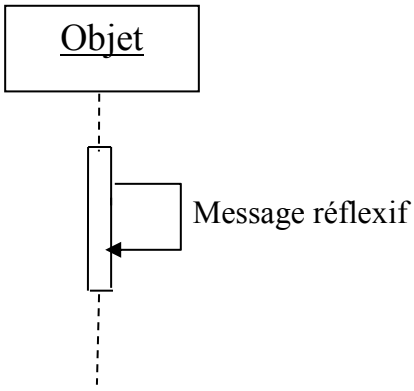
Message récursif
L'envoi de messages récursifs se représente par un dédoublement de la bande d'activation. Dans ce cas, l'objet apparait alors comme s'il était actif plusieurs fois.

Exemple :



Message réflexif
Il est possible de montrer un objet qui s'envoie un message à lui-même à l'aide d'une flèche qui boucle. Dans ce cas, la flèche commence et se termine sur la ligne de vie du même objet.

Exemple :



Contraintes temporelles

Des repères temporels avec des contraintes peuvent être placés le long de la ligne de vie. Un message avec un temps de propagation non négligeable peut être représenté sous la forme d'un trait oblique ou en l'écrivant explicitement.

Exemple :

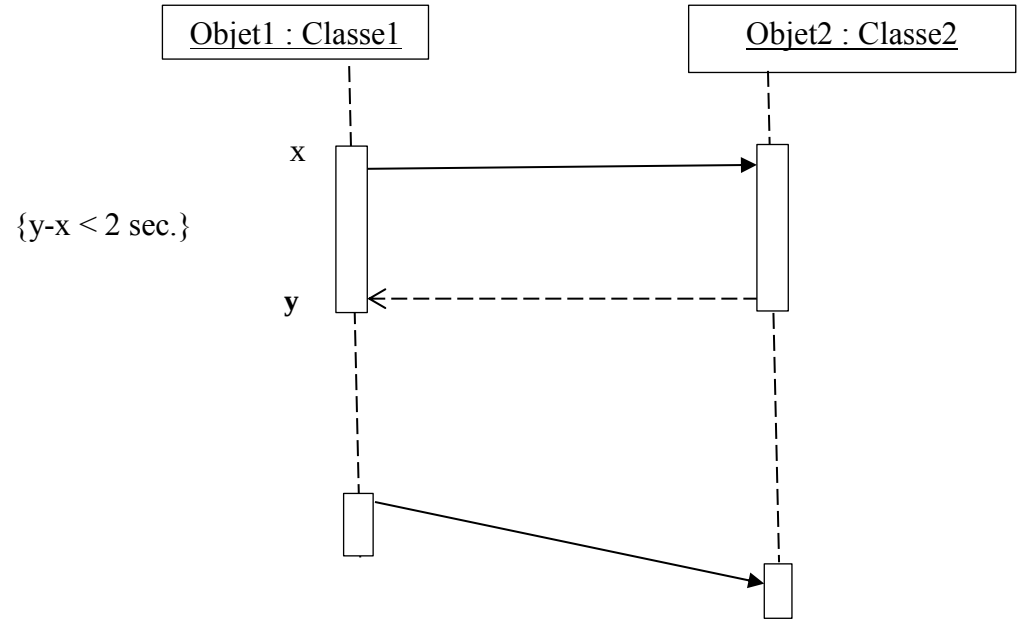


Figure 5.2 : Exemple type de représentation de contrainte temporelle.

Syntaxe des messages

La syntaxe des messages est :

<nomDuSignalOuDeOpération> ['(' [<argument>',' … ')']]

où la syntaxe des arguments est la suivante :

Si entrée : [<nomParamètre> '='] <valeur de l'argument>

Si sortie ou entrée-sortie : <nomParamètre> [':' <valeur de l'argument>]

Exemple :

initialiser(x = 100) est un message dont l'argument en entrée reçoit la valeur 100.

f(x :12) est un message avec un argument en entrée/sortie x qui prend initialement la valeur 12.

Syntaxe des réponses

Le récepteur du message peut aussi vouloir répondre et transmettre un résultat via un message de retour.

La **syntaxe de réponse** à un message est la suivante :

[<attribut> '='] message [':' <valeur de retour>]

où message représente le **message d'envoi**.

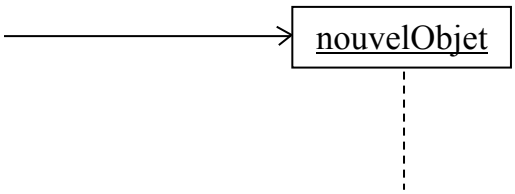
Exemples :

- y = f est un message de réponse à un message f ; la valeur de retour est affectée à y.
- y = f(0) est un message de réponse à un message f(0) ; la valeur de retour est affectée à y.
- y = f(x = 0) est un message de réponse à un message f(x = 0) ; la valeur de retour est affectée à y.
- y = f(x) : 0 est un message de réponse à un message f(x) ; la valeur de retour 0 est affectée à y.

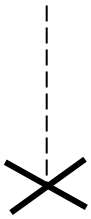
Création et destruction d'objets

Certains objets vivent pendant tout le diagramme, d'autres sont créés et/ou meurent pendant la séquence.

A. Création d'objets : Si un objet est créé par une opération, celui-ci n'apparait qu'au moment où il est créé. La création d'un objet est concrétisée par une flèche qui pointe sur le sommet d'une ligne de vie.



B. Destruction d'objets : Si un objet est détruit par une opération, la destruction se représente par une croix « X » qui marque la fin de la ligne de vie de l'objet.



Fragment d'interactions combinées

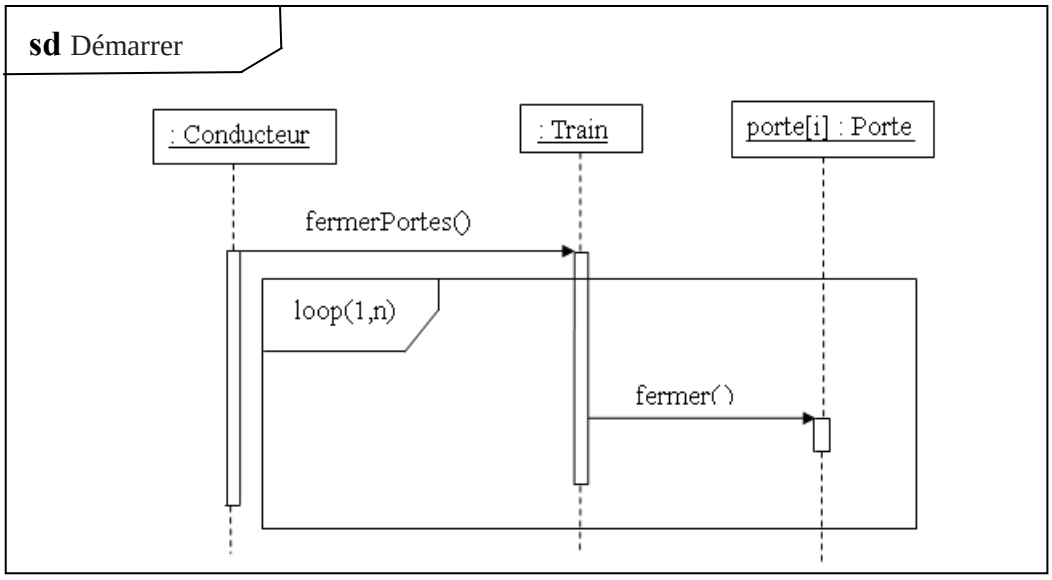
Un fragment d'interaction correspond à un ensemble d'interaction auquel on applique un opérateur. Un fragment d'interaction se représente globalement comme un diagramme de séquence avec indication dans le coin à gauche du nom de l'opérateur.

Les opérandes d'un opérateur d'interaction sont séparés par une ligne pointillée. Les conditions de choix des opérandes sont données par des expressions booléennes entre crochets. La liste suivante regroupe les opérateurs d'interaction par fonctions :

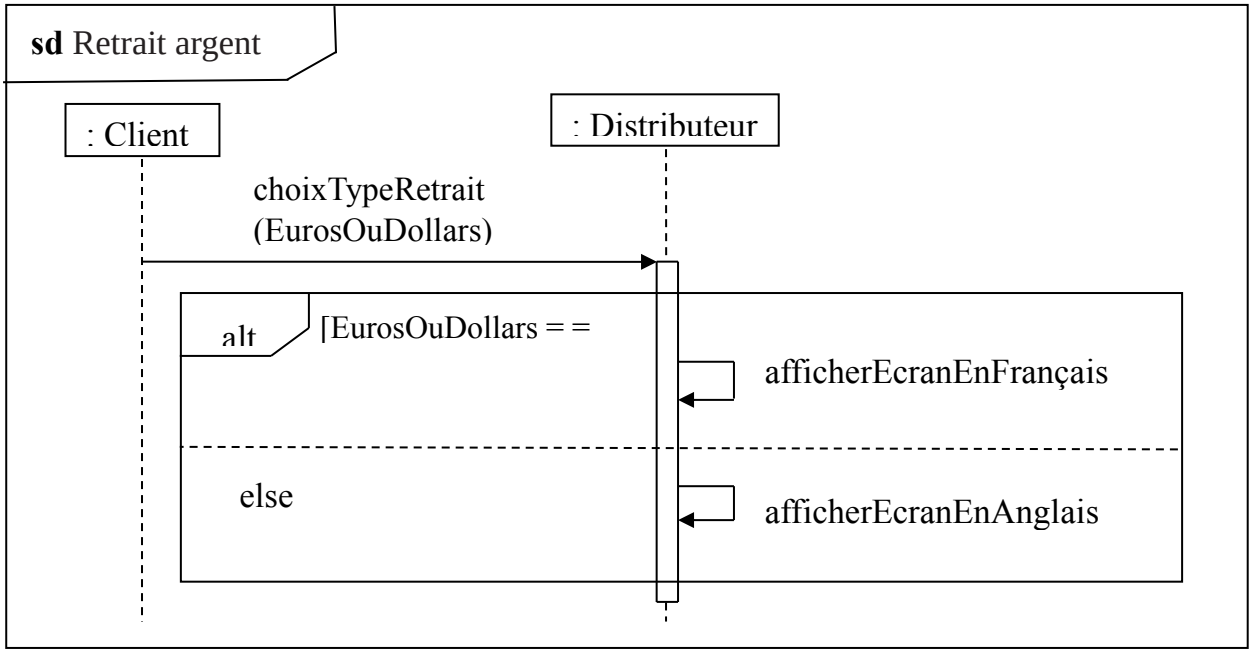
- Choix et boucle : alternative, option, break, loop
- Contrôle d'envoi en parallèle de messages : parallel, critical region
- Contrôle d'envoi de messages : ignore, consider, assertion, negative
- Ordre d'envoi des messages : weak sequencing, strict sequencing
- Référencement : ref

Opérateur loop : Cet opérateur correspond à une instruction de boucle. La syntaxe d'une boucle est la suivante :
loop ['(' <minint> [',' <maxint>] ')']

Exemple :
Cet exemple illustre la fermeture en boucle de toutes les portes d'un train.



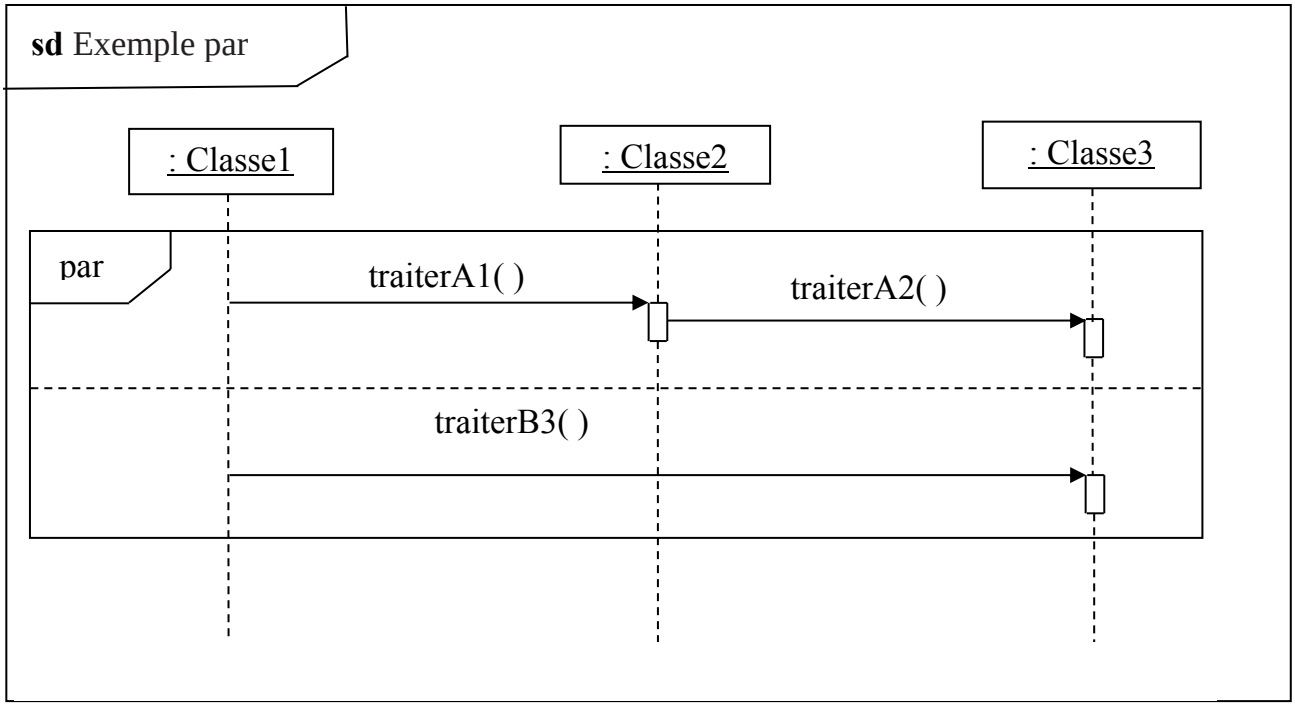
Opérateur alt : Cet opérateur correspond à une instruction de test avec une ou plusieurs alternatives possibles. Il est aussi permis d'utiliser les clauses de type sinon (else).



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

L'opérateur ref permet de faire appel à un autre diagramme de séquence.

Opérateur par : L'opérateur par permet de représenter deux séries d'interactions qui se déroulent en parallèle.



Dans cette figure, les traitements A1 et A2 sont menés en parallèle au traitement B3.

Nous venons de voir les 4 fragments d'interactions les plus utilisés (**opt**, **alt**, **loop** et **par**). Il en existe en réalité 13 au total, ci-dessous la liste des neuf (9) autres :

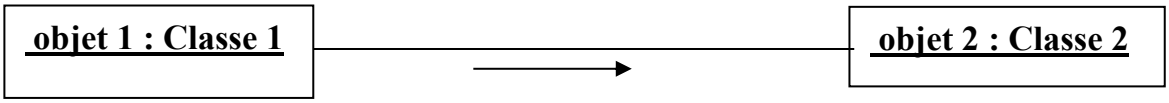
- **break** : permet de représenter une situation exceptionnelle correspondant à un scénario de rupture par rapport au scénario général.
- **critical** : pour les fragments qui doivent se dérouler sans être interrompus.
- **ignore** : pour les fragments facultatifs.
- **consider** : pour les fragments obligatoires.
- **assert** : permet d'indiquer qu'une séquence d'interactions est l'unique séquence possible en considérant les messages échangés dans le fragment. Toute autre configuration de message est invalide.
- **neg** : pour indiquer que la séquence à l'intérieur du fragment n'est pas valide.
- **weak** : pour les fragments dont l'ordre des messages n'a pas d'importance
- **strict** : pour les fragments dont les messages doivent se dérouler dans un ordre bien précis.
- **ref** : permet de faire appel à un autre diagramme de séquence.

2.2 Diagrammes de communication

Le diagramme de communication montre l'interaction entre les objets ou les rôles associés à la ligne de vie et les messages directement transmis par la ligne de vie. Dans les versions

antérieures d'UML, le diagramme de communication s'appelait diagramme de collaboration et la notation était différente.
Un diagramme de communication est un diagramme interactif qui peut être utilisé pour explorer le comportement dynamique d'un système ou d'une application logicielle. Il fournit les mêmes informations que le diagramme de séquence via une autre vue. Le diagramme de séquence est destiné à décrire la séquence des messages après une période de temps alors que le diagramme de communication est destiné à décrire la structure des messages passés entre les objets participant à l'interaction. Ces diagrammes illustrent le flux de messages entre les objets et les relations implicites entre les classes.

Le lien entre les objets est symbolisé par un trait matérialisant le support des messages échangés.



Sens et identification du message

La syntaxe complète des messages est :
[<numéro_séquence>] [<expression>] : <message>
où
Message a la même forme que dans les diagrammes de séquence ;

- Numéro_séquence numéro hiérarchique du message de type 1.1, 1.2… avec utilisation de lettre pour indiquer la simultanéité d'envoi de message ;
- Expression précise une itération (boucle) ou un embranchement (opérateurs de choix)

  - [<clause d'itération>] représente une itération. La clause d'itération peut être exprimée dans le format i := 1…n.
  - [<clause de condition>] représente un choix. La clause de condition est une condition booléenne.

Exemples :

2 : affiche(x, y) est un message simple.

1.3.1 : trouve("Hadock") est un appel emboîté.

4 [x < 0] : inverse(x, couleur) est un message conditionnel.

3.1 *[i :=1..10] : recommencer() représente une itération.

Exemple :
Dans la Figure 5.3, l'envoi du message 1.2 suit immédiatement celui du message 1.1 et ces deux messages font partie du flot des messages 1.

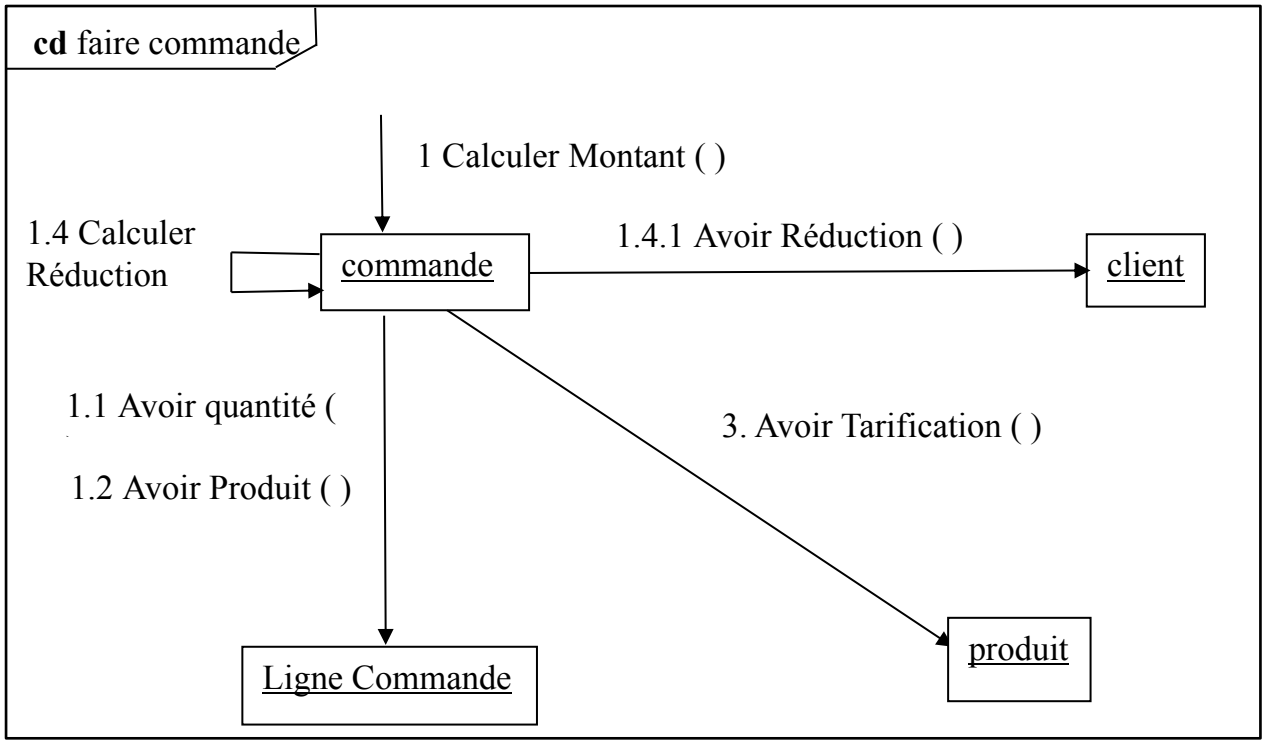


Figure 5.3 : Exemple d'un diagramme de communication.

3. Diagramme d'états-transitions

Diagramme d'états-transitions, représentant le modèle de comportement. Il exprime le comportement du système en décrivant l'état du système et les événements qui provoquent la transition de l'état du système, en indiquant quelles actions seront effectuées à la suite d'un événement spécifique (comme le traitement de données, etc.). Diagramme d'états-transitions décrit comment le système répond aux événements externes et comment agir. Ces diagrammes se produisent dans la phase d'analyse des exigences de l'ingénierie logicielle. Le modèle d'état est un modèle de comportement qui décrit la réponse du système aux événements internes ou externes. Il décrit l'état et les événements du système, ainsi que la transition du système entre les états déclenchés par l'événement. Ce modèle convient pour décrire des systèmes en temps réel.

3.1. Notion d'automate à états finis

Un automate à états finis est un automate dont le comportement des sorties dépend non seulement de l'état de ses entrées, mais aussi d'un historique des sollicitations passées. Cet historique est caractérisé par un état global qui est un jeu de valeurs d'objet, pour une classe donnée, produisant la même réponse face aux événements. Toutes les instances de la même classe avec le même état global réagissent de la même manière à un événement.

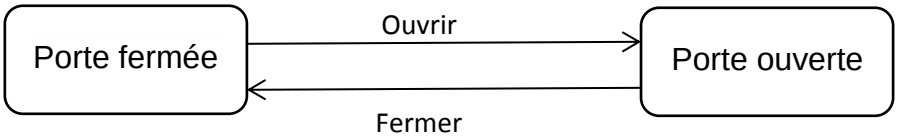
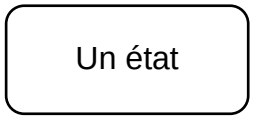


Figure 5.4 : Diagramme d'états-transitions simple.

3.2 Composants d'un diagramme d'états-transitions

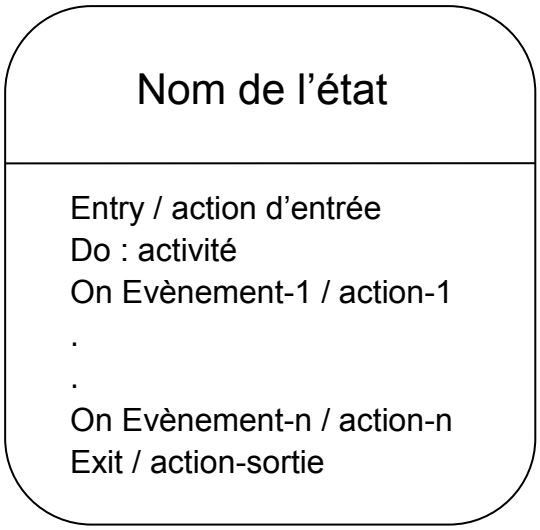
Les éléments qui composent un diagramme d'états-transitions sont les suivants :
Etat : un état est une condition au cours de la vie d'un objet. Il peut remplir certaines conditions pour effectuer certaines activités ou attendre de recevoir certains événements. Graphiquement, un état est symbolisé par un rectangle aux coins arrondis.



Forme générale d'un état

Un état est généralement un ensemble de valeurs d'attribut dans un objet de classe donné, qui doit être unique dans le contexte dans lequel il se trouve, mais il peut être anonyme. Lors de la modélisation du système, on ne peut se soucier que des attributs qui affectent évidemment le comportement de l'objet et l'état de l'objet exprimé par eux, et ignorer l'état qui n'a rien à voir avec le comportement de l'objet. Les états peuvent contenir des actions qui seront exécutées lors de l'entrée ou de la sortie d'un état ou lors de l'occurrence d'un événement alors que l'objet est dans l'état en question. Un état se représente comme une classe à l'aide d'un rectangle comportant plusieurs compartiments. Les compartiments de base sont :

- Le nom 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.



Une action est une opération instantanée qui ne peut être interrompue, elle est associée à l'événement qui déclenche une transition. Elle peut être exprimée par la syntaxe suivante : événement / action. Ceci exprime que la transition, qui est déclenchée par l'événement cité, entraîne l'exécution de l'action spécifiée sur l'objet, à l'entrée du nouvel état.

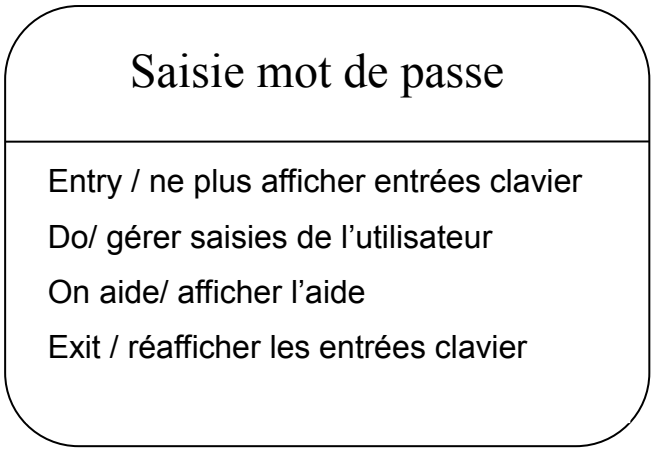
Exemple :
il pleut / ouvrir parapluie

Une action correspond à une opération disponible dans l'objet dont on représente les états. De plus, les actions propres à un état peuvent aussi être documentées directement à l'intérieur de l'état.

UML définit un certain nombre de champs qui permettent de décrire les actions dans un état :

entry / action : action exécutée à l'entrée de l'état
exit / action : action exécutée à la sortie de l'état
on événement / action : action exécutée à chaque fois que l'événement cité survient
do / action : action récurrente ou significative, exécutée dans l'état

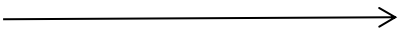
Exemple :



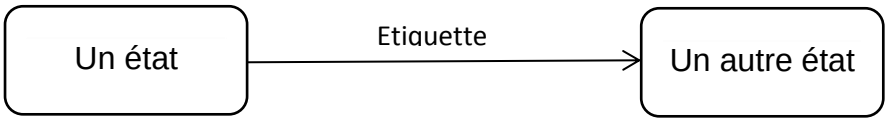
Etat initial : est un pseudoétat qui montre l'état de départ, par défaut, lorsque le diagramme d'états-transitions, ou l'état enveloppant, est invoqué. Lorsqu'un objet est créé, il entre dans l'état initial. L'état initial est symbolisé par un gros point noir.

Etat final : est un pseudoétat qui montre que le diagramme d'états-transitions, ou l'état enveloppant, est terminé. L'état final est symbolisé par un gros point noir encerclé.

Transition : est la relation entre deux états, indiquant que l'objet dans le premier état effectuera certaines actions et entrera dans le second état lorsqu'un événement spécifié se produit et que les conditions spécifiées sont remplies. Les transitions sont matérialisées par des arcs orientés/flèches liant les états entre eux.



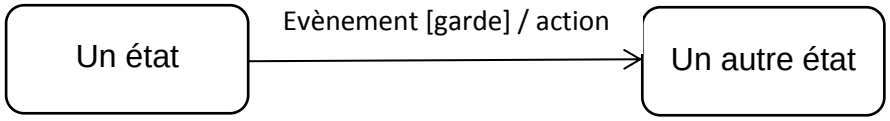
Exemple :



Remarque

Une transition peut-être nommée par un évènement, et un évènement peut réflexif et conduire au même état.

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.

Super-état, historique : Un super état permet de structurer le diagramme en indiquant plusieurs niveaux de distinction entre les états. Le symbole de modélisation « historique » représente (mémorise) le dernier sous-état actif d'un super-état, pour y revenir directement ultérieurement.

Exemple 1 :

Dans cet exemple, la Figure (1) montre que l'objet décrit par l'automate réagit de la même manière à l'arrivée de l'événement Event2. Qu'il soit à l'état X ou à l'état Y, l'objet passe à l'état Z. Par conséquent, ce comportement peut être généralisé comme le montre la Figure (2), en créant un super état groupant l'état X et l'état Y.

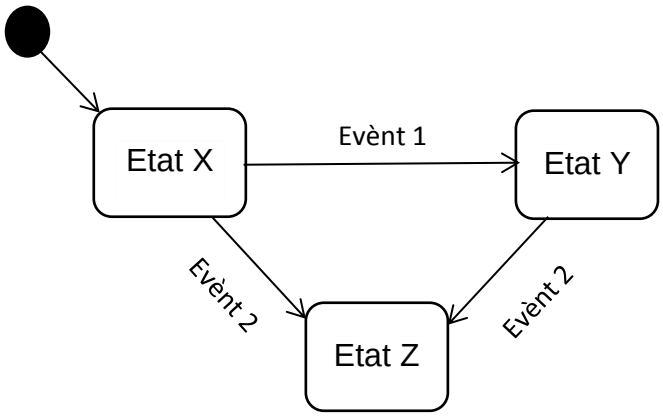


Figure (1)

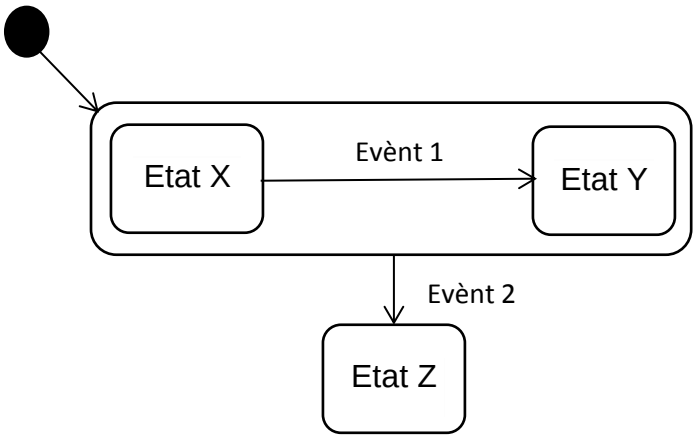
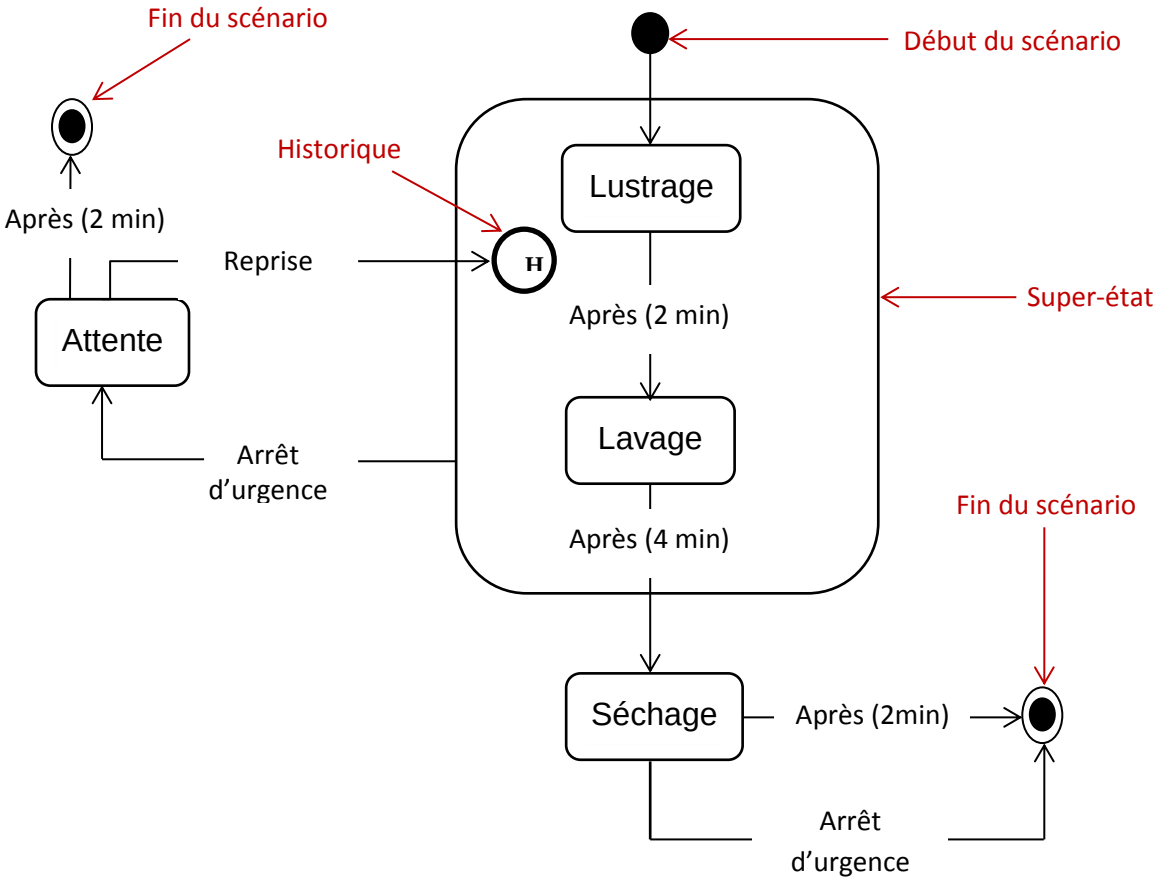


Figure (2)

Exemple 2 :

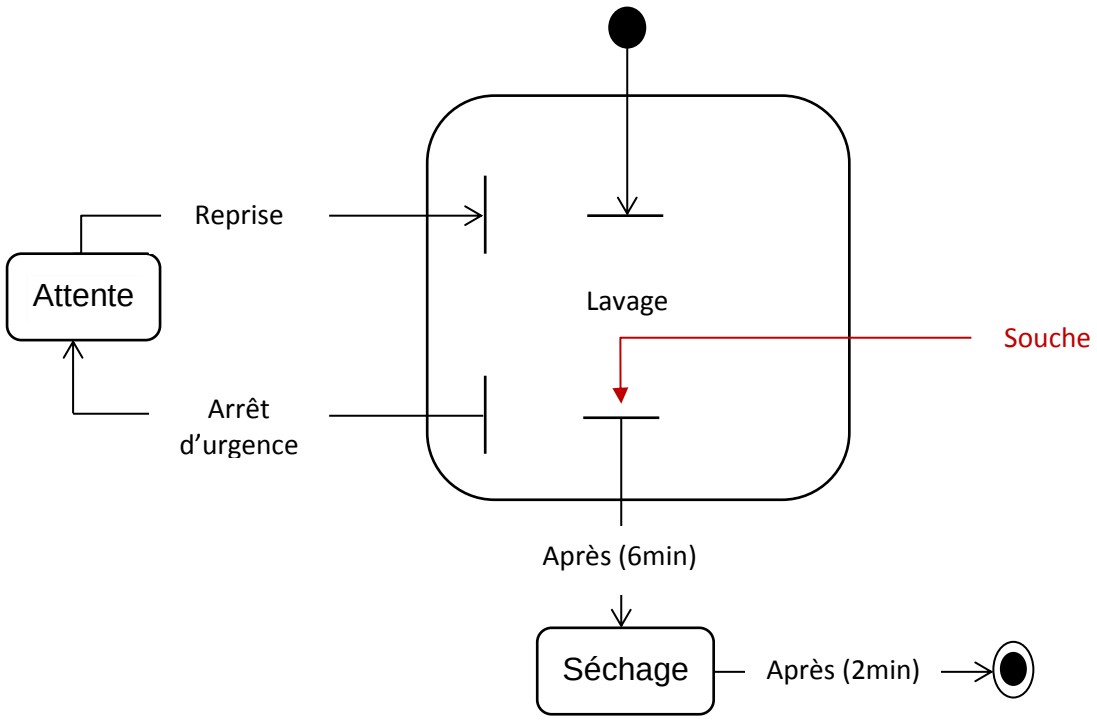
Cet exemple illustre les différents états par lesquels passe une machine à laver de voiture. Le client peut appuyer sur le bouton d'arrêt d'urgence pendant la phase de lustrage ou de lavage. S'il appuie sur ce bouton, la machine se met en attente. Il dispose alors de deux minutes pour reprendre le lavage ou le lustrage (la machine continue dans la phase de lavage ou de lustrage, selon l'état dans lequel elle a été interrompue), sinon la machine s'arrête. Le client peut

également interrompre la machine pendant la phase de séchage, dans ce cas, la machine s'arrêtera définitivement avant de reprendre un autre cycle complet.



Souche : il permet de réduire la charge d'information, tout en matérialisant la présence de sous-états, à l'aide de souches pour introduire plus d'abstraction dans un diagramme d'états-transitions complexe.

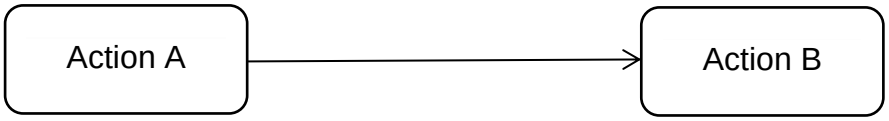
Exemple :



4. Diagramme d'activités

Le diagramme d'activité est le diagramme le plus simple parmi de nombreux diagrammes UML car même les parties prenantes du système logiciel peuvent également comprendre le diagramme d'activité similaire à l'organigramme. Les diagrammes d'activités permettent de mettre l'accent sur les traitements. De ce fait, ils sont particulièrement adaptés à la modélisation du routage des flux de contrôle et des flux de données. Ainsi, les diagrammes d'activité permettent de symboliser le comportement d'une méthode, le déroulement d'un cas d'utilisation et/ou les enchainements d'activités. En bref, le diagramme d'activités est un autre outil couramment utilisé par UML pour modéliser le comportement dynamique du système. Il décrit la séquence des activités et montre le flux de contrôle d'une activité à une autre. Le diagramme d'activité est essentiellement un organigramme. Le diagramme d'activité se concentre sur le flux de contrôle d'une activité à une autre et est un processus piloté par un traitement interne.

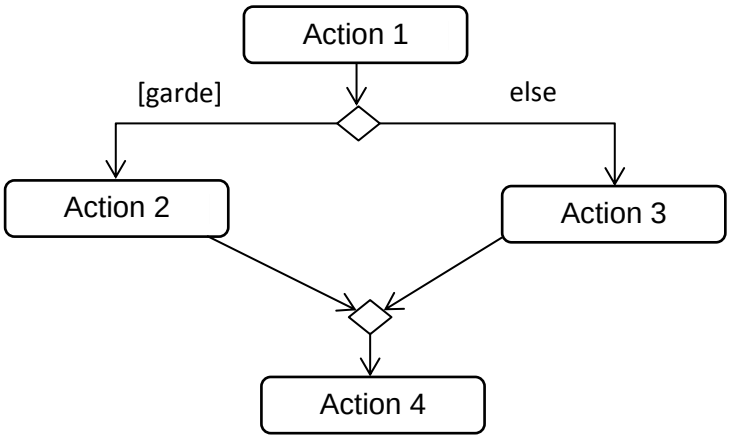
Une activité représente une exécution d'un mécanisme, un déroulement d'étapes séquentielles. Le passage d'une activité vers une autre est concrétisé par une transition. Bien que les transitions soient automatiques entre les activités, il est inutile de spécifier les événements. Aussi, les transitions sont déclenchées par la fin d'une activité et provoquent le début immédiat d'une autre. Graphiquement, les activités sont symbolisées par des rectangles aux bords arrondis.



4.1 Décision / Fusion

Le comportement conditionnel est représenté par des **décisions** des **fusions**. Une **décision** (ou branchement) permet de représenter des transactions conditionnelles en utilisant des expressions booléennes. Une **décision** comporte une seule entrée et au moins deux sorties. Une **fusion** permet de rassembler plusieurs flots alternatifs entrants en un seul flot sortant. Il n'est pas utilisé pour synchroniser des flots concurrents (c'est le rôle du nœud d'union), mais pour accepter un flot parmi plusieurs. Une **fusion** marque la fin d'un comportement conditionnel. Graphiquement, une décision et une fusion sont matérialisées par un losange.

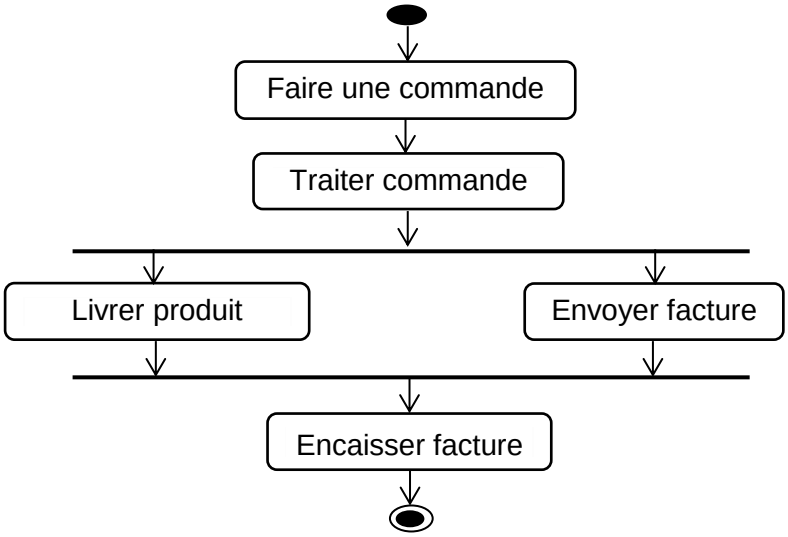
Exemple :



4.2. Débranchement et jonction

Il peut y avoir au moins deux flux de contrôle exécutés simultanément lorsque l'objet est en cours d'exécution. Afin de modéliser le flux de contrôle simultané, le concept de **débranchement** et de **jonction** est introduit dans UML. Le **débranchement** est utilisé pour diviser le flux d'actions en deux ou plusieurs branches qui s'exécutent simultanément, tandis que la **jonction** est utilisée pour synchroniser ces branches simultanées afin d'atteindre l'objectif de compléter une transaction ensemble.

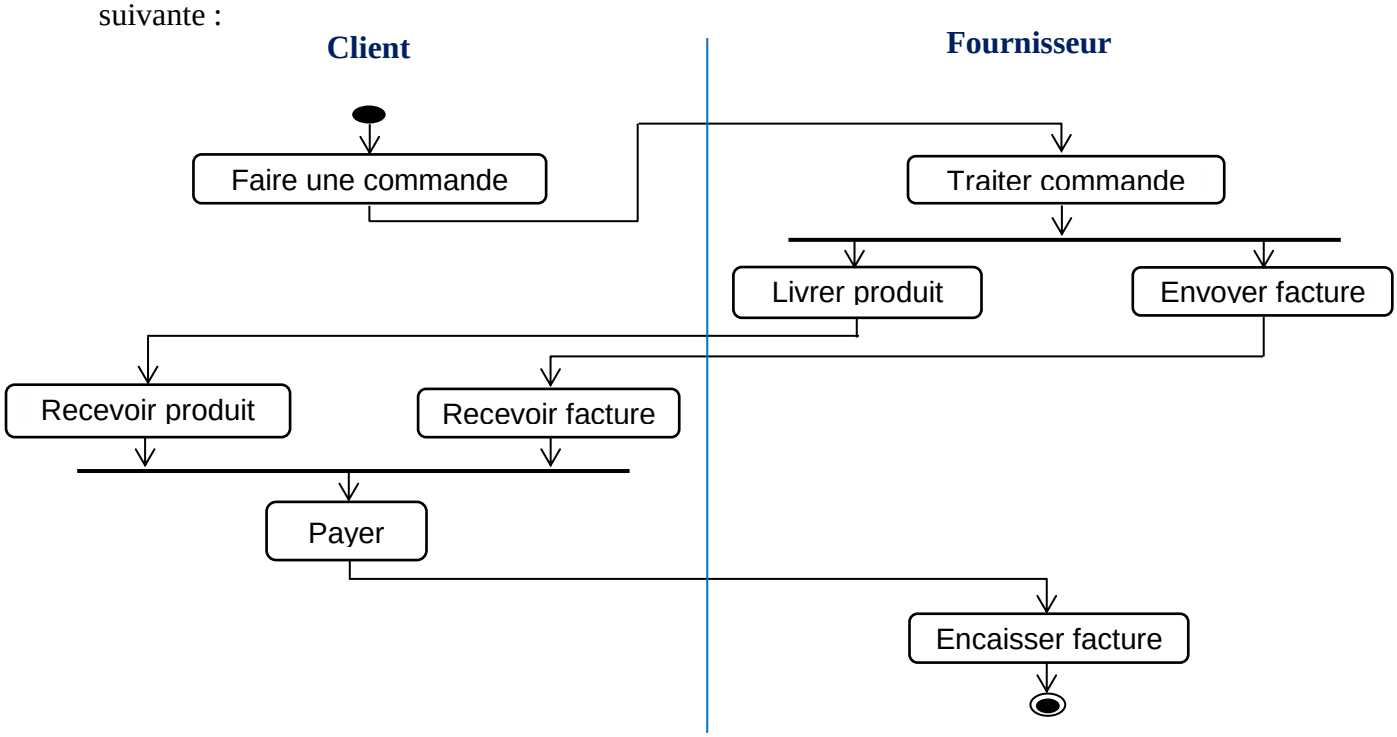
Exemple :
Le traitement d'une commande par un diagramme d'activités est illustré dans la figure suivante :



4.3. Couloir d'activités

Le couloir divise les activités du diagramme d'activités en plusieurs groupes et affecte chaque groupe à l'organisation commerciale responsable de ce groupe d'activités, c'est-à-dire l'objet. Dans le diagramme d'activités, le couloir distingue les objets responsables des activités, et il indique clairement quelles activités sont effectuées par quels objets. Dans un diagramme d'activités qui comprend des couloirs, chaque activité ne peut appartenir clairement qu'à un couloir.
Les couloirs d'activités sont tracés avec des lignes pleines verticales et les zones séparées par des lignes verticales sont des couloirs d'activités. En haut de couloir, vous pouvez donner le nom de couloir ou le nom de l'objet, qui est responsable de toutes les activités dans le couloir. Il n'y a pas d'ordre dans les couloirs. Les activités dans les différents couloirs peuvent être exécutées séquentiellement ou simultanément. Le flux d'actions et d'objets est autorisé à traverser la ligne de séparation.

Exemple :
Le traitement d'une commande passée par un client à son fournisseur est illustré par la figure suivante :



5. Conclusion

Le diagramme de séquence est un bon diagramme utilisé pour enregistrer les exigences du système et organiser la conception du système. La raison pour laquelle le diagramme de séquence est si utile est qu'il montre la logique d'interaction entre les objets du système en fonction de la séquence temporelle de l'interaction.

Le diagramme de communication permet de valider les associations du diagramme de classes en les utilisant comme support de transmission de messages. En d'autres termes, un diagramme de communication est une représentation simplifiée d'un diagramme de séquence se concentrant sur l'échange de messages entre objets.

Le diagramme d'états est principalement utilisé pour établir le modèle de comportement dynamique de la classe d'objets ou de l'objet, représentant la séquence d'états subis par un objet, les événements qui provoquent la transition d'état ou d'activité et les actions qui accompagnent la transition de l'état ou de l'activité.

Nous consacrerons le prochain chapitre à la présentation de quelques autres concepts et diagrammes UML, à savoir le diagramme de composants, le diagramme de déploiement, le diagramme de structure composite, ainsi que les mécanismes d'extension.

6. Exercices

Exercice 1 :
Le déroulement normal d'utilisation d'un distributeur automatique de billets est le suivant :

- Le client introduit sa carte de la banque.
- La machine vérifie alors la validité de la carte bancaire et demande le code au client si le code est correct, elle envoie une demande d'autorisation de prélèvement au groupement de banques. Ce dernier renvoie le solde autorisé à prélever.
- Le distributeur propose alors plusieurs montants à prélever.
- Le client saisit le montant à retirer après contrôle du montant par rapport au solde autorisé, le distributeur demande au client s'il désire un ticket.
- Après la réponse du client, la carte est éjectée et récupérée par le client.
- Les billets sont alors délivrés (ainsi que le ticket).
- Le client récupère enfin les billets et son ticket.

Travail demandé :
Modéliser cette situation à l'aide d'un diagramme de séquence en ne prenant en compte que le cas où tout se passe bien.

NB : on identifiera les scénarios qui peuvent poser problème en incluant des commentaires dans le diagramme.

Exercice 2 :
Décrire par un diagramme de communications le fonctionnement d'un ascenseur (une personne appuie sur un bouton, l'ascenseur arrive, les portes s'ouvrent, la personne entre...).

Exercice 3 :
Un ensemble de personnes décident d'établir un contrat. Pour ce faire elles rédigent un projet par itération successive. Le contrat est ensuite informellement accepté par les parties, et devient ce que l'on appelle un préaccord. A ce stade il peut toujours être l'objet de modification et revenir à l'état de projet. Une fois le préaccord définitivement établi, le contrat est signé par les parties. Dès ce moment les partenaires sont liés. Une fois signé, le contrat peut être rendu exécutoire par une décision d'une des parties. Un contrat en exécution peut faire l'objet de discussions qui sont réglées par un arbitre désigné à cet effet. Le contrat une fois exécuté prend fin.

Travail demandé :
Dessinez un diagramme d'état/transition résumant les états possibles d'un objet "contrat" tel que décrit dans l'énoncé ci-dessus.

Exercice 4 :
Voici la recette pour faire une bonne mousse au chocolat :

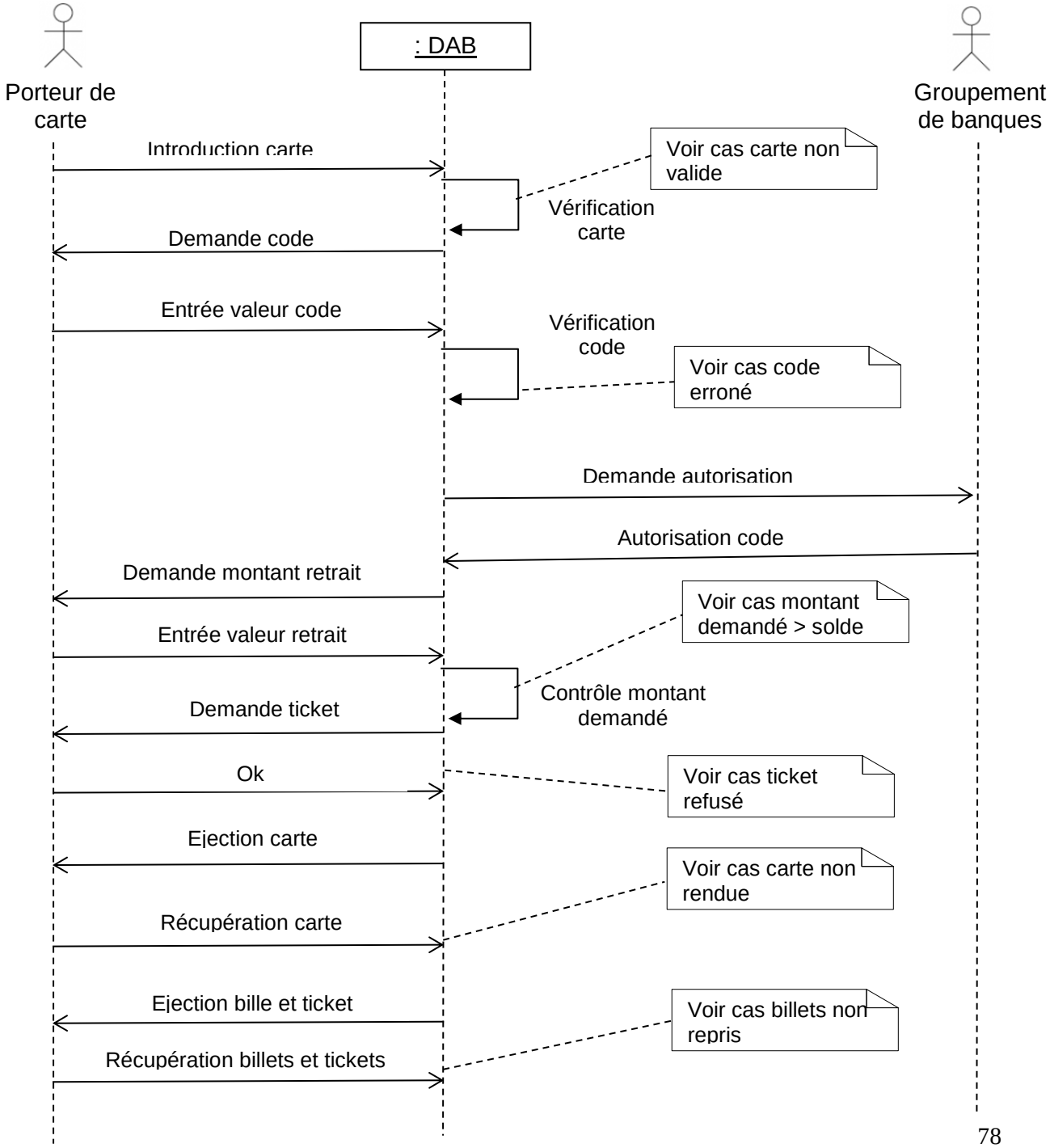
- Commencer par casser le chocolat en morceaux, puis le faire fondre.
- En parallèle, casser les œufs en séparant les blancs des jaunes.
- Quand le chocolat est fondu, ajouter les jaunes d'œuf.
- Battre les blancs en neige jusqu'à ce qu'ils soient bien fermes.
- Les incorporer délicatement à la préparation chocolat sans les briser.
- Verser dans des ramequins individuels.
- Mettre au frais au moins 3 heures au réfrigérateur avant de servir.

Travail demandé :

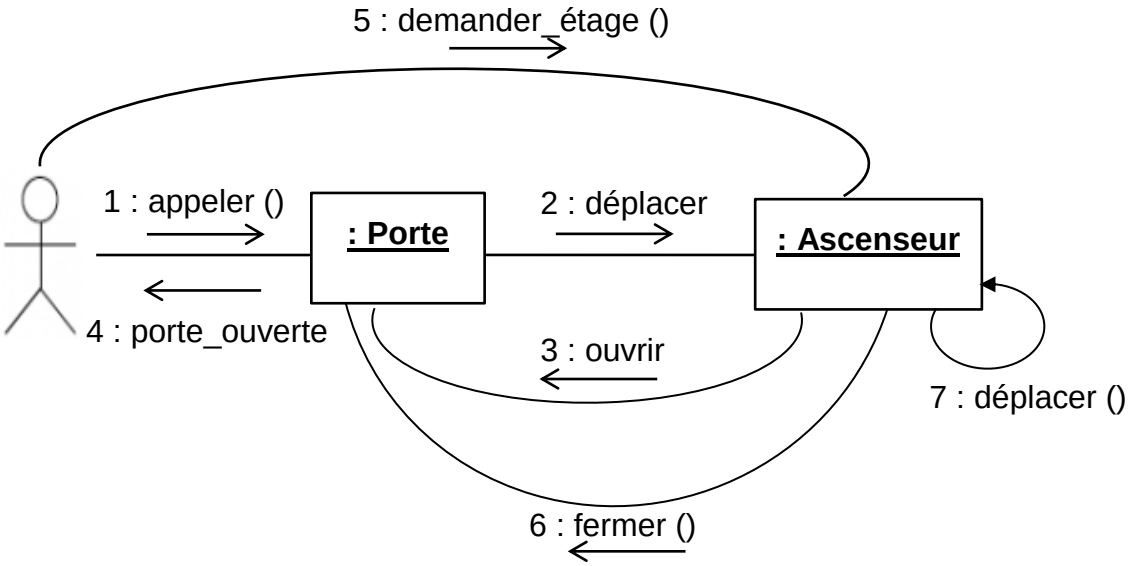
1. Etablir le diagramme d'activités pour modéliser la recette.
2. Dans le diagramme d'activité de la question 1, on ne voit pas encore les ingrédients manipulés. Ajouter les flots d'objets (objet et son état) pour compléter le diagramme.
3. Le chef et son assistant vont partager le travail pour préparer la recette. Créer un autre les partitions représentant les entités responsables des actions.

7. Corrigés des exercices

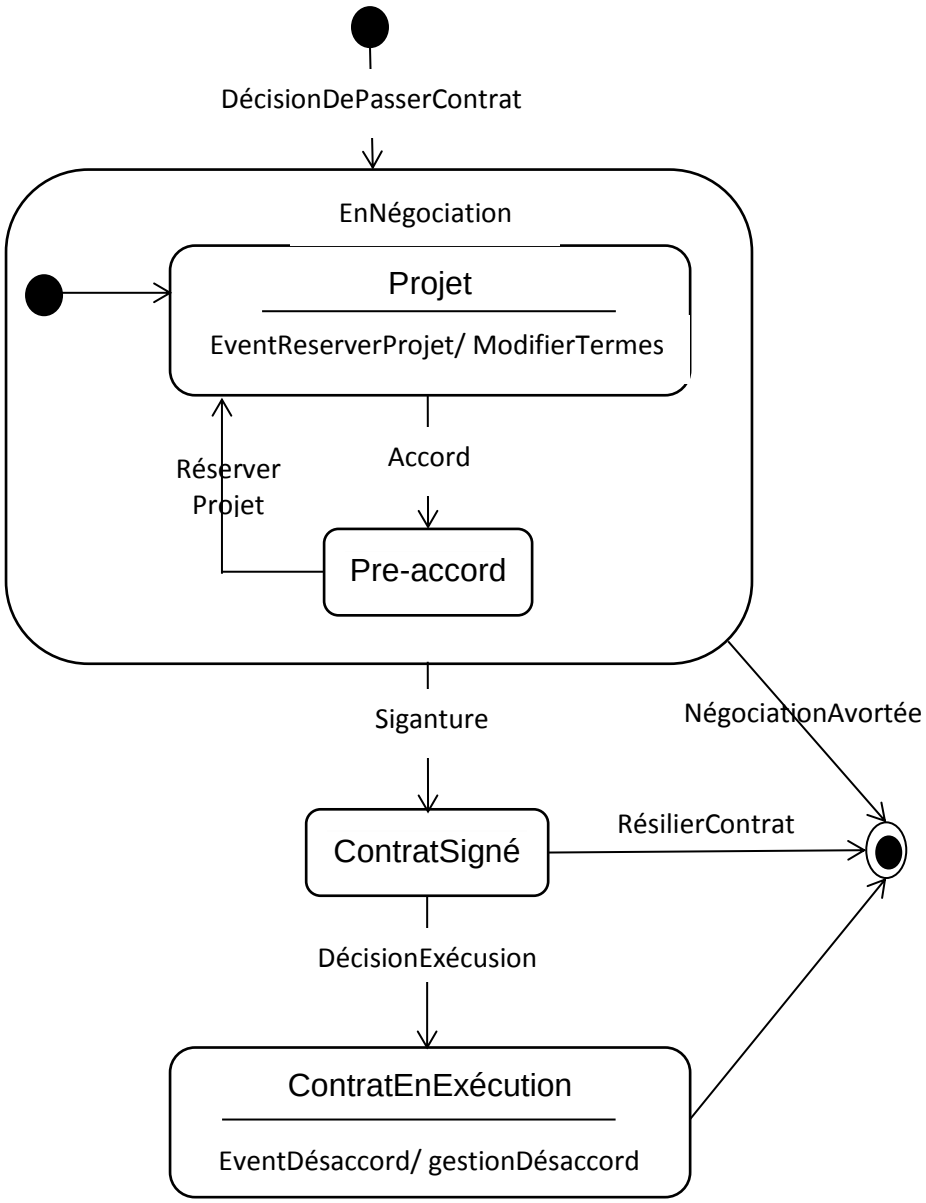
Corrigé de l'exercice 1



Corrigé de l'exercice 2

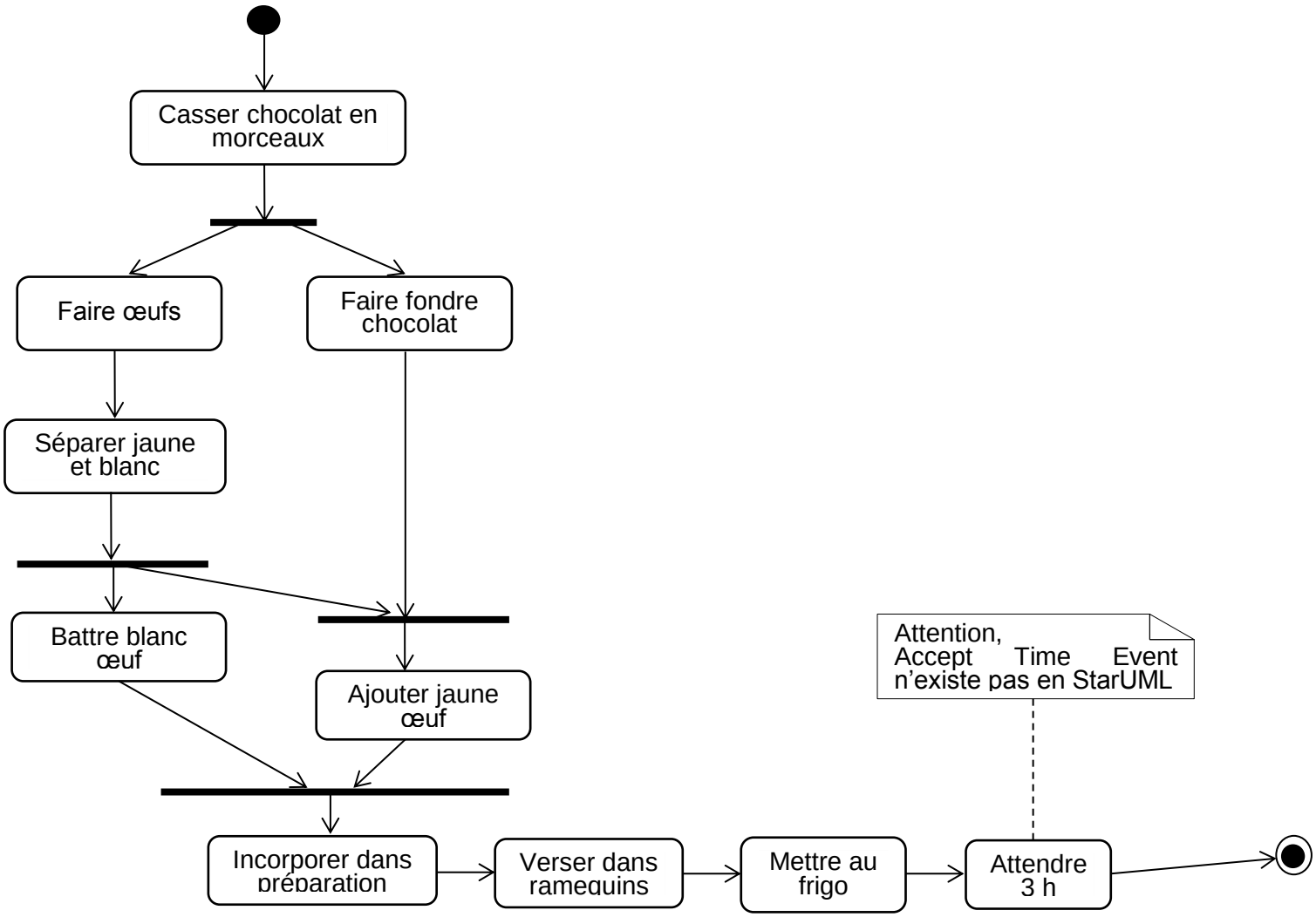


Corrigé de l'exercice 3

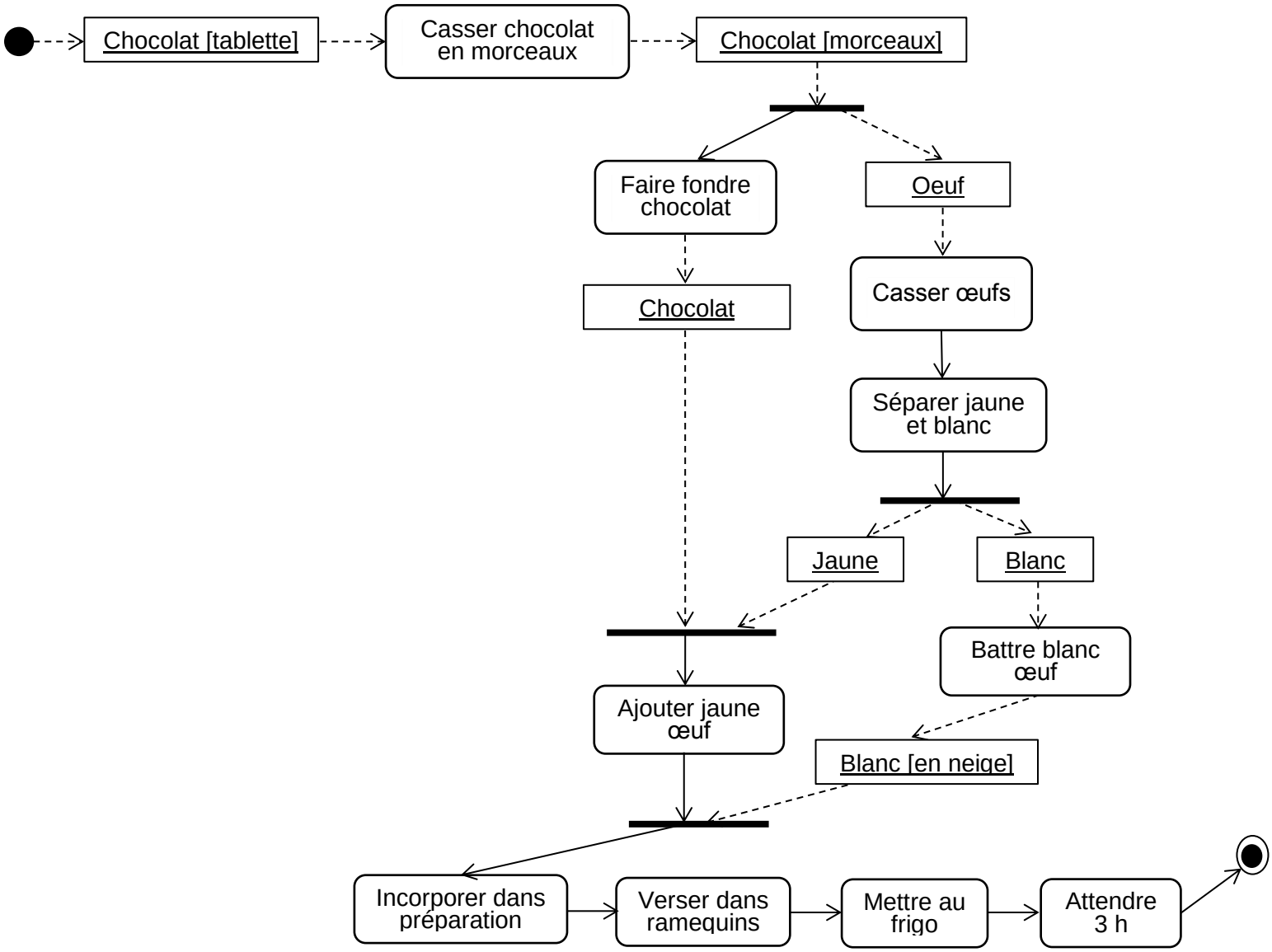


Corrigé de l'exercice 4

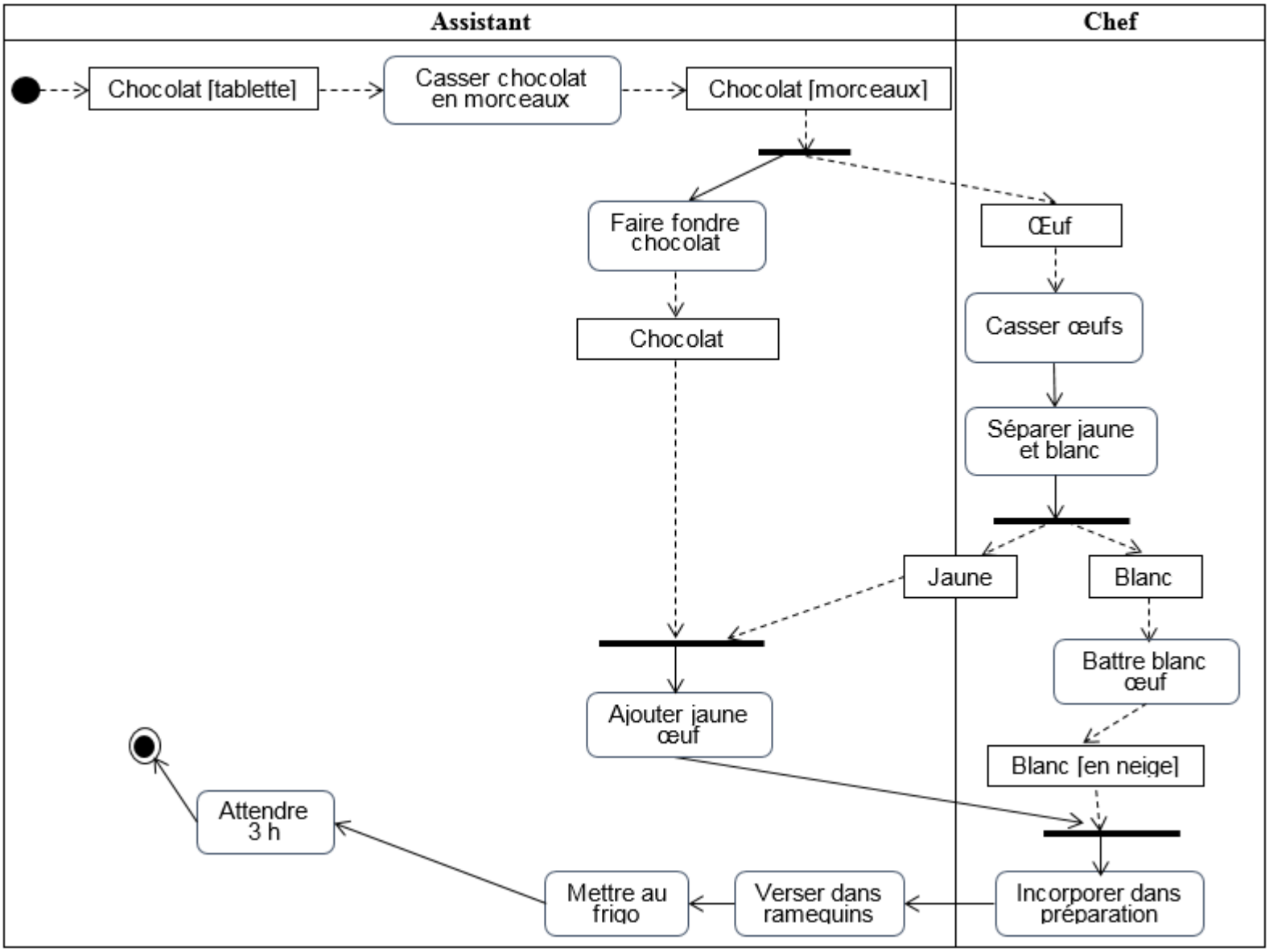
1. Etablir le diagramme d'activités pour modéliser la recette.



2. Dans le diagramme d'activité de la question 1, on ne voit pas encore les ingrédients manipulés. Ajouter les flots d'objets (objet et son état) pour compléter le diagramme.



3. Le chef et son assistant vont partager le travail pour préparer la recette. Créer un autre les partitions représentant les entités responsables des actions.



Chapitre 6 : Autres notions et diagrammes UML

Objectif

À l'issue de ce chapitre, l'apprenant sera capable de :

- Apprendre à décrire le système modélisé sous forme de composants réutilisables, tout en mettent en évidence leurs relations de dépendance ;
- Apprendre à élaborer le diagramme de déploiement ;
- Apprendre à établir le diagramme de de structure composite qui permet d'approfondir la description de la structure interne de plusieurs classes ;
- Surmonter les limitations de la représentation graphique UML en spécifiant précisément la conception détaillée du système.
- Connaître deux mécanismes d'extension permettant la personnalisation et l'extension de la propre syntaxe et sémantique d'UML pour pouvoir s'adapter à des domaines d'application spécifiques.

1. Introduction

Le langage de modélisation unifié (UML) a été conçu pour être la distillation des meilleures pratiques en matière de développement logiciel. Pour atteindre cet objectif ambitieux, UML fournit un ensemble complet d'outils de création de diagrammes, à savoir le diagramme de composants, diagramme de déploiement, diagramme de structure composite, etc. Bien qu'UML soit une notation à usage général, il peut localiser sa capacité à modéliser certains domaines particuliers, pour lesquels des langages spécialisés peuvent être plus appropriés. Pour remédier à ce problème, UML propose des mécanismes d'extension permettant la personnalisation et l'extension de sa propre syntaxe et sémantique en fonction de domaines d'application typiques.

2. Diagrammes de composants

2.1. Définition

Les diagrammes de composants sont utilisés pour modéliser les aspects physiques des systèmes orientés objet. Ces systèmes sont utilisés pour visualiser, spécifier et documenter les systèmes basés sur des composants, et sont également utilisés pour créer des systèmes exécutables par ingénierie directe et inverse. Les diagrammes de composants sont essentiellement des diagrammes de classes. Ils se concentrent sur les composants du système. Ces composants sont généralement utilisés pour modéliser la vue de réalisation statique du système.

2.2. Éléments du diagramme de composants

Les éléments du diagramme de composants comprennent des composants, des interfaces, des relations, des ports et des connecteurs. Parmi eux, il existe une relation de dépendance entre les composants et une relation d'implémentation entre les composants et les interfaces.

2.2.1. Composant

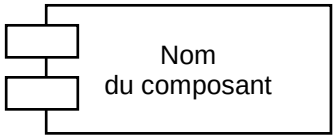
UML utilise un terme spécial pour représenter les entités physiques qui est composant. Le composant implémente un certain nombre d'interfaces et est utilisé pour désigner généralement les éléments de la représentation physique du modèle. Une icône spéciale peut être utilisée pour représenter graphiquement un composant - un rectangle avec deux rectangles plus petits sur la gauche. À l'intérieur du rectangle englobant se trouve le nom du composant et éventuellement des informations supplémentaires. L'apparence de ce symbole peut varier légèrement en fonction du type d'informations associées au composant.

Dans le métamodèle UML, un composant est un descendant d'un classificateur. Il fournit une organisation dans un package physique pour les éléments de modèle associés. En tant que classificateur, un composant peut également avoir ses propres propriétés telles que des attributs et des opérations.

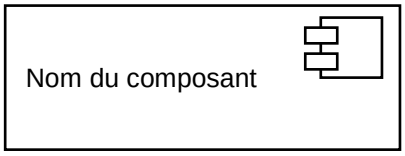
Notation graphique d'un composant

Le formalisme général d'un composant et les quelques manières de présenter un composant sont les suivantes :

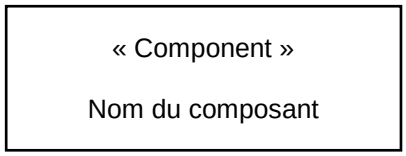
- **Première représentation :** Deux petits rectangles l'un au-dessus de l'autre chevauchant le côté gauche du grand rectangle et le nom du composant se trouve à l'intérieur du grand rectangle.



- **Deuxième représentation :** le  symbole en haut à droit et le nom du composant se trouve à l'intérieur du grand rectangle.



- **Troisième représentation :** le nom de stéréotype « Component » et le nom du composant se trouve à l'intérieur du grand rectangle.

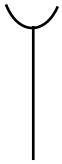


2.2.2. Les interfaces

Interface : ensemble d'opérations qui déclare le contrat de service fourni ou demandé par le composant. Ce contrat est respecté par les composants qui implémentent et utilisent cette interface.

Du point de vue de la subordination des appels, les interfaces peuvent être divisées en interfaces de demande et en interfaces de fourniture.

A. **Interface d'exigence :** également appelée interface requise, elle fait référence à l'interface qu'un composant doit suivre lors de la demande de services comme les autres composants. Elle est représentée graphiquement par un demi-cercle et raccordée au composant par une ligne continue.



B. **Interface fournie :** également appelée interface d'approvisionnement, elle fait référence aux caractéristiques et contraintes mises en œuvre lorsqu'un composant fournit des services à d'autres composants. Elle est représentée graphiquement par un cercle et raccordée au composant par une ligne continue.

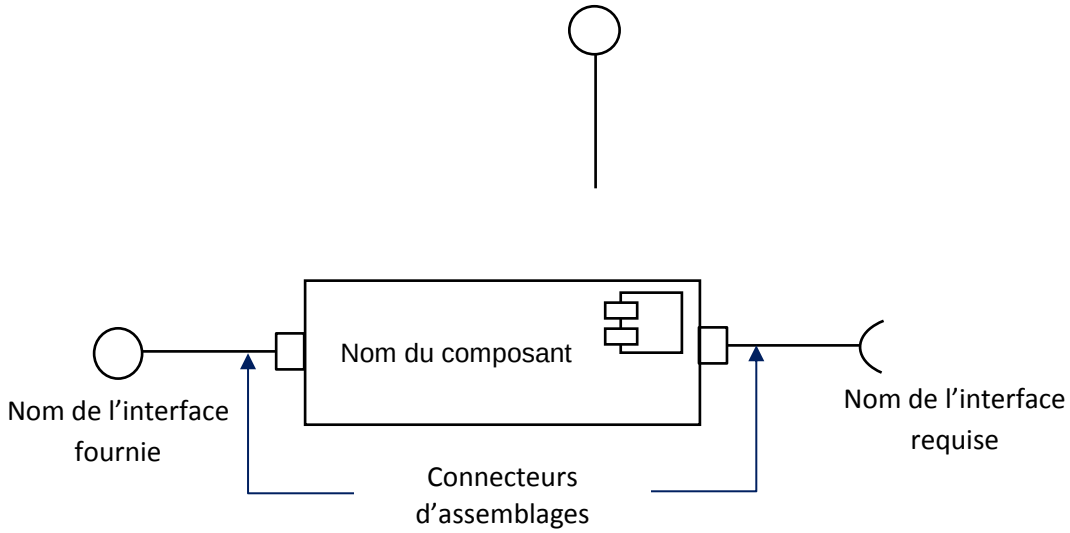


Figure 6.1 : Représentation graphique des interfaces avec des connecteurs d'assemblages.

2.2.3. Relation

Réalisation, dépendance, généralisation, agrégation, et composition.

2.2.4. Les ports

Il appartient à l'interface externe et est le point d'interaction entre le composant emballé et le monde extérieur. Le composant qui implémente l'interface utilise le port pour envoyer et recevoir des messages et interagir avec le monde extérieur. Un port est représenté

symboliquement par une petite boîte chevauchant la bordure du contour du classeur. Le nom du port peut apparaître près de sa représentation. C'est la matérialisation de l'interface.

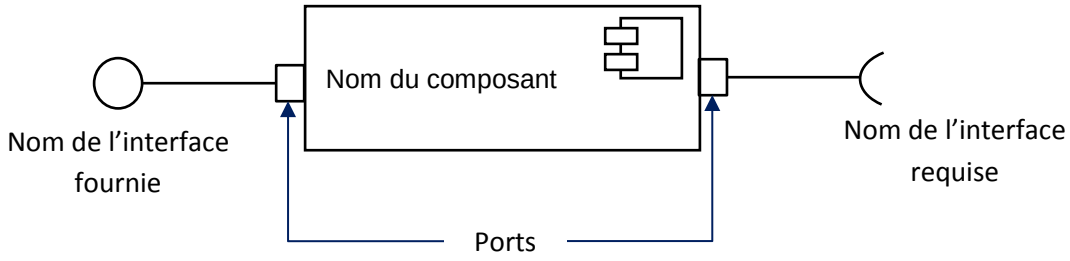


Figure 6.2 : Représentation des ports de connexion.

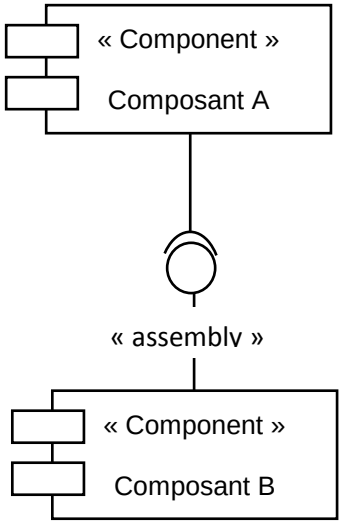
- **Relation avec l'interface :** l'interface d'exigence demande des services de l'extérieur via le port, et l'interface fournit des services vers l'extérieur via le port.
- **Relation avec les composants :** les composants peuvent interagir via des ports, tels que l'envoi et la réception de messages.

2.2.5. Les connecteurs

Connecteur : la relation de communication entre deux composants ou deux ports. Il existe deux types de connecteurs :

A. **Connecteur proxy :** le connecteur entre le port externe (port) et l'interface interne.

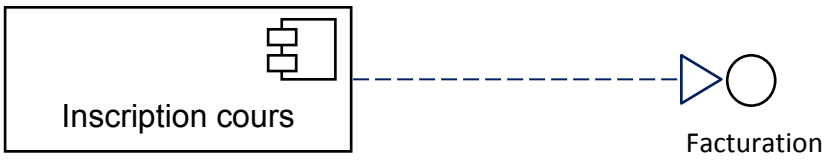
B. **Connecteurs assemblés :** connecteurs entre composants. Le connecteur établit une connexion entre l'interface de demande d'un composant et l'interface fournie d'un autre composant, de sorte que le premier composant puisse appeler le service fourni par le dernier composant.



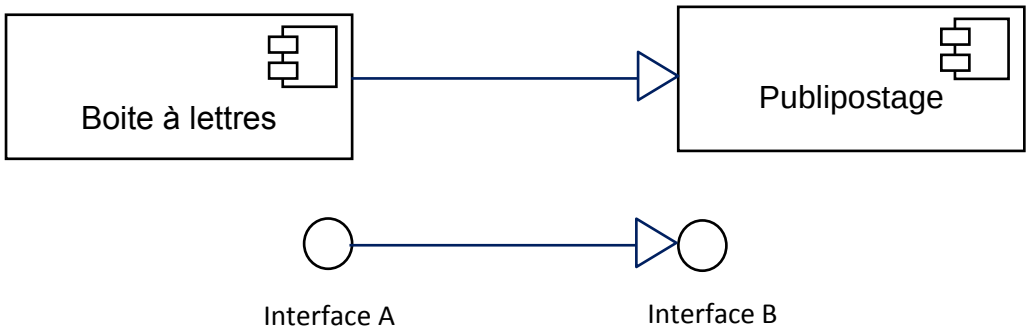
2.2.6. Relation dans les diagrammes de composants

Les relations dans le diagramme de composants incluent la réalisation, la dépendance et la généralisation, qui impliquent principalement entre les composants et les composants, entre les composants et les interfaces, et entre les interfaces et les interfaces.

A. **Réalisation :** la relation entre les composants et les interfaces.

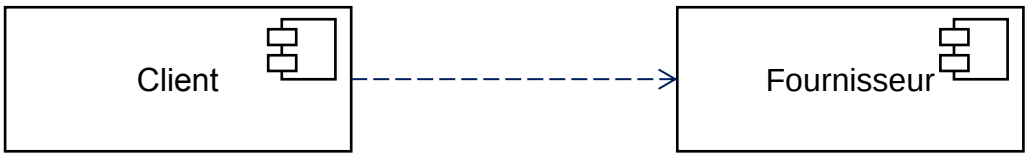


B. **Généralisation :** entre composants et entre interfaces
S'il existe une relation de généralisation entre deux classes dans deux composants ou entre deux interfaces, alors la relation entre ces deux composants peut-être exprimée comme une relation de généralisation.



C. Association : C'est l'agent susmentionné. Certains outils sont des lignes d'association sans flèches, et d'autres sont des lignes d'association directionnelles avec des flèches et des invites de délégation ou de délégation.

D. Dépendance : la relation entre les composants et les composants
Si deux classes dans deux composants ont une relation de dépendance, alors la relation entre les deux composants peut être exprimée comme une relation de dépendance.



3. Diagramme de déploiement

3.1 Objectif du diagramme de déploiement

Le diagramme de déploiement est utilisé pour montrer l'architecture physique du logiciel et du matériel dans le système. À partir de ce diagramme, on peut comprendre la relation physique entre les composants logiciels et matériels et la distribution des composants des nœuds de traitement. Autrement dit, le diagramme de déploiement décrit les nœuds matériels du système

et la manière dont les nœuds sont connecté. Pareillement, il montre le matériel physique exécutant le système logiciel et comment déployer le logiciel sur le matériel.

3.2 Les éléments du diagramme de déploiement

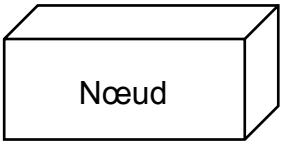
Le diagramme de déploiement se compose de deux parties : les nœuds et les connexions.

3.2.1 Les nœuds

Les nœuds représentent des ressources matérielles dans un système informatique d'exécution. Les nœuds ont généralement de la mémoire et une puissance de traitement. Par exemple, un ordinateur, une station de travail et d'autres périphériques informatiques sont tous des nœuds. Ils pourraient être connectés via des voies de communication pour créer des systèmes de réseau d'une complexité arbitraire. Les ressources matérielles sont parfois symbolisées avec le stéréotype « device ».

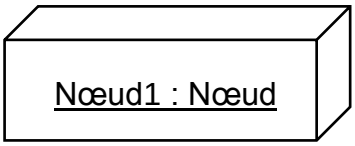
Notation graphique

En UML, un nœud est représenté par un cube. Chaque nœud doit avoir un nom qui le distingue des autres nœuds. Le nom du nœud est une chaîne, située à l'intérieur de l'icône du nœud.



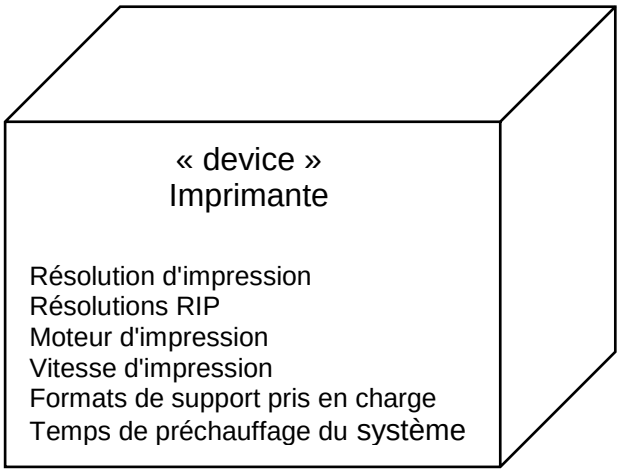
Instance de Nœuds

La notation UML de l'instance de nœuds dans un diagramme de déploiement d'objets est très simple. Une instance d'un nœud est un parallélépipède rectangle avec le nom de l'instance et le nom de nœud séparés par le signe « : » et soulignés.



Un nœud possède des attributs qui peuvent être précisé à l'intérieur du cube.

Exemple :



Il existe deux possibilités pour montrer qu'un composant est affecté à un nœud. La première consiste à mettre le composant dans le nœud, et la seconde consiste à relier le composant et le nœud par une relation de dépendance stéréotypée « support », tout en orientant la direction de la flèche vers le nœud.

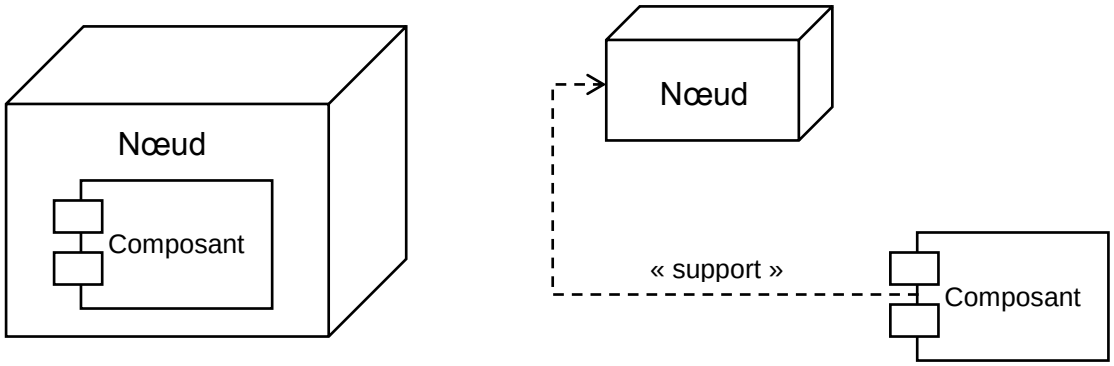


Figure 6.3 : Représentation de l'affectation d'un composant à un nœud dans deux possibilités.

3.2.2 Les supports de communications

Le support de communication est un lien qui permet de modéliser de manière simpliste la communication entre deux nœuds. Il est a priori bidirectionnel. Il est également possible d'exprimer sur le lien de communication qui relie deux nœuds : (i) des multiplicités, (ii) des contraintes entre accolades, et (iii) et la nature de liaison entre nœuds qui peut être précisée par un stéréotype placé à proximité de ce lien.

Exemple :

Figure 6.4 : Représentation de support de communication entre deux nœuds.

3.2.3. Les artefacts

Un artefact représente un élément concret existant dans le monde réel tel qu'un document, un exécutable, un fichier, des tables de base de données, un script, etc.

Représentation graphique

Un artefact peut être formalisé comme un classeur par un rectangle contenant le stéréotype « artifact » avec son nom juste en dessous et/ou une icône particulière dans le coin droit du rectangle.

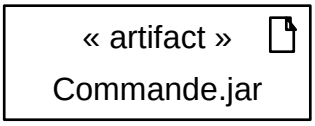


Figure 6.5 : Représentation d'un artefact.

L'implémentation des modèles (classes, etc.) s'effectue sous la forme d'un ensemble d'artefacts. Nous disons qu'un artefact peut manifester, c'est-à-dire résulter et implémenter, un ensemble d'éléments de modèle. La relation entre un élément de modèle et l'artefact qui l'implémente s'appelle une manifestation.

Représentation graphique

Une manifestation peut se représenter symboliquement par une relation de dépendance stéréotypée « manifest ».

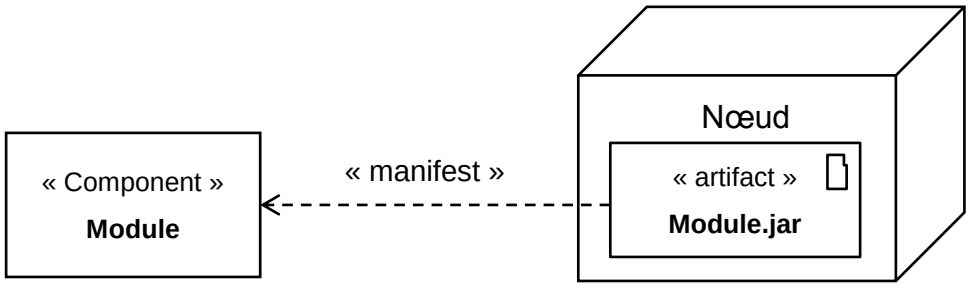


Figure 6.6 : Représentation du déploiement dans un nœud d'un artefact manifestant un composant.

Une instance d'un artefact se déploie sur une instance de nœud. Graphiquement, nous utilisons une relation de dépendance stéréotypée « déploy » désignant le nœud en question.

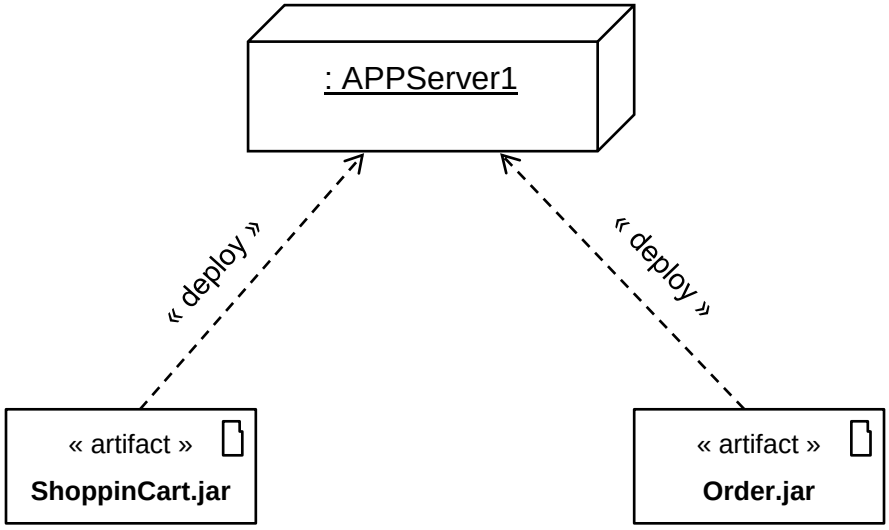


Figure 6.7 : Représentation du déploiement de deux artefacts dans un nœud utilisant la relation de dépendance stéréotypée « deploy ».

L'artefact peut également être inclus directement dans le cube représentant le nœud. À proprement parler, seuls les artefacts doivent être déployés sur les nœuds. Un composant doit donc se manifester par un artefact qui, lui-même, peut être déployé sur un nœud.

4. Diagramme de structure composite

4.1. Définition

Le diagramme de structure composite décrit la structure interne du classificateur, à savoir ses points d'interaction avec d'autres parties du système. De plus, il montre la configuration et la relation de chaque partie, qui ensemble, exécutent le comportement du classificateur contenant.

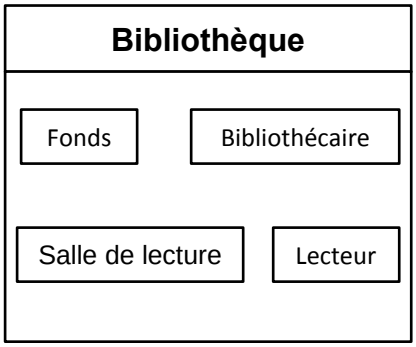
4.2. Les objets du diagramme de structure composite

Les éléments principaux du diagramme de structure composite sont les suivants :

4.2.1. Classificateur

Un classificateur est un concept abstrait de classification de métaclasse qui sert de mécanisme pour afficher les interfaces, les classes, les types de données et les composants. Il décrit un ensemble d'instances qui ont des caractéristiques comportementales (opérations) et structurelles (attributs) communes. Un classificateur est un espace de noms dont les membres peuvent spécifier une hiérarchie de généralisation en référençant ses classificateurs généraux. De plus, il est un type et peut posséder des généralisations, ce qui permet de définir des relations de généralisation à d'autres classificateurs. Enfin, un classificateur est un élément redéfinissable, car il est possible de redéfinir des classificateurs imbriqués.

Exemple :



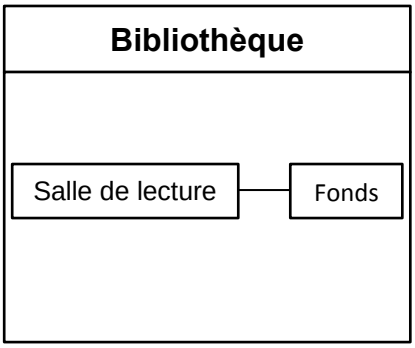
Remarque

Tous les objets qui peuvent avoir des instances sont des classificateurs.

Classificateur structurés

Il représente une classe, qui est fréquemment abstraite, dont le comportement peut être entièrement ou partiellement décrit par des interactions entre les parties.

Exemple :

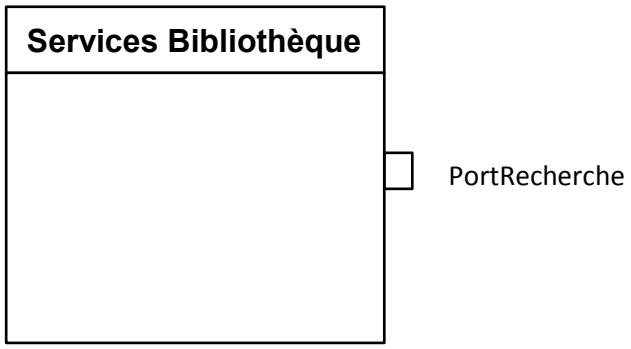


Classificateur encapsulé

Un classificateur encapsulé est une forme de classificateur structuré qui contient des ports. Ces derniers peuvent être utilisés pour isoler le classificateur de son environnement. Par conséquent, tout port précise un point d'interaction existant entre le classificateur et son environnement.

Exemple :

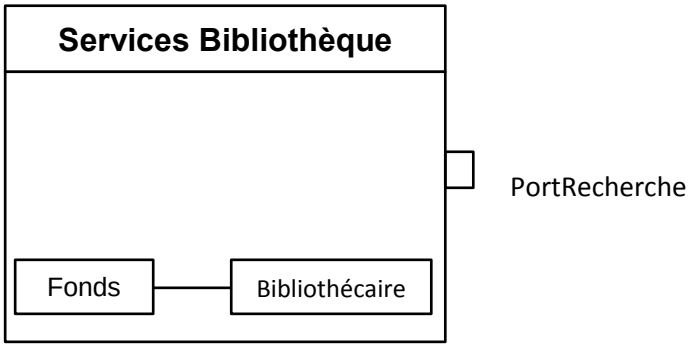
Dans cet exemple, le classificateur « services Bibliothèque » est encapsulé via le port de recherche.



Classe structurée

La classe structurée est formalisée en tant que classe, classe structurée ou classe encapsulée. C'est pourquoi elle peut posséder des ports et de la structure interne.

Exemple :

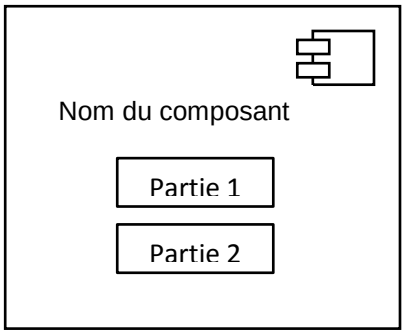


4.2.2 Partie

Une partie permet de définir une zone bien délimitée à l'intérieur d'une classe ou d'un composant. Une partie est un élément qui représente un ensemble d'une ou plusieurs instances appartenant à une instance de classificateur contenant. Ainsi, par exemple, si une instance de diagramme possédait un ensemble d'éléments graphiques, les éléments graphiques pourraient être représentés comme des parties ; s'il était utile de le faire, de modéliser une sorte de relation entre eux. Les parties peuvent être raccordées aux autres parties ou ports, soit directement, soit à travers un port situé à l'extérieur de la partie. Les parties sont raccordées entre elles via des connecteurs d'assemblage. Par contre, elles sont raccordées aux ports situés à l'extérieur d'une classe ou d'un composant via un connecteur de délégation.

Une partie peut être supprimée de son parent avant que le parent ne soit supprimé, de sorte que la partie ne soit pas supprimée en même temps.

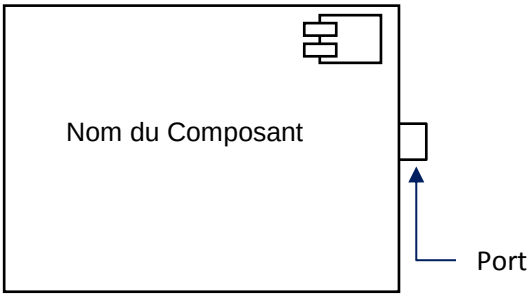
Représentation graphique
Une partie se représente symboliquement par un rectangle contenu dans le corps d'une classe ou d'un élément de composant.



4.2.3 Port

Un port est un élément tapé, qui représente une partie visible de l'extérieur d'une instance de classificateur contenant. Ports définissent l'interaction entre un classificateur et son environnement. Un port peut apparaître sur la frontière d'une partie contenue, comme une classe ou une structure composite. Un port peut préciser les services fournis par un classificateur, ainsi que les services qu'elle exige de son environnement.

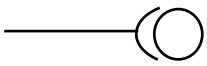
Représentation graphique d'un port
Un port est représenté par un rectangle nommé sur le bord de limite de son classificateur propriétaire.



4.2.4 Les connecteurs

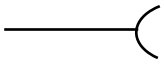
Le connecteur spécifie un lien qui permet la communication entre au moins deux instances.

A. **Connecteurs d'assemblage :** Il permet de raccorder les parties entre elles.



B. **Connecteur de délégation :** Il permet de raccorder les parties aux ports à l'extérieur des classificateurs.

C. **Lien de prérequis :** Il permet de raccorder les classes, les composants, les ports, les parties et les interfaces.



4.3. Collaboration

Une collaboration représente un ensemble de rôles d'éléments coopérants utilisés collectivement en vue de réaliser une fonction donnée. Une collaboration ne doit afficher que les rôles et attributs requis pour exécuter sa fonction définie. Une collaboration peut être représentée selon deux possibilités :

a) Représentation par une collaboration de rôles

Une collaboration de rôles est symboliquement représentée par un ovale en pointillé dans lequel apparaissent les rôles des éléments qui interagissent pour réaliser la fonction souhaitée.

Exemple :

Cet exemple montre la fonction "Persistance des objets métier" qui résulte d'une collaboration entre deux rôles d'éléments, à savoir "mapping : classeMétier" et "stockage : tableBDD".

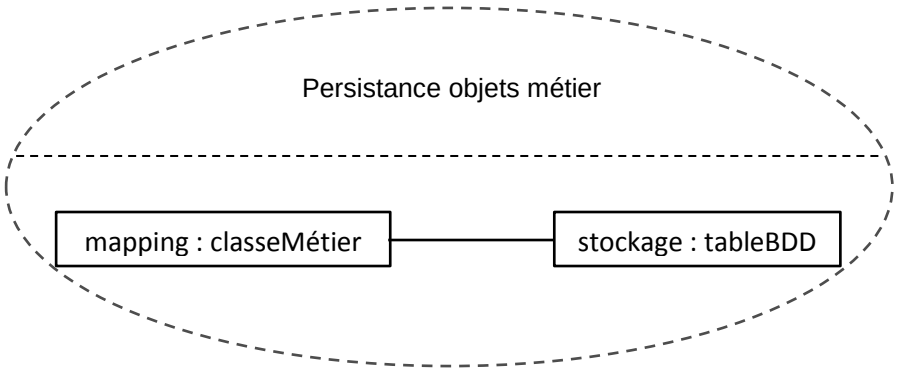


Figure 6.8 : Représentation d'une structure composite à l'aide d'une collaboration de rôles**.**

b) Représentation par une structure composite

Une collaboration par une structure composite est symboliquement représentée par un ovale en pointillé. Les éléments participant à la collaboration, y compris la classe, le composant, etc. apparaissent en dehors de la collaboration. En revanche, les rôles considérés dans chaque participation sont représentés sur les liens entre les éléments participants et la collaboration. Enfin, la spécification des uniques attributs des classes participantes qui sont considérés en fonction des rôles pris en compte est assurée par ce type de représentation.

Exemple :

Cet exemple représente la fonction "Persistance objets métier" qui découle d'une collaboration entre la classe "ClasseMétier" qui est considérée suivant le rôle "mapping" et la classe TableBDD qui considérée suivant le rôle "stockage".

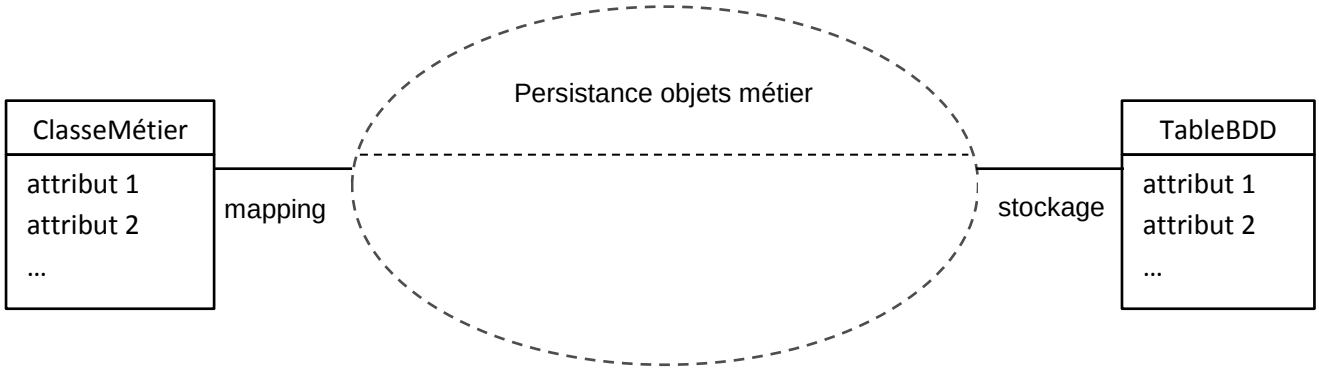


Figure 6.9 : Représentation de collaboration d'instances par un diagramme de structure composite.

5. Mécanismes d'extension

La modélisation logicielle est traditionnellement devenue synonyme de génération de diagrammes. La plupart des modèles consistent en une série de séquences et de flèches. Les informations véhiculées par un tel modèle ont tendance à être incomplètes, irrégulières, imprécises et parfois même instables. Par conséquent, l'un des objectifs de la modélisation logicielle est de créer un modèle précis et suffisamment standardisé. Afin de construire un modèle plus précis, aussi proche que possible de l'activité réelle associée, nous devons généralement ajouter des contraintes.

Bien que l'UML fournisse divers concepts génériques pour la modélisation de logiciels et de systèmes, il ne peut pas couvrir tous les scénarios d'application imaginables prêts à l'emploi. Au lieu de cela, il peut être étendu avec des profils pour ajouter des éléments de modèle personnalisés en fonction de vos besoins.

5.1 Langage de contrainte OCL

5.1.1 Définition

Le langage OCL (Object Constraint Language), est un moyen d'indiquer les restrictions dans un système de modélisation utilisateur. Il s'agit d'un contenu supplémentaire facultatif de langage UML qui peut être utilisé pour mieux définir le comportement de l'objet et spécifier des contraintes pour tout élément de classe. En d'autre termes, le langage de contraintes objet a été créé afin de résoudre le problème d'expression des contraintes. Il a été initialement conçu avec succès par l'entreprise IBM (International Business Machines Corporation), et a été accepté par l'organisation de normalisation OMG et fait désormais partie de la norme UML. Le langage OCL est un langage formel, a les caractéristiques de non-ambiguïté du langage formel, qui élimine la complexité du langage formel.

Dans le langage OCL, l'objet représente le composant du système, il définit un projet complet et la contrainte représente la restriction, et le langage ne fait pas référence à un langage informatique formel. OCL est un langage formel, qui peut être appliqué à n'importe quel langage informel de mise en œuvre. Ce langage n'a aucun contrôle sur les diagrammes ou d'autres composants dans UML, il renvoie simplement des valeurs lorsqu'il est utilisé. De plus,

il ne modifie pas l'état d'un objet, mais est utilisé pour indiquer quand la modification de l'état se produit.
Les expressions OCL expriment des contraintes sur l'objet avec des conditions et des restrictions attachées aux éléments de modèle, y compris des expressions d'invariants ou de contraintes attachées aux éléments de modèle, des pré-conditions et post-conditions attachées aux opérations et aux méthodes, etc.

5.1.2. Comment utiliser le langage de contrainte OCL

Le langage de contrainte OCL peut être utilisé avec des objectifs différents dans plusieurs cas à savoir : (1) pour spécifier des invariants sur des classes ou des types dans un modèle de classes, (2) pour spécifier un type invariant pour un stéréotype, (3) pour décrire des pré et post conditions sur des opérations et des méthodes, (4) pour décrire les gardes, (5) l'utiliser comme langage de navigation, et (6) pour spécifier les contraintes sur les opérations.

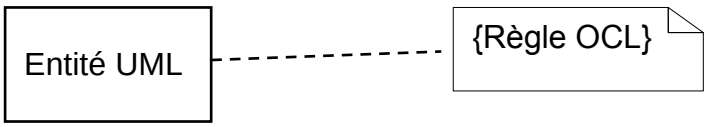
5.1.3. Typologie des contraintes

5.1.3.1 Contexte

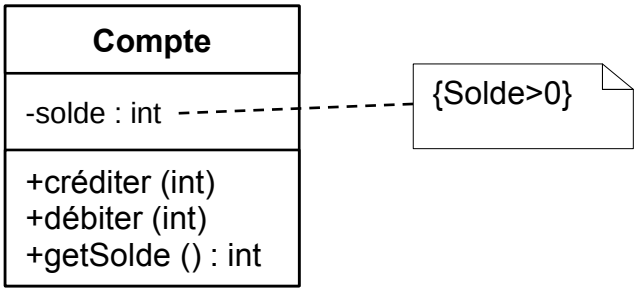
Chaque contrainte OCL est liée à un contexte définissant l'élément de modélisation auquel la contrainte se rapporte. La spécification de contexte d'une contrainte OCL se fait en deux manières :

- **De manière graphique :** en écrivant la contrainte entre accolades {} dans une note.

Notation :



Exemple :



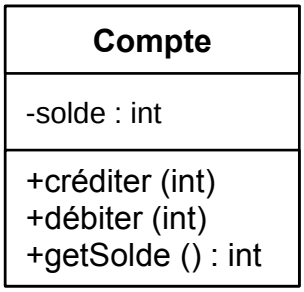
- **De manière textuelle :** en utilisant le mot-clef **context** dans un document accompagnant le diagramme. La syntaxe est la suivante :

  context
  <élément>

  <élément> peut être une classe, une opération, etc.

Pour faire référence à un élément op (comme une opération) d'un classeur C (comme une classe), ou d'un paquetage… il faut utiliser les :: comme séparateur (comme C::op).

Exemple :



Le contexte s'écrit de la manière suivante :

1. Le contexte est la classe Compte :
   context Compte

2. Le contexte est l'opération **getSolde()** de la classe Compte :
   context Compte::getSolde()

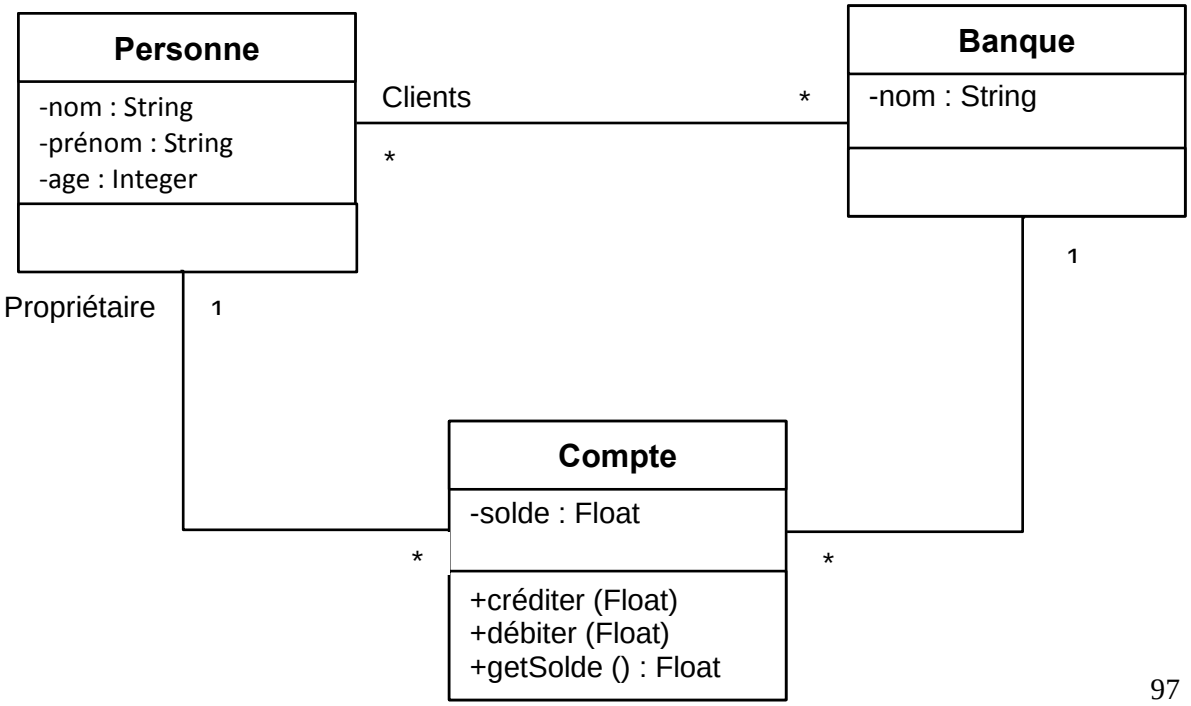
5.1.3.2 Invariants

Un invariant exprime une contrainte (expression booléenne, sans effet de bord) prédicative sur un objet, ou un groupe d'objets, qui doit être respectée en permanence par toutes les instances de la classe ou du type concerné. La syntaxe est la suivante :

Context entitéUML **inv:**
<Expression OCL>

Où, <Expression OCL> est une expression logique qui doit toujours être vraie.

Exemples :
Soit le diagramme de classe suivant :



1. Le solde d'un compte doit toujours être positif :

   context Compte
   inv: solde > 0

2. L'âge est compris entre 0 et 120 ans

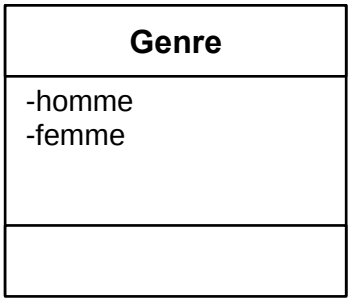
   context Personne
   inv: (age >=0) and (age <= 120)

3. L'âge est compris entre 0 et 120 ans

   context Personne
   inv: (**self.**age >=0) and (**self.**age <= 120)

   context p:Personne
   inv borneAge: (p.age >=0) and (p.age <= 120)

4. Les hommes (au sens de l'association) des personnes doivent être des hommes (au sens du genre).



   context Personne
   inv : homme->forAll(genre=Genre::homme)

Remarque

Le point (**.**) permet d'accéder à une caractéristique d'un objet (méthode, attribut, terminaison d'association) et **self** objet désigné par le contexte.

5.1.3.3 Préconditions et postconditions

Une précondition et une postcondition permettent de spécifier une contrainte prédicative qui doit être vérifiée avant (respectivement après) l'appel d'une opération. En d'autres termes, elles sont des contraintes qui doivent être respectées respectivement avant ou après l'exécution d'une opération.

La syntaxe de précondition est la suivante :

context typeName::opérationName(param1: type1, . . .):ReturnType
pre: <Expression OCL>

La syntaxe de postcondition est suivante :

context typeName::opérationName(param1: type1, . . .):ReturnType
post: <Expression OCL>

Où, <Expression OCL> est une expression logique qui doit toujours être vraie.

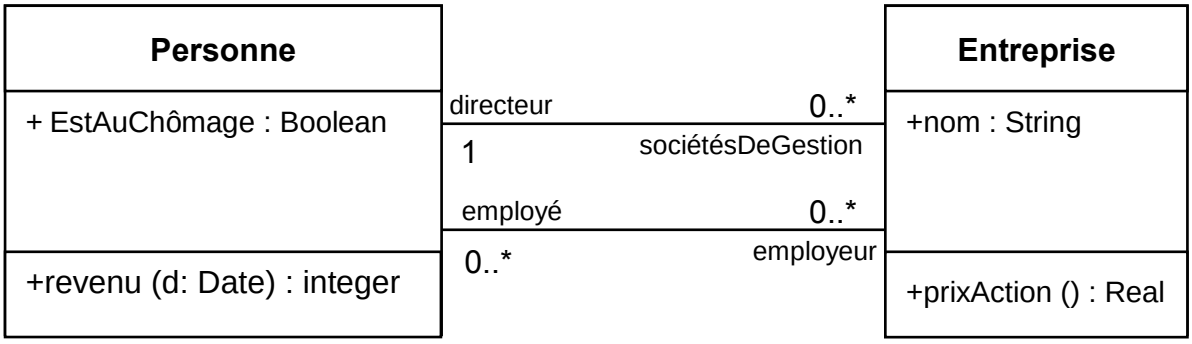
Exemple :

1. Pour que l'appel à l'opération "debiter (somme)" soit valide, la somme à débiter doit être positive. De plus, après avoir exécuté cette opération, la valeur de l'attribut "solde" doit être la différence entre sa valeur avant l'appel et la somme passée en paramètre.

   context Compte::débiter(somme : Real)
   pre : somme > 0
   post : solde = solde**@pre** – somme

2. L'appel de l'opération "getSolde()" doit retourner un résultat égal à l'attribut solde :

   context Compte::getSolde() : Real
   post : result = solde



4. L'appel de l'opération "revenu()" doit retourner un résultat égal à 2500 :
   context Personne::revenu(date:Date):integer
   post: result = 2500

5.1.4. Écriture d'une expression

5.1.4.1 Types de base et opérations associées

Type	Exemples	Opérateurs
Boolean	true ; false	and ; or ; xor ; not ; implies ; if-then-else-endif ; etc.
Integer	1 ; -5 ; 3 ; 97 ; 5 ; 52743 ; etc.	* ; + ; - ; / ; abs() ; max() ; etc.
Real	1,5 ; 3,14 ; -1.816 ; etc.	* ; + ; - ; / ; abs() ; floor() ; etc.
String	‘’Object Constraint Language’’	concat() ; size() ; toUpper() ; substring() ; etc.

Table 6.10 : Types et opérations prédéfinis dans les contraintes OCL.

Opérateurs conditionnels

Certaines contraintes sont dépendantes d'autres contraintes. Il existe deux formes pour gérer cette situation :

- **if** <expression_1> **then** < expression_2> **else** < expression_3> **endif**
  Si l'expression <expression_1> est vraie alors <expression_2> doit être vraie sinon <expression_3> doit être vraie.

- <expression_1> implies <expression_2>
  Si l'expression <expression_1> est vraie, alors <expression_2> doit l'être également.
  Si <expression_1> est fausse, alors l'expression complète est vraie

Exemples :

context Personne
inv : if age >=22 then
majeur=vrai
else majeur=faux endif

context Personne
inv : marié **implies** majeur

Définir plusieurs contraintes pour un invariant, une pré ou postcondition :

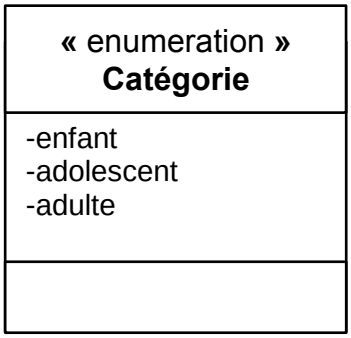
- and : " et logique" : l'invariant, pré ou postcondition est vrai si toutes les expressions reliées par le '**'and''** sont vraies.
- or et xor

Types énumérés

Une énumération est un type de données UML, ayant un nom, et utilisé pour énumérer un ensemble de littéraux correspondant à toutes les valeurs possibles qu'une expression de ce type peut prendre. Un type énuméré est défini par un classeur ayant le stéréotype «énumération». Pour éviter les conflits de nom, nous utilisons le nom d'énumération : **Enum::valeur**. Auparavant, leurs valeurs apparaissaient précédées du symbole "#" (exemple, **#valeur**).

Exemples :

Considérons la classe suivante :



context Personne
inv : if age <=13 then cat =Catégorie::enfant

else if age <=18 then cat =Catégorie::adolescent
else cat=Catégorie::adulte
endif endif

context Personne
inv: genre = Genre::femme

Types reliés par une relation de spécialisation

Les types de modèles qui peuvent être utilisés dans OCL sont des «classificateurs», en particulier des classes, des interfaces et des associations. En particulier, nous pouvons écrire des expressions impliquant des objets dont les types sont liés par une relation de spécialisation, grâce aux opérateurs suivants :

- **oclIsKindOf(type)** : vrai si l'objet est du type type ou un de ses sous-types
- **oclIsTypeOf(type)** : vrai si l'objet est du type type
- **oclAsType(type)** : l'objet est <casté> en type type

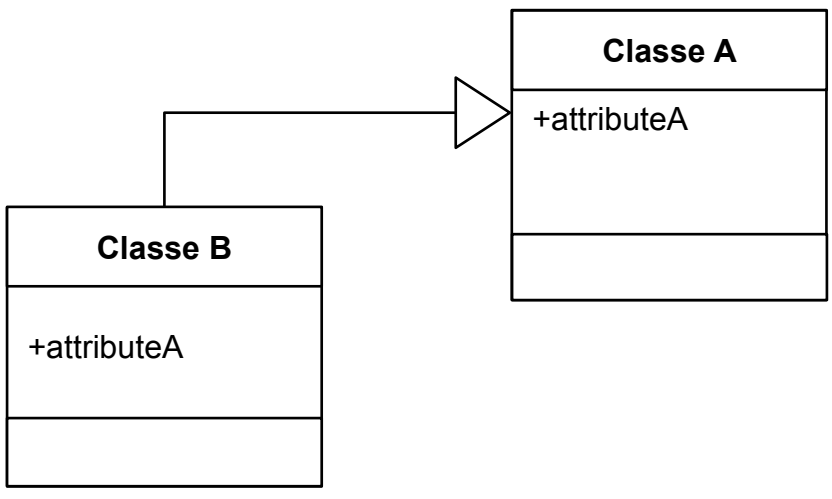
Exemple 1 :
Une classe B hérite d'une classe A :

- Dans le contexte de classeB, self.oclIsKindOf(ClasseB) et self.oclIsKindOf(ClasseA) sont vraies ;
- Dans le contexte de classeA, self.oclIsKindOf(ClasseB) est fausse ;
- Dans le contexte Société, directeur.oclIsTypeOf(Personne) est vraie alors que self.oclIsTypeOf(Personne) est fausse.

Exemple 2 :
Accès aux propriétés des supertypes

Context B **inv:**
Self.oclAsType(A).attributeA
Self.attributeA

Nous avons accès à la propriété définie dans la classe A. Ensuite, nous accédons à la propriété définie dans la classe B.



Types de collection

Comme le montre la Figure 6.10, le langage de contrainte OCL définit les quatre (4) types de collections suivants :

- **Set** : est un ensemble non ordonné d'éléments uniques.
  Exemple : {2, 3, 1, 9}

- **OrderedSet** : ensemble ordonné d'éléments uniques ({ordered}).
  Exemple : {1, 2, 4, 78}

- **Bag** : collection non ordonnée d'éléments identifiables. En d'autres termes, cette collection accepte plusieurs mêmes éléments, les doublons sont admis.
  Exemple : {2, 2, 5, 3, 3, 9}

- **Sequence** : collection ordonnée d'éléments identifiables. En d'autres termes, Cet collection peut comporter des doublons et tous ses éléments sont ordonnés.
  Exemple :{2, 2, 3, 3, 5, 9}

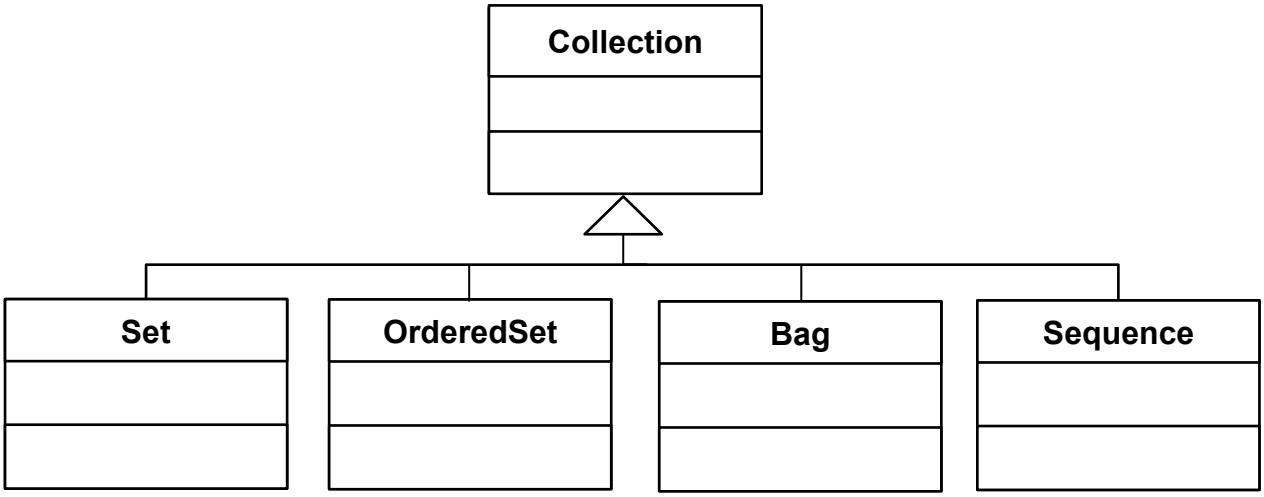


Figure 6.10 : Types de collections.

Remarque

Toutes ces classes sont des sous-types du type abstrait Collection.

5.1.4.2 Opérateur '@pre'

L'opérateur '**@pre**' est utilisé pour spécifier les pre et post-conditions sur les opérations et méthodes dans UML. L'opérateur '**@pre**' est postfixé et indique la valeur de la propriété au début de l'opération.

Exemple :

La valeur de l'attribut "age" doit être la somme entre sa valeur avant l'appel de l'opération AnniversaireArrive(age) et une année.

context Person:: AnniversaireArrive()

post: age=age**@pre** + 1

5.1.4.3 Opérations sur objets et collections

Langage de contrainte OCL utilise un ensemble de primitives utilisables sur les collections, à savoir :

- **size()** : retourne le nombre d'éléments de la collection.
- **isEmpty()** : retourne vrai si la collection est vide.
- **notEmpty()** : retourne vrai si la collection n'est pas vide.
- **includes(obj)** : vrai si la collection inclut l'objet obj.
- **excludes(obj)** : vrai si la collection n'inclut pas l'objet obj.
- **including(obj)** : la collection référencée doit être cette collection en incluant l'objet obj.
- **excluding(obj)** : idem mais en excluant l'objet obj.
- **includesAll(ens)** : la collection contient tous les éléments de la collection ens.
- **excludesAll(ens)** : la collection ne contient aucun des éléments de la collection ens.

Remarque

Self : pseudo-attribut référençant l'objet courant.

Exemples :

- propriétaire -> notEmpty() : il y a au moins un objet Personne associé à un compte
- propriétaire -> size() = 1 : le nombre d'objets Personne associés à un compte est de 1
- banque.clients -> size() >= 1 : une banque a au moins un client
- banque.clients -> includes(propriétaire) : le propriétaire du compte (l'ensemble des clients de la banque associée au compte)
- banque.clients.compte -> includes(self) : le compte appartient à un des clients de la banque du client en question.

5.1.4.4 Sélection dans un sous ensemble

select(col) : retourne le sous-ensemble de la collection col dont les éléments respectent la contrainte spécifiée. La syntaxe de cette opération se présente comme suit :

Collection->select(…)

Exemple :

context Entreprise **inv:**
self.employé**->select(age>50)->notEmpty()**

Remarque

L'opérateur -> est utilisé pour appeler les opérations définies sur les collections.

5.1.4.5 Rejet d'un élément d'une collection

reject(col) : idem mais ne garde que les éléments ne respectant pas la contrainte. La syntaxe de cette opération se présente comme ci-dessous :

Collection->reject(…)

Exemple :
context Company **inv:**
self.employee**->reject(**isMarried**)->isEmpty()**

5.1.4.6 Existence

exists (col): retourne vrai si au moins un élément de col respecte la contrainte spécifiée et faux sinon. La syntaxe de cette opération se présente comme ci-dessous :

Collection->exists(…)

Exemple :

context Entreprise **inv:**
self.employé->**exists**(forename =‘Jack’)

5.1.4.7 Iteration

L'opération iterate est légèrement plus compliquée mais très générique. Toutes les autres opérations (select, reject, forAll, exists, collect) peuvent être exprimées sous forme de l'opération iterate. La syntaxe de cette opération se présente comme ci-dessous :

collection -> **iterate**(elem: Type; acc: Type = <Expression>|Expression_avec_elem_acc)

La variable **elem** est un itérateur, comme dans la définition de **select**, **forAll**, etc. La variable **acc** est un accumulateur. L'accumulateur a une valeur initiale **<Expression>**. Quand l'opération **iterate** est évaluée, **elem** itère sur toute la collection et l' Expression_avec_elem_acc est évaluée pour chaque **elem**. De cette façon, la valeur de **acc** s'est accumulée pendant l'itération de la collection.

Exemple :

self.employee->**iterate**(p:Person;acc=Bag{}
|acc->including (p.forename<>’Jack’)

5.1.4.8 Définition d’attributs et de méthodes (def et let…in)

Parfois, une sous-expression est utilisée plusieurs fois dans une expression. let permet de déclarer et de définir la valeur (i.e. initialiser) d’un attribut qui pourra être utilisé dans l’expression qui suit le in.
def est un type de contrainte qui permet de déclarer et de définir la valeur d’attributs comme la séquence let…in. def permet également de déclarer et de définir la valeur retournée par une opération interne à la contrainte. La syntaxe de let…in est la suivante :

let <déclaration> = <requête> **in** <expression>

Où, un nouvel attribut déclaré dans <déclaration> aura la valeur retournée par l’expression <requête> dans toute l’expression <expression>.

Exemple :

```
context Person inv:
  let income: Integer=self.job.salary-> sum()
  let hasTitle(t: string):Boolean=
                        self.job->exists(title=t) in
  if isUnemployed then
        self.income <100
  else
        self.income>=100 and
   self.hasTitle('manager')
  endif
```

Syntaxe de def est la suivante :

def : <déclaration> = <requête>
Où, <déclaration> peut correspondre à la déclaration d'un attribut ou d'une méthode. <requête> est une expression qui retourne un résultat dont le type doit être compatible avec le type de l'attribut, ou de la méthode, déclaré dans <déclaration>. Dans le cas où il s'agit d'une méthode, <requête> peut utiliser les paramètres spécifiés dans la déclaration de la méthode

Exemple :
Pour imposer qu'une personne majeure doit avoir de l'argent, nous pouvons écrire indifféremment :

1. **context** Personne
   inv : **let** argent=compte.solde->**sum() in** age>=18 **implies** argent>0

2. **context** Personne
   def : argent : int = compte.solde->sum()
   context Personne
   inv : age>=18 **implies** argent>0

5.1.4.9 Accès aux membres d'un objet

L'accès à un membre (attribut ou méthode) d'un objet donné se fait à l'aide de la notation à point. Cette notation possède deux parties séparées par un point : à gauche du point, on trouve la référence de l'objet et à sa droite le membre :
NomDeObjet.NomAttribut
Ou NomDeObjet.NomMethode( )

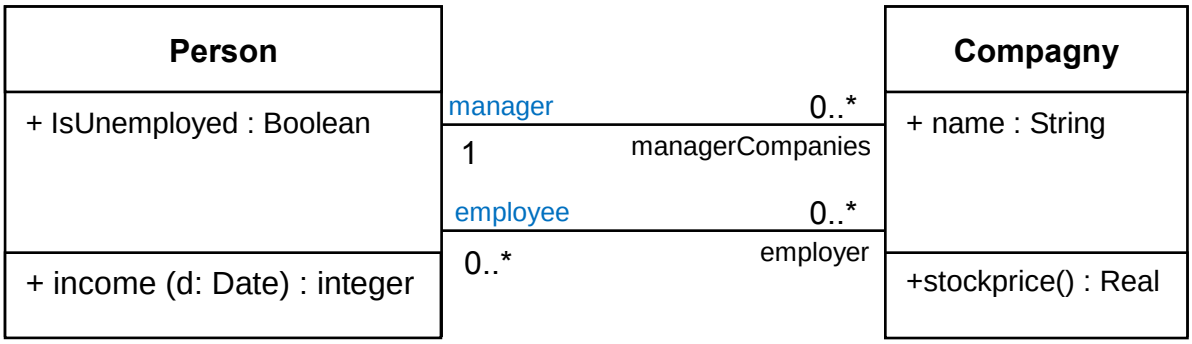
1. **Accès à un attribut :**
   Exemple :

```
context Person inv:
self.age > 0
```

2. **Accès à une opération :**
   Exemple :
   aPerson.income(aDate)

5.1.4.10 Accès à une fin d'association

La syntaxe d'accès à une fin d'association se présente comme ci-dessous :
Objet.nomDuRôleOpposé



context Company
inv: self.manager.isUnemployed = **false**
inv: self.employee->notEmpty()

context Person **inv:**
self.employer->isEmpty()
//Ensemble de personnes qui ont le rôle employeur

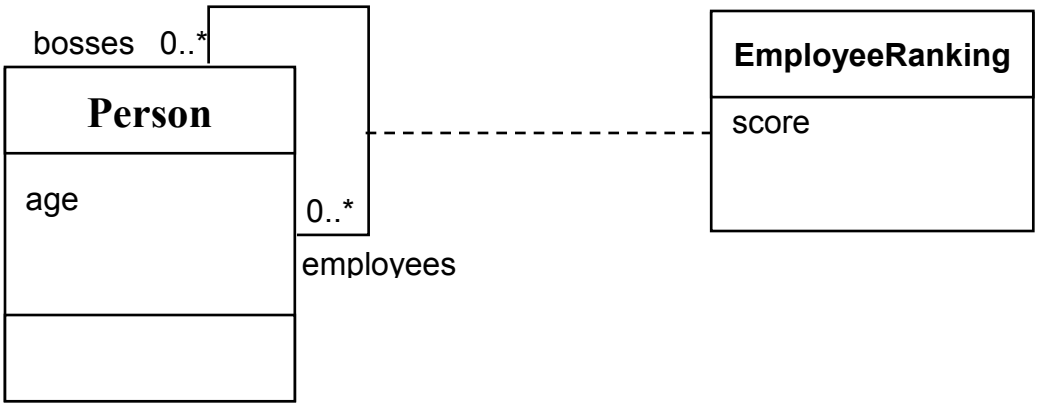
5.1.4.11 Navigation vers une classe association

Pour naviguer vers une classe association, il faut utiliser la notation pointée classique en précisant le nom de la classe association en minuscules. Par exemple, dans le contexte de la classe Société (context Société), pour accéder au salaire de tous les employés, il faut écrire :
self.poste.salaire

Cependant, dans le cas où l'association est réflexive (c'est le cas de la classe association Mariage), il faut en plus préciser par quelle extrémité il faut emprunter l'association. Pour cela, on précise le nom de rôle de l'une des extrémités de l'association entre crochets ([]) derrière le nom de la classe association. Par exemple, dans le contexte de la classe Personne (context Personne), pour accéder à la date de mariage de toutes les femmes, il faut écrire :

self.mariage[femme].date

Exemple :



Context Person **inv:**
  self.employeeRanking[bosses]->sum() > 0
→ Ensemble des « employeeRanking » qui appartiennent à la collection « bosses »

5.2 Les profils

Dans le langage UML un profil fournit un mécanisme d'extension générique, dont l'objectif est de personnaliser les modèles UML pour des domaines et des plates-formes spécifiques. En effet, la transition vers un profil UML est assurée par les mécanismes d'extension fournis par UML 2.0 afin d'améliorer la sémantique standard de façon strictement additive, les empêchant de contredire la sémantique standard.

Les profils sont définis en fonction de stéréotypes, de définitions de balises et de contraintes qui sont appliqués à des éléments de modèle spécifiques y compris les classes, les attributs, les opérations et les activités. De plus, divers profils UML, qui sont un ensemble d'extensions de ce type, ont été émis par l'OMG afin de personnaliser UML pour des domaines applicatifs spécifiques tels que la santé, l'aérospatiale, les transports, la finance, l'ingénierie des systèmes ou une plate-forme, à savoir EJB, J2EE, et .NET.

6. Conclusion

Ce chapitre introduit d'autres notions et diagrammes UML. En résumé : Le diagramme de composants est un diagramme UML spécial pour décrire la vue de réalisation statique du système. Ce composant comprend des composants physiques tels que des bibliothèques, des fichiers, des dossiers, etc. de plus, il est utilisé dans une perspective d'implémentation. L'utilisation de techniques d'ingénierie directe et inverse permet de créer des diagrammes de composants de fichiers exécutables.

Le diagramme des composants est utilisé pour décrire la vue de déploiement statique d'un système. Ces schémas sont principalement utilisés par les ingénieurs système. Le diagramme de déploiement est composé de nœuds et des relations entre eux. Un diagramme de déploiement efficace fait partie intégrante du développement de logiciels d'application.

Le langage de contrainte OCL est une partie intégrante d'UML car il participe au mécanisme de navigation dans les diagrammes de classes et d'objets. De plus, il permet d'exprimer la dynamique du modèle dans les diagrammes d'états-transitions. Enfin, le langage UML présente l'avantage de pouvoir se décliner sous forme de « profils » permettant l'extension de ces diagrammes pour spécifier un aspect particulier d'un système.

Le chapitre suivant fera l'objet d'une présentation de certaines méthodes de développement logiciel construites sur UML telles que la méthode : UP, RUP et XP.

7. Exercices

Exercice 1 :

On considère une application constituée des fichiers suivants :

- Un code source registre.cpp.
- Un programme exécutable registre.exe.
- Des librairies dynamiques personne.dll et cours.dll. Les librairies à liens dynamiques sont utilisées lors de l'exécution d'une application.

Travail demandé :
Elaborez le diagramme de composants correspondant.

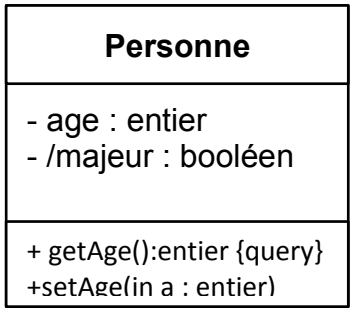
Exercice 2 :
Une architecture matérielle est constituée d'un serveur, d'un kiosque et d'une console. Le serveur utilise une tour de disques durs montés en RAID. la console communique avec le serveur par une liaison Ethernet 100b et la console par une liaison RS232C. Les caractéristiques du serveur sont :

- Processeur Pentium
- Mémoire vive : 4 Go
- Ecran 17 pouces

Travail demandé :
1. Représentez le diagramme de déploiement correspondant.
On peut aussi préciser les modules exécutables sur chaque nœud :
- sur le kiosque : user.exe
- sur le serveur : dbadmin.exe et tkmstr.exe
- sur la console : admin.exe et config.exe

2. Représentez le diagramme correspondant.

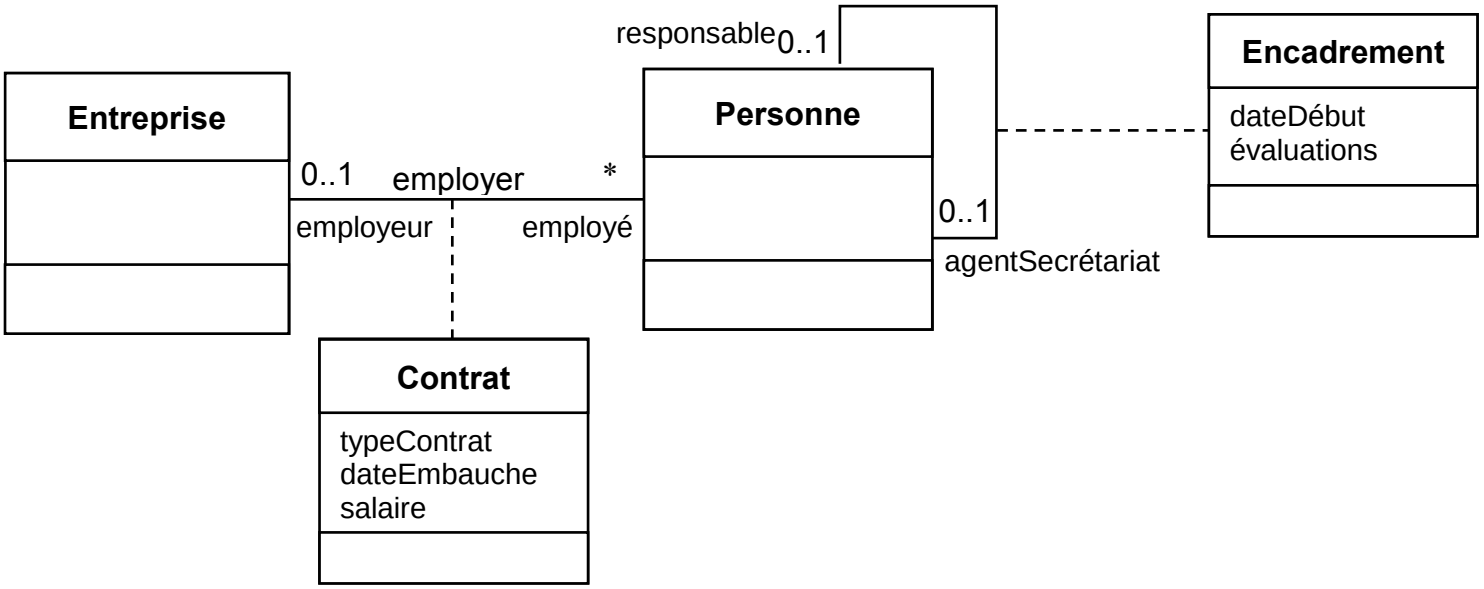
Exercice 3 :



- Ajoutez un attribut mère de type Personne dans la classe Personne.
- Ecrivez une contrainte précisant
  - que la mère d'une personne ne peut être cette personne elle-même.
  - et que l'âge de la mère doit être supérieur à celui de la personne.

Exercice 4 :

Étudier le diagramme de classes UML suivant :

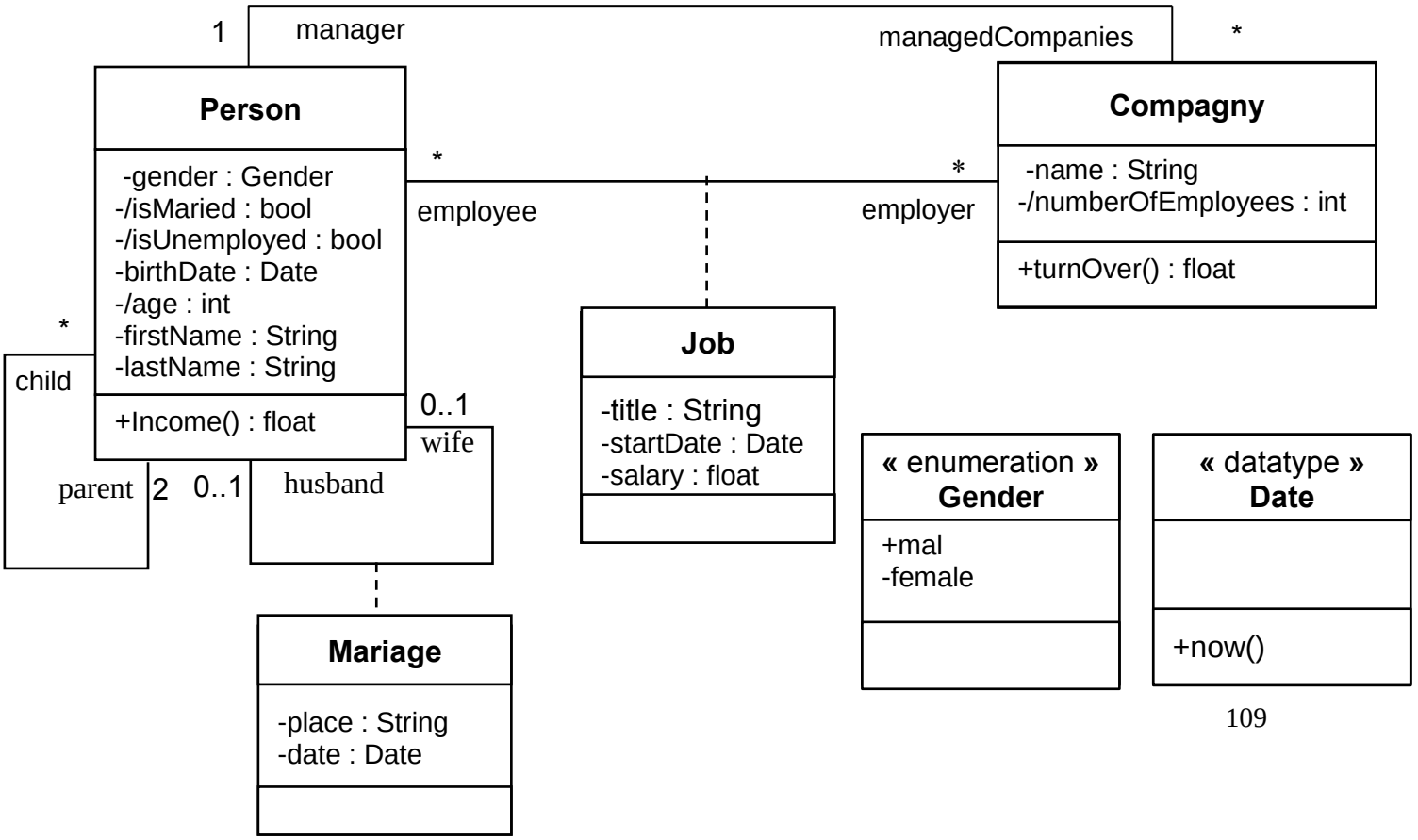


Écrire les contraintes OCL suivantes :

- Le salaire d'un agent de secrétariat est inférieur à celui de son responsable ?
- Un agent de secrétariat a un type de contrat 'agentAdministratif' (String) ?
- Un agent de secrétariat a une date d'embauche antérieure à la date de début de l'encadrement (on suppose que les dates sont des entiers)
- Même chose dans le contexte de la classe Personne

Exercice 5 :

Étudier le diagramme de classes UML suivant :

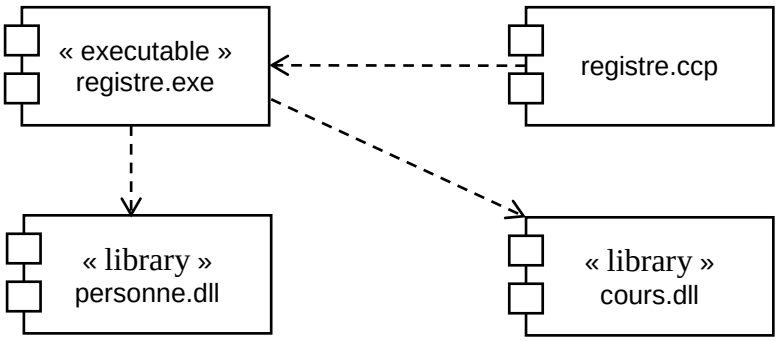


Écrire les contraintes OCL suivante :

1. Initialement le salaire d'un employé est de 1000 euros.
2. Le directeur d'une société doit avoir plus de 40 ans.
3. L'âge d'un individu est fonction du moment présent et de sa date de naissance.
4. Si le chiffre d'affaires (turnover) d'une société est supérieur à 1 million d'euros, elle doit avoir plus que 10 employés.
5. On ne peut pas commencer à travailler avant sa date de naissance.
6. Un homme est marié avec une femme et une femme avec un homme.
7. Pour être mariés, il faut avoir plus que 18 ans.
8. On ne peut pas commencer à travailler le jour de son mariage.

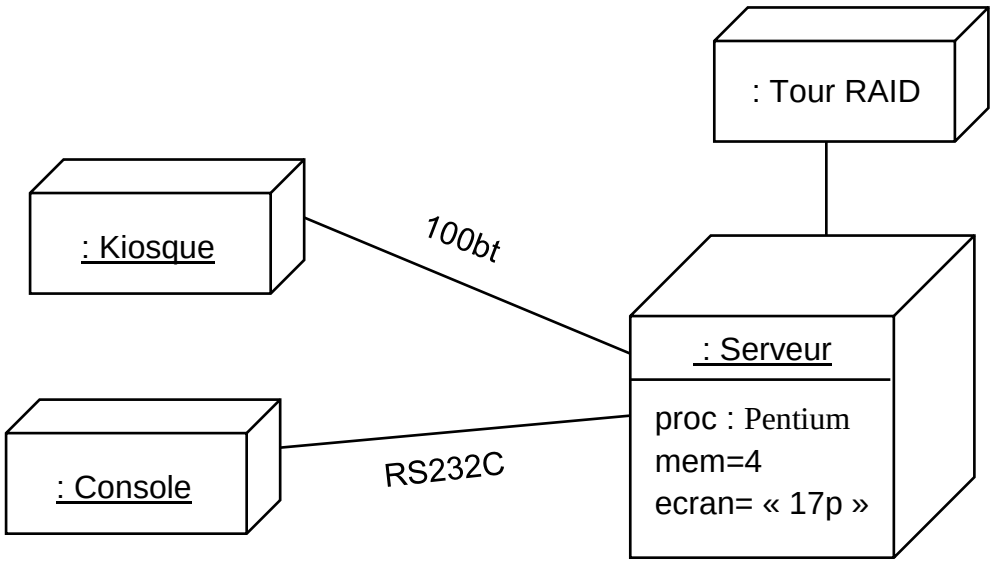
8. Corrigés des exercices

Corrigé de l'exercice 1

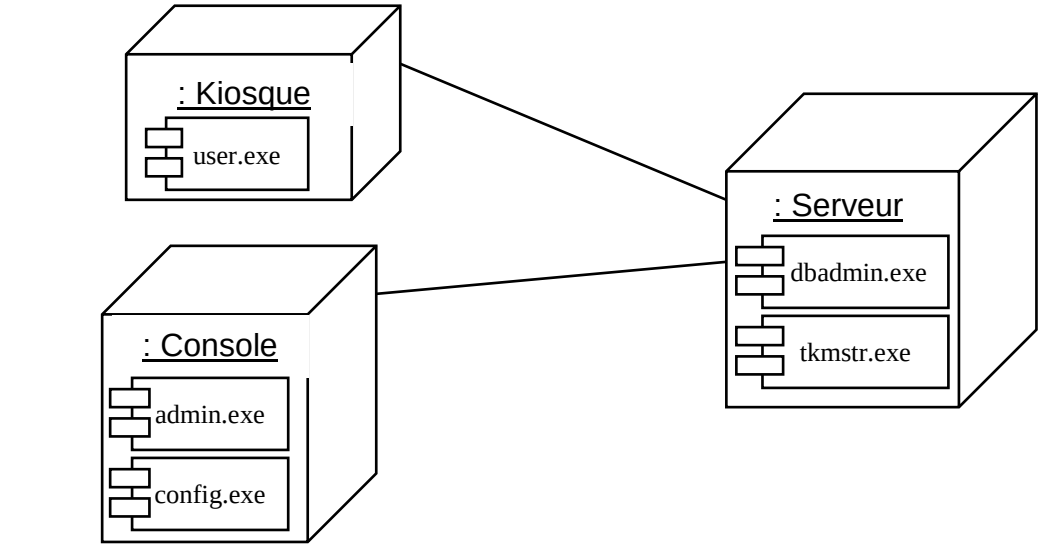


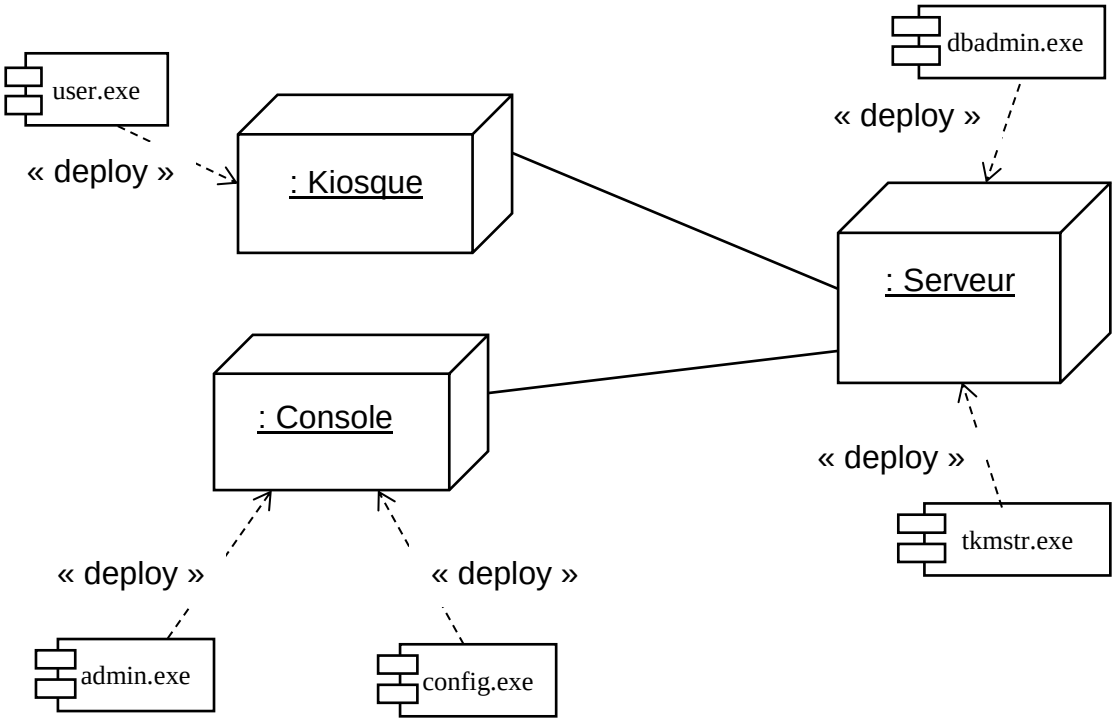
Corrigé de l'exercice 2

1.



2.





Corrigé de l'exercice 3

- Ecrivez une contrainte précisant
  - que la mère d'une personne ne peut être cette personne elle-même.
  - et que l'âge de la mère doit être supérieur à celui de la personne.

context Personne **inv**:

self.mere <> self and self.mere.age > self.age

Corrigé de l'exercice 4

- Le salaire d'un agent de secrétariat est inférieur à celui de son responsable ?
- Un agent de secrétariat a un type de contrat 'agentAdministratif' (String) ?

context p : Personne …
p.encadrement[responsable] -- 1
p.Encadrement[agentSecrétariat] – 2

- Le salaire d'un agent de secrétariat est inférieur à celui de son responsable ?
  context e : encadrement **inv** :
  e.responsable.contrat.salaire >= e.agentSecrétariat.contrat.salaire

- Un agent de secrétariat a un type de contrat ‘agentAdministratif‘ (String) ?
  context e : encadrement **inv** :
  e.agentSecrétariat.contrat.typeContrat='agentAdministratif'

- Un agent de secrétariat a une date d'embauche antérieure à la date de début de l'encadrement (on suppose que les dates sont des entiers)

  context e : Encadrement **inv** :
  e.agentSecrétariat.contrat.dateEmbauche <= e.dateDebut

- Même chose dans le contexte de la classe Personne
  context p : Personne **inv** :

  p.agentSecrétariat.contrat.dateEmbauche

  <= p.encadrement[agentSecrétariat].dateDebut

Corrigé de l’exercice 5

1. Initialement le salaire d’un employé est de 1000 euros.
   context Job
   init : self.salary = 1000

2.Le directeur d’une société doit avoir plus de 40 ans.
   context Company::manager : Person
   inv : self.age > 40

3.L’âge d’un individu est fonction du moment présent et de sa date de naissance.
   context Person
   derive : self.age = Date::now() – self.birthdate

4. Si le chiffre d’affaires (turnover) d’une société est supérieur à 1 million d’euros, elle doit avoir plus que 10 employés.
context Company
inv : (self.turnOver() > 1000000) implies (self.numberOfEmployees > 10)
ou
context Company
inv : (NOT (self.turnOver() > 1000000)) OR (self.numberOfEmployees > 10)

5.On ne peut pas commencer à travailler avant sa date de naissance.

context Job
inv : self.startDate > self.employee.birthDate

6.Un homme est marié avec une femme et une femme avec un homme.
context Marriage
inv : (self.husband.gender=Gender::male) AND (self.wife.gender=Gender::female)

7. Pour être mariés, il faut avoir plus que 18 ans.
context Person
inv : (NOT self.isMarried) OR (self.age >= 18)

8.On ne peut pas commencer à travailler le jour de son mariage.
context Job
inv : NOT (self.startDate = self.employee.marriage.date)

Chapitre 7 : Introduction aux méthodes de développement : (RUP, XP)

Objectif

À l'issue de ce chapitre, l'apprenant sera capable de :

- Maîtriser la complexité des projets informatiques en diminuant les risques.
- Produire un logiciel de qualité en respectant des contraintes de délai, de coûts et de performance ;
- Développer efficacement un logiciel de qualité, en réduisant les risques et en améliorant les prévisions ;
- Connaitre les meilleures méthodes de travail pour apprendre des expériences précédentes.

1. Introduction

UML n'est qu'un langage de modélisation qui spécifie comment décrire des cas d'utilisation, des classes, des interactions, etc. Il ne préjuge pas de la démarche employée.
Un processus (démarche) définit QUI fait QUOI, QUAND et COMMENT pour atteindre un certain objectif.
Il existe plusieurs processus pour développer un logiciel, à savoi: UP (Unified Process), RUP (Rational UnifiedProcess), XP (eXtreme Programming), etc. UP est un processus de développement qui possède les caractéristiques suivantes :

- Itératif et incrémental ;
- Pilotés par les cas d'utilisation ;
- Centré sur l'architecture.

2. Unified Process (UP)

Le processus unifié comme le montre la Figure 7.1 propose quatre phases sur une échelle temporelle : Pré-étude (Inception), Elaboration, Construction et Transition.

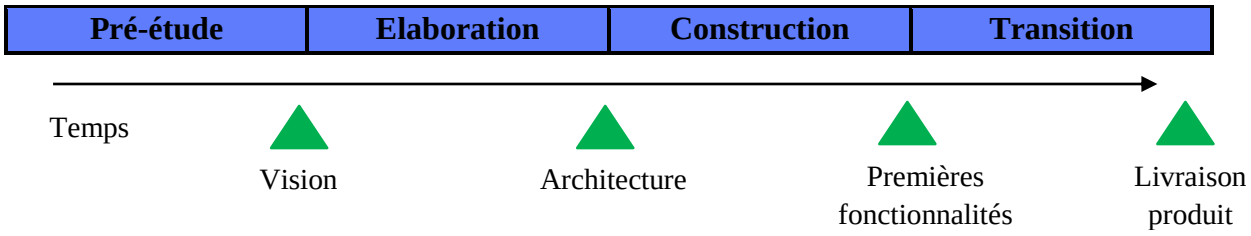


Figure : 7.1 : Processus Unifié.

a. Pré-étude

- Délimiter la portée du système ;
- Définir les frontières et identifier les interfaces ;
- Développer les cas d'utilisation ;
- Décrire et esquisser l'architecture candidate ;
- Identifier les risques les plus sérieux ;
- Démontrer que le système proposé est en mesure de résoudre les problèmes ou de prendre en charge les objectifs fixés.

Le résultat -> Vision : Glossaire, détermination des parties prenantes et des utilisateurs, détermination de leurs Besoins fonctionnels et non fonctionnels, Contraintes de conception.

b. Elaboration

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

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

c. Construction

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

Le résultat -> Produit : Premières fonctionnalités.

b. Transition

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

Le résultat : Livraison du produit aux utilisateurs.

Les activités

- Modélisation métier ;

- Recueil et expression des besoins ;
- Analyse et conception ;
- Implémentation ;
- Test, Déploiement.

3. Rational Unified Process (RUP)

RUP est une version « commerciale » du processus unifié qui est proposée par la société Rational Software, qui appartient à IBM (International Business Machines Corporation).

RUP va plus loin que le processus unifié simple puisqu'elle propose notamment, en plus des spécifications techniques, une méthode de management des équipes et du calendrier ainsi qu'un certain nombre de modèles de documents justificatifs.

4. eXtreme Programming (XP)

La méthode XP, quant à elle, appartient à la famille des méthodes de développement AGILES. XP est basée sur des réalisations très rapides (1 semaine) et implique que le client soit disponible et fortement impliqué et intégré dans l'équipe de développement. Essentiellement concentrée sur l'application elle-même, la méthode XP propose une documentation très réduite, voire inexistante et s'intéresse à la satisfaction du client.

5. XP vs RUP

Inconvénients de XP

Les inconvénients de cette méthode sont les suivants :

- Focalisation sur l'aspect individuel du développement, au détriment d'une vue globale et des pratiques de management ou de formalisation.
- Manquer de contrôle et de structuration, risques de dérive.

Inconvénients de RUP

Les principaux inconvénients de la méthode RUP sont les suivants :

- Fait tout, mais lourd, usine à gaz
- Parfois difficile à mettre en œuvre de façon spécifique.

Remarque

Il est conseillé d'adopter la méthode XP pour les petits projets en équipes de douze (12) personnes maximum. En revanche, pour les grands projets qui génèrent beaucoup de documentation, la méthode RUP peut être satisfaisante.

6. Conclusion

Ce chapitre présente une synthèse des trois principaux processus de développement d'objets associés à UML. Il s'agit de UP (Unified Process), RUP (Rational Unified Process) et Xp (eXtreme Programming). La méthode XP est un mouvement axé sur les programmeurs, alors que RUP a évolué en combinant l'expérience de décennies de développement logiciel avec les nouvelles expériences de développement orienté objet. Le chapitre suivant est réservé aux modèles de conception et à leur place dans le processus de développement.

Chapitre 8 : Patrons de conception et leur place au sein du processus de développement

Objectif

À l'issue de ce chapitre, l'apprenant sera capable de :

- Connaître ce qu'est un patron de conception dans le développement de logiciels ;
- Connaître des procédés de conception généraux qui permettent de capitaliser l'expérience appliquée à la conception de logiciel.

1. Introduction

Depuis plusieurs dizaines d'années, les programmeurs se sont heurtés aux mêmes problèmes lors de la programmation ou la modélisation en utilisant des langages Objet. Les Patrons de Conception (Design Patterns) sont les solutions à ces problèmes, ce sont des façons éprouvées de modéliser des comportements ou des relations.

2. Définition de patron de conception

Les patrons de conception représentent les meilleures pratiques utilisées par les développeurs de logiciels orientés objet expérimentés. Les patrons de conception sont des solutions aux problèmes généraux auxquels les développeurs de logiciels ont été confrontés lors du développement de logiciels. Ces solutions ont été obtenues par essais et erreurs par de nombreux développeurs de logiciels sur une période assez longue.

Les patrons de conception sont des descriptions d'objets et de classes communicantes qui sont adaptées à la résolution d'un problème général de conception dans un contexte particulier. Les patrons peuvent être utilisés dans de nombreux domaines distincts, y compris le développement de logiciels.

Remarque

Tout patron est une règle en trois parties, qui exprime une relation entre un certain contexte, un problème et une solution.

3. But des patrons de conception

Les patrons sont importants car ils offrent des indices visuels sur un ordre sous-jacent. Si vous pouvez déverrouiller un patron, vous avez la possibilité de le modifier ou de le façonner afin d'obtenir un certain effet. Les patrons peuvent également être utilisés comme modèle qui permettra d'analyser rapidement une situation et de comprendre son fonctionnement.

Exemple :

- Apprendre à développer un bon logiciel est similaire à apprendre à bien jouer aux échecs.

D'abord apprendre les règles

- Par exemple le nom des pièces, les mouvements permis, la géométrie de l'échiquier, etc.

Puis apprendre les principes

- Par exemple la valeur relative de certaines pièces, la valeur stratégique des emplacements centraux, etc.

Cependant, pour devenir un champion aux échecs, il faut étudier le jeu d'autres champions

- Ces jeux contiennent des patrons qui doivent être compris, mémorisés, puis appliqués de manière répétée.

Il y a des centaines de patrons

D'abord apprendre les règles

- c.a.d. les algorithmes, les structures de données, UML, les langages de programmation, etc.

Puis apprendre les principes

- Par exemple la programmation structurée, la conception UML, la programmation orientée-objet, la programmation générique, etc.

Mais pour devenir un champion de la conception logicielle, il est important d'étudier la conception d'autres champions

- Ces conceptions contiennent des patrons qui doivent être compris, mémorisés, et appliqués de manière répétée.

Il y a des centaines de patrons

4. Historique des patrons logiciels

L'historique des patrons logiciels peut se résumer comme suit :

1987 : cunningham et Beck utilisent les idées d'Alexander pour développer un petit langage de patrons pour Smalltalk.

1990 : le Gang des 4 (« Gang of Four » : Gamma, Helm, Johnson and Vlissides) commence à travailler à la compilation d'un catalogue de patrons de conceptions.

1991 : Bruce Anderson donne le premier workshop de Patrons au OOPSLA.

1993 : Kent Beck et Grady Booch sponsorisent la première réunion de ce qui est maintenant connu comme le groupe Hillside http://www.hillside.net/

1995 : Le Gang des 4 (GoF) publie le livre des Patrons de conception

5. Types de patrons de conception

Les patrons de conception ont été formellement reconnus en 1994 à la suite de la parution d'un livre co-écrit par Gamma et al. (Gang of Four - GoF ; en français « la bande des quatre »). Dans leur livre, la bande des quatre a défini 23 patrons de conception qui peuvent être classés dans les trois catégories suivantes :

A. ***Patrons de création :*** ces patrons de conception permettent de créer des objets tout en masquant la logique de création, plutôt que d'instancier des objets directement à l'aide d'un nouvel opérateur. Cela donne au programme plus de flexibilité pour décider quels objets doivent être créés pour un cas d'utilisation donné.
Les différents patrons de création sont les suivant :

- **Fabrique**
  Une méthode dans une classe dérivée créé les instances associées
- **Fabrique abstraite**
  Fabrique pour construire des objets liés
- **Monteur**
  Fabrique pour construire des objets complexes de manière incrémentale
- **Prototype**
  Fabrique pour cloner de nouvelles instances d'un prototype
- **Singleton**
  Fabrique pour n'avoir qu'une seule et unique instance

B. ***Patrons de structure :*** ces patrons de conception concernent la composition des classes et des objets. Le concept d'héritage est utilisé pour composer des interfaces et définir des manières de composer des objets pour obtenir de nouvelles fonctionnalités. Les différents patrons de structure sont les suivants :

- **Adaptateur**
  Un traducteur qui adapte une interface de serveur pour un client
- **Pont**
  Découpler l'interface d'une classe et son implémentation
- **Objet composite**
  Structure pour construire des agrégats récursifs
- **Décorateur**
  Etend un objet de manière transparente
- **Façade**
  Façade simplifie l'interface pour un sous-système
- **Poids-mouche**
  De nombreux objets partagés efficacement
- **Proxy**
  Un objet est l'approximation d'un autre

C. ***Patrons de comportement :*** ces patrons de conception concernent spécifiquement la communication entre les objets. Les différents patrons de comportement sont les suivants :

- **Chaine de responsabilité**
  Requête déléguée au fournisseur de service responsable
- **Commande**
  Requête comme objet de première classe

- **Interpréteur**
  Interpréteur de langage pour une petite grammaire
- **Itérateur**
  Eléments d'un agrégat sont atteints séquentiellement
- **Médiateur**
  Médiateur coordonnes les interactions entre ses associés
- **Memento**
  Une photo qui capture et restaure des états d'objets
- **Observateur**
  Les observateurs sont mis au courant des changements des observés
- **Etat**
  Object dont le comportement dépend de son état
- **Stratégie**
  Abstraction pour la sélection d'un parmi plusieurs algorithmes
- **Patron de méthode**
  Algorithme avec des pas fournit par une classe dérivée
- **Visiteur**
  Opérations appliquée aux éléments d'une structure d'objet hétérogène

Une autre catégorie de modèle de conception peut être également abordée : les modèles de conception J2EE.

D. Patrons J2EE : ces patrons de conception concernent spécifiquement le niveau de présentation. Ces modèles sont identifiés par Sun Java Center.

6. Avantages et inconvénients de l'utilisation des patrons de conception

Avantage :

Les patrons de conception offrent donc les avantages suivants :

- Ils offrent au développeur une sélection de solutions éprouvées avec lesquelles travailler ;
- Ils sont indépendants du langage et peuvent donc être appliqués à n'importe quel langage prenant en charge l'orientation objet ;
- Ils facilitent la communication par le fait même qu'ils sont bien documentés et peuvent faire l'objet de recherches si ce n'est pas le cas ;
- Ils ont fait leurs preuves car ils sont déjà largement utilisés et réduisent ainsi le risque technique pour le projet ;
- Ils sont très flexibles et peuvent être utilisés dans pratiquement n'importe quel type d'application ou de domaine.

Inconvénients :

Les patrons de conception souffrent donc les inconvénients suivants :

- Les patrons sont simplistes, mais ils doivent être appliqués à notre problème ;

- Utiliser des patrons requiert la capacité de faire le lien entre un patron et notre problème ;
- Les patrons sont validés par l'expérience, plutôt que par des tests automatiques ;

7. Conclusion

Les patrons de conception représentent un investissement très rentable malgré leur courbe d'apprentissage initiale. Ils nous permettront de mettre en œuvre des solutions éprouvées aux problèmes, économisant ainsi du temps et des efforts lors de la phase de mise en œuvre du cycle de vie du développement logiciel. En utilisant des solutions bien comprises et documentées, le produit final aura un degré de compréhension beaucoup plus élevé. Si la solution est plus facile à comprendre, alors par extension, elle sera également plus facile à maintenir. En fin, les patrons de conception représentent une excellente garantie pour un produit logiciel de qualité, un code réutilisable et facile à entretenir.

8. Exercices

Exercice 1 :

Un éditeur de jeux développe un jeu permettant aux enfants de connaître les animaux. Les enfants peuvent, en particulier, apprendre la forme et le cri des animaux parmi lesquels le chat et la vache. Le chat est modélisé par la classe LeChat possédant au moins les deux méthodes formeChat() et criChat() et la vache est modélisée par la classe LaVache possédant les deux méthodes criVache() et formeVache().
Comme le montrent les noms des méthodes, la première spécification de ce jeu est propre aux animaux modélisés. L'éditeur souhaite améliorer ce jeu en créant une interface commune à tous les animaux qui lui permette d'en ajouter de nouveaux, sans modifer l'interface avec le code client, et d'utiliser le polymorphisme dans la gestion des animaux (manipuler des troupeaux hétéroclites...).

Travail demandé :

1. Proposez une modélisation des classes pour cette nouvelle version du jeu en faisant apparaître le client. Les classes seront Chat, Vache... et non plus LeChat, LaVache...

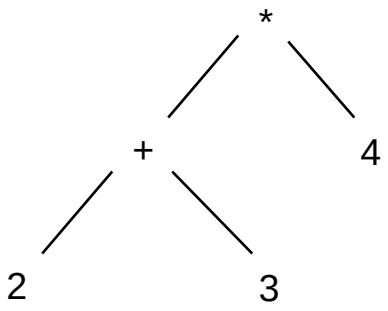
2. On souhaite réutiliser tout le code développé dans la version précédente (avec LeChat, LaVache...) dans le nouveau logiciel utilisant les nouvelles classes (Chat, Vache...).

Proposez une modélisation permettant d'incorporer les anciennes méthodes pour éviter de les récrire. Vous pourrez utiliser le patron de conception "adapter ".

Exercice 2 :

Une expression arithmétique peut se représenter de manière arborescente. Par exemple, l'expression (2+3)*4 peut se représenter comme le résultat de l'opération * appliquée à 4 et au résultat d'une seconde opération + appliquée à 2 et à 3. 4 et +(2,3) sont dits opérandes de l'expression qui a * comme opérateur. Selon ce principe, une autre notation pour cette expression arithmétique est *(+(2,3),4), aussi appelée notation " polonaise " et notamment utilisée sur les calculatrice HP. Cette notation peut se représenter de avec un arbre comme suit.

Il est à noter qu'il devient ainsi possible de se passer tout à fait de parenthèses : il n'y a aucune ambiguïté lorsqu'on utilise la notation *+234.



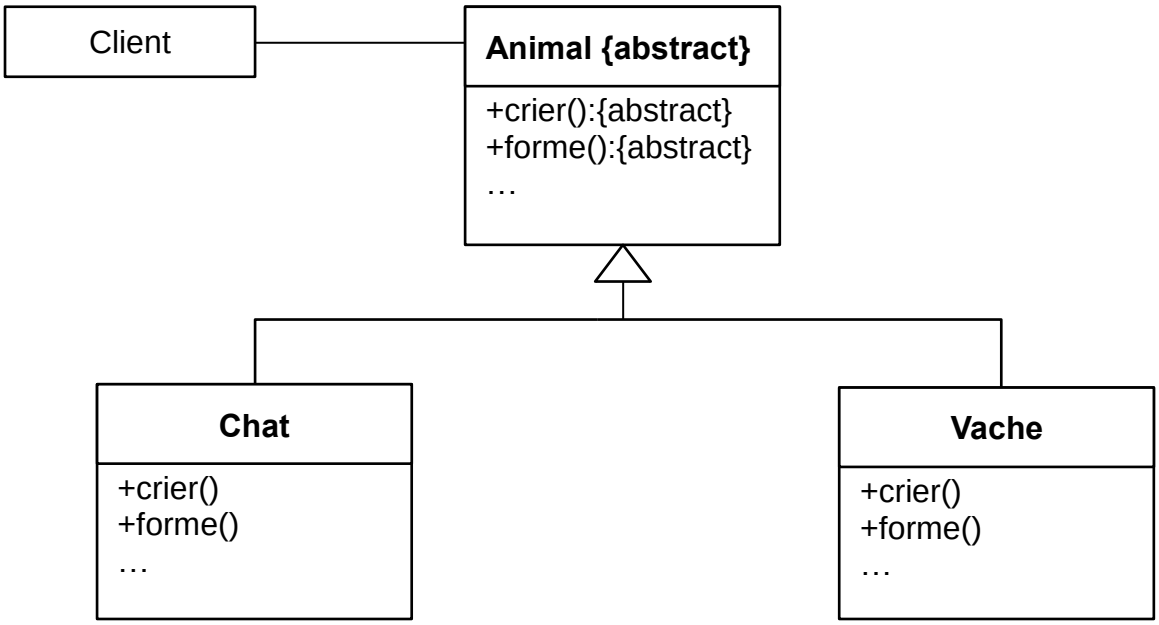
Il y a deux types d'expressions : les expressions binaires comme +(2,3) et les expressions unaires qui utilisent des opérations ne prenant qu'un seul argument, comme par exemple l'opérateur de changement de signe - dans l'expression -1. Ces deux types d'expression sont caractérisées par un opérateur et disposent d'une opération calculerValeur(). Le " terme " est un concept plus général qu'une expression : il peut être soit une valeur constante (1,2, 3 ou 4) soit une expression comme +(2,3). Dans tous les cas, un terme doit disposer d'une opération calculerValeur(). Un autre type de terme peut être une variable qui, en plus d'une valeur comme pour les constantes, dispose d'un nom.

Travail demandé :
Utilisez le pattern " composite " pour produire un diagramme de classes adéquat pour la représentation des expressions arithmétiques.

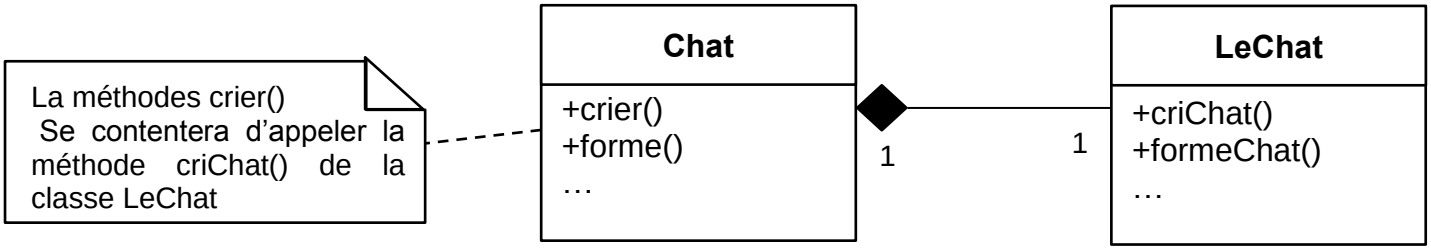
9. Corrigés des exercices

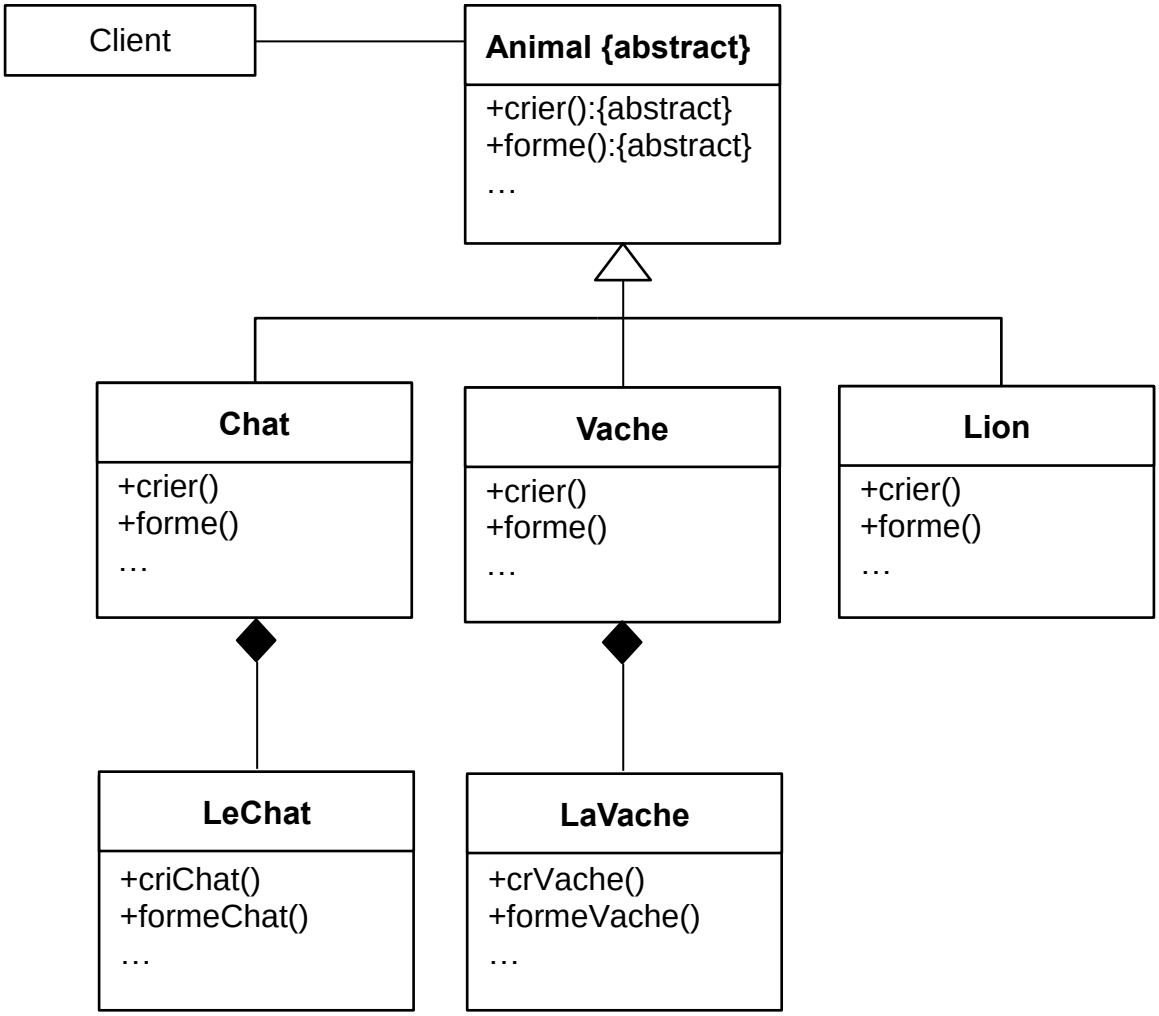
Corrigé de l'exercice 1

1.

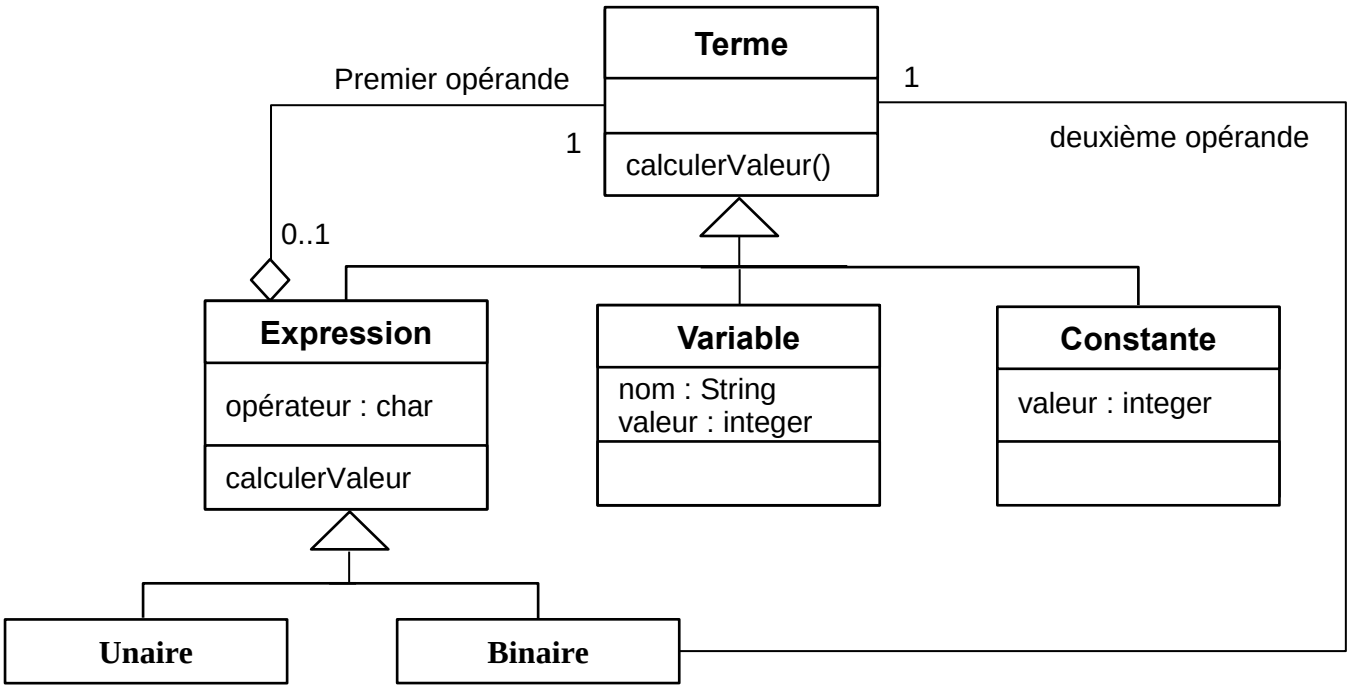


2.


La méthodes crier()
Se contentera d'appeler la méthode criChat() de la classe LeChat
Chat
+crier()
+forme()
…
1
1
LeChat
+criChat()
+formeChat()
…


Client
Animal {abstract}
+crier():{abstract}
+forme():{abstract}
…
Chat
+crier()
+forme()
…
Vache
+crier()
+forme()
…
Lion
+crier()
+forme()
…
LeChat
+criChat()
+formeChat()
…
LaVache
+crVache()
+formeVache()
…

Corrigé de l'exercice 2



Conclusion générale

Ce document tente de donner aux apprenants une idée générale du génie logiciel en indiquant non seulement les raisons de son émergence, mais aussi son adoption dans le monde informatique. L'émergence du génie logiciel comme nouvelle discipline à part entière est due à l'éclatement de la «crise du logiciel» à la fin des années 1960, dénotant la situation générale qui caractérise les difficultés rencontrées dans le développement de logiciels. Le génie logiciel est la science qui traite des objectifs, des moyens, et des problèmes rencontrés en développant des logiciels. Cette science étudie notamment la répartition des phases dans le temps, les bonnes pratiques concernant les documents clés que sont le cahier des charges, et le diagramme de classes. L'objectif est d'obtenir un logiciel à grande échelle, fiable, de qualité et répondant aux attentes des utilisateurs. Néanmoins, le développement de logiciels comprend l'étude, la conception, la construction, la transformation, la mise au point, la maintenance et l'amélioration du logiciel. Tout ce travail est réalisé par des salariés des éditeurs de logiciels, des sociétés de services et d'ingénierie informatiques, des membres de la communauté du logiciel libre et des travailleurs indépendants. À l'heure actuelle, le *génie logiciel* est le domaine de l'informatique qui concerne la production de logiciel qui sont si grands et/ou si complexes que seules des équipes importantes et organisées sont capables de les construire. De plus, les systèmes logiciels produits est de qualité à bon prix et dans des délais raisonnables.

L'objectif de ce polycopié de cours était de faire en sorte que les apprenants maîtrisent les constituantes théoriques et pratiques du *génie logiciel*. À l'issue de ce support de cours, l'apprenant sera capable d'acquérir des connaissances enrichissantes des techniques et méthodes de conception, de réalisation, de mise en œuvre et de maintenance de logiciels. Ce cours de génie logiciel a été complété par des méthodologies sur la gestion de projet et la qualité dans le but de donner aux étudiants une dimension leur permettant de maîtriser des projets complexes dans un environnement industriel.

J'espère que vous trouverez ce document utile et que vous en prendrez autant de plaisir dans sa lecture. En souhaitant pareillement que ce cours vous aidera aisément à enrichir vos connaissances relatives à la conception et à la production de logiciels complexes et de qualité. Moi, docteur **Mohamed MOHAMMEDI** auteur de ce document, je vous remercie pour le temps que vous avez consacré à la lecture de ce support de cours.

Bibliographie

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

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

[3] I. Boussaïd, " Cycle de vie d'un logiciel ", Université des Sciences et de la Technologie Houari Boumediene, Licence 3 Académique, site http://sites.google.com/site/ilhemboussaid, 2010.

[4] E. Cariou, "Object Constraint Language (OCL)", Université de Pau et des Pays de l'Adour, UFR Sciences Pau – Département Informatique, 2019.

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

[6] L. HAMZA, Génie logiciel, support de cours, Université de bejaia, https://elearning.univ-bejaia.dz/course/view.php?id=5958, 2019.

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

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

[9] E. Gamma, R. Helm, R. Johnson, et J. Vlissides, "Design Patterns Elements of Reusable Object-Oriented Software," Addison-Wesley, Reading, 1995.

[10] P. Gérard, Introduction à UML 2 : Modélisation Orientée Objet de Systèmes Logiciels, Université de Paris 13-IUT Villetaneuse, DUT Informatique-S2D, 2009.

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

[12] L. Matignon, "OCL - Object Constraint Language", Département Informatique - Polytech Lyon, université Claude Bernard Lyon 1, 2012.

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

[14] R. Yende. Support de cours de génie logiciel. Licence. RDC (BÉNI), CongoKinshasa. ffcel-01988734, 2019.

Génie Logiciel

Mohamed Mohammedi

Ce support de cours est destiné à tous les étudiants de 3ème année Licence en informatique, professionnels, concepteurs et développeurs, qui souhaitent mieux maîtriser les principes d'ingénierie et les appliquer au domaine de la création de logiciels. Il s'agit d'identifier et d'utiliser des méthodes, des pratiques et des outils permettant de maximiser les chances de réussite d'un projet logiciel.

Il propose une approche pédagogique de l'aspect normatif du génie logiciel afin de concevoir et produire des logiciels de qualité avec maîtrise des coûts et délais. Le lecteur suit un apprentissage progressif appuyé sur de nombreux exemples, et exercices corrigés permettant d'appliquer les concepts présentés.

Ce document avait le plus grand mérite de :

- Présenter la naissance et l'importance de Génie Logiciel.
- Étaler les grands enjeux et les bonnes pratiques liés à l'activité de réalisation de logiciels ;
- Donner une description détaillée de la plus part des diagrammes fondamentaux d'UML ;
- Introduire quelques méthodes de développement logiciel construites sur UML pour apprendre des expériences précédentes ;
- Exposer les patrons logiciels, qui sont des modèles de solutions à des problématiques fréquentes d'architecture ou de conception.