الجمهورية الجزائرية الديمقراطية الشعبية

وزارة التعليم العالي و البحث العلمي

جامعة قالمة

كلية الرياضيات و الإعلام الآلي وعلوم المادة

قسم: الإعلام الآلي

Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

Université 08 mai 1945 Guelma

Faculté des Mathématiques et de l'Informatique et des Sciences de la Matière

Département d'Informatique

Polycopié de Cours - Génie Logiciel

3ème Année Systèmes Informatiques (SI)

Licence Académique

Par : DJAKHDJAKHA Lynda.

2022/2023

Syllabus

Licence : Systèmes Informatique (SI)

Unité d'Enseignement Fondamentale : UEF2

Matière : Génie Logiciel

Semestre : 5 **Année universitaire :** 2022/2023

Crédits : 05 **Coefficient :** 03

Volume Horaire Total : 63H

Cours (V. H. Hebdomadaire) : 1H 30 min

Travaux Dirigés (V. H. Hebdomadaire) : 1H 30 min

Travaux Pratiques (V. H. Hebdomadaire) : 1H 30 min

Enseignant responsable de la matière : DJAKHDJAKHA Lynda, Grade : M.C.B.

Email : djakhdjakha.lynda@univ-guelma.dz

ldjakhdjakha@yahoo.fr

Objectifs

Apprendre à appliquer une méthodologie d'analyse et de conception pour le développement des logiciels. En particulier, apprendre la modélisation objet avec le langage universel UML.

Connaissances préalables recommandées

Algorithmique, Système d'information, Programmation Orienté Objet.

Contenu de la matière

Chapitre 1. Introduction

- Définitions et objectifs
- Principes du Génie Logiciel
- Qualités attendues d'un logiciel
- Cycle de vie d'un logiciel

- Modèles de cycle de vie d'un logiciel

Chapitre 2. Modélisation avec UML

- Introduction
- Modélisation, Modèle, Modélisation Orientée Objet, UML en application.
- Éléments et mécanismes généraux
- Les diagrammes UML
- Paquetages

Chapitre 3. Diagramme UML de cas d'utilisation : Vue fonctionnelle

- Intérêt et définition, Notation

Chapitre 4. Diagrammes UML de classe et d'objet : Vue statique

- Diagramme de classes
- Diagramme d'objets

Chapitre 5. Diagrammes UML : vue dynamique

- Diagramme d'interaction (Séquence et collaboration)
- Diagramme d'activités
- Diagramme d'états/transitions

Chapitre 6. Autres notions et diagrammes UML

- Composants, déploiement, structures composites.
- Mécanismes d'extension : langage OCL + les profils.

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

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

Mode d'évaluation : Examen (60%), contrôle continu (40%)

Références :

1. Bern Bruegge and Allen H. Dutoit, Object-Oriented Software Engineering – using UML, Patterns and Java. Third Edition, Pearson, 2010.
2. G. Booch, J. Rumbaugh, I. Jacobson, « The Unified Modeling Language (UML) Reference Guide », Addison-Wesley, 1999.
3. G. Booch, J. Rumbaugh, I. Jacobson, « The Unified Modeling Language (UML) User Guide », Addison-Wesley, 1999.
4. G. Booch et al., « Object-Oriented Analysis and Design, with applications », Addison- Wesley, 2007.

5. Laurent Audibert. Cours UML 2.0, dispobible sur http://www.developpez.com.
6. M. Blaha et J. Rumbaugh. Modélisation et conception orientées objet avec UML 2. 2ème édition. Pearson Education, 2005.
7. Pierre-Alain Muller. Modélisation objet avec UML. Éditions Eyrolles, 2003.
8. Shari Lawrence Pfleeger and Joanne M. Atlee.Software Engineering. Fourth Edition, Pearson, 2010.

Signature du responsable de la matière

Introduction générale

Le génie logiciel est une discipline qui vise à appliquer des principes d'ingénierie pour concevoir, développer, tester et maintenir des logiciels de qualité. Il est devenu essentiel dans le monde de l'informatique, car les logiciels sont devenus omniprésents dans notre vie quotidienne, tant dans les appareils électroniques que dans les systèmes d'information professionnels.

Ce support de cours de génie logiciel est destiné aux étudiants en licence informatique. Il couvre plusieurs chapitres clés pour comprendre les principes fondamentaux du génie logiciel.

Le premier chapitre est une introduction générale au génie logiciel, qui explique les principes de base, les défis et les avantages de cette discipline.

Le deuxième chapitre porte sur la modélisation en UML. Il explique les concepts de base d'UML et comment les utiliser pour concevoir des modèles de logiciels.

Le troisième chapitre traite de la vue fonctionnelle du logiciel. Cette vue décrit les fonctionnalités du logiciel et les tâches qu'il doit effectuer. Ce chapitre explique comment spécifier les exigences fonctionnelles du logiciel et comment concevoir des diagrammes des cas d´utilisation.

Le quatrième chapitre aborde la vue statique du logiciel. Cette vue décrit la structure du système, c'est-à-dire les classes, les interfaces et les relations entre elles. Ce chapitre explique comment concevoir des diagrammes de classes et des diagrammes d´objets pour représenter la structure du logiciel.

Le cinquième chapitre se concentre sur la vue dynamique du logiciel. Cette vue décrit le comportement du logiciel lorsqu'il est en cours d'exécution. Ce chapitre explique comment concevoir des diagrammes d´activités, des diagrammes dìntéraction et des diagrammes d'états pour modéliser le comportement du logiciel.

Le sixième chapitre traite autres notions et diagrammes UML. Il décrit les diagrammes de composants, de déploiement et de structures composites. Ce chapitre décrit ainsi quelques mécanismes d'extension tels que langage OCL.

Le septième chapitre couvre le processus unifié de développement de logiciels. Ce processus est une méthode de développement de logiciels bien établie qui suit une approche itérative et incrémentale. Ce chapitre explique les différentes phases du processus unifié et comment il peut être utilisé pour développer des logiciels de qualité.

Le huitième chapitre traite des patrons de conception. Les patrons de conception sont des solutions éprouvées à des problèmes de conception de logiciels courants. Ce chapitre explique les différents types de patrons de conception et comment les utiliser pour résoudre des problèmes de conception de logiciels.

Table des matières

Liste des figures

1.Chapitre 01/ Introduction

Introduction

Le concept du Génie Logiciel (GL) est né en 1968 à Garmisch (Allemagne) (« 1st conference on software engineering », lorsque des ingénieurs informatiques et des experts de l'informatique ont commencé à s'intéresser aux méthodes permettant de créer des logiciels plus efficaces et de meilleure qualité.

Le génie logiciel est devenu de plus en plus important à mesure que les logiciels sont devenus de plus en plus complexes et omniprésents dans la vie quotidienne. Il est essentiel pour la création de logiciels de haute qualité qui sont utilisés dans de nombreux domaines tels que les entreprises, la santé, l'éducation, les transports, la communication, etc.

1.1. Définitions et objectifs

Selon ''IEEE´´[1]: «Le génie logiciel est l'application d'une approche systématique, disciplinée et quantifiable au développement, à l´exploitation e t à l a maintenance des logiciels; c'est-à-dire l'application de l'ingénierie aux logiciels».

Selon '' Fritz Bauer''[2]: «Le génie logiciel est l´établissement et l´utilisation de principes d´ingénierie afin d´obtenir des logiciels économiques, fiables et efficaces sur des machines réelles».

Donc, le génie logiciel est une discipline de l'informatique qui concerne la conception, le développement, la maintenance et la gestion de logiciels. Il s'agit d'une approche systématique et structurée pour produire des logiciels de haute qualité en utilisant des méthodes, des techniques et des outils de développement logiciel.

Il comprend plusieurs activités telles que l'analyse des besoins des utilisateurs, la conception de l'architecture logicielle, la programmation, les tests, la documentation et la maintenance.

Le génie logiciel vise à produire des logiciels qui sont fiables, efficaces, maintenables, évolutifs, conviviaux et qui répondent aux besoins des utilisateurs.

[1] IEEE : http://www.ieee.org
[2] P. Naur, B. randall (eds) "Software Engineering: A Report on a Conference Sponsored by the NATO Science Committee", NATO, Bruxelles, Belgique, 1969.

1.2. **Les principes du génie logiciel**

Le génie logiciel vise à produire des logiciels de haute qualité de manière efficiente en utilisant des principes tels que[3] :

- **Le principe de rigueur** : consiste à adopter des pratiques rigoureuses et à suivre des normes strictes tout au long du processus de développement pour minimiser les erreurs et les dysfonctionnements. Pour ce faire, il est important de remettre en question la validité de chaque action, de documenter les étapes de développement, d'utiliser des outils de vérification et de tests automatisés, et de respecter les normes de codage. La rigueur garantit la qualité, la fiabilité, la maintenance et l'évolutivité à long terme du logiciel.

- **Le principe de généralisation** : il est important de concevoir des solutions qui puissent être appliquées à une grande variété de situations. Les solutions génériques peuvent être réutilisées et adaptées plus facilement que des solutions spécifiques.

- **Le principe de structuration** : il s'agit de diviser un système complexe en sous-systèmes plus petits et plus faciles à gérer. Ce principe permet de mieux comprendre le système dans son ensemble, de le développer plus rapidement et de le maintenir plus facilement.

- **Le principe de modularité** : il s'agit de concevoir un système comme une collection de modules indépendants qui peuvent être développés, testés et déployés de manière autonome. Ce principe permet de réduire la complexité du système et d'améliorer sa maintenabilité.

- **Le principe d'abstraction** : il s'agit de représenter les concepts complexes à l'aide de concepts plus simples et plus abstraits. Ce principe permet de simplifier la conception du système, d'améliorer la compréhension du système et de faciliter sa maintenance.

- **Le principe de construction incrémentale** : il s'agit de développer un système par étapes successives, en ajoutant des fonctionnalités au fil du temps. Ce principe permet de réduire les risques associés au développement de logiciels et de fournir des versions intermédiaires du système qui peuvent être testées et validées par les utilisateurs.

- **Le principe de documentation** : il est important de documenter le système pour faciliter sa compréhension et sa maintenance. La documentation peut prendre plusieurs formes, telles que des spécifications de conception, des manuels d'utilisation ou des commentaires de code.

[3] Raphael Yende, support de cours de génie logiciel. Licence. RDC (Béni), Congo Kinshasa. 2019. cel-01988734.

- **Le principe de vérification** : Le principe de vérification consiste à s'assurer que le logiciel développé répond aux exigences fonctionnelles et non-fonctionnelles définies au préalable. Cela implique de vérifier chaque étape du processus de développement, depuis la conception jusqu'à la mise en production, en utilisant des techniques et des outils adaptés.

1.3. Qualité d'un logiciel[4]

La qualité d'un logiciel peut se définir par sa facilité d'utilisation, sa robustesse, sa fiabilité, sa sécurité, sa performance et sa conformité aux normes et aux exigences du marché.

Dans l'ISO 8402 (1994): « La qualité est un ensemble des traits et des caractéristiques d'un produit ou d'un service qui leur confèrent l'aptitude à satisfaire des besoins exprimés ou implicites ».

Dans l'ISO 9000 (version 2000): « La qualité est l'aptitude d'un ensemble de caractéristiques intrinsèques d'un produit, d'un système ou d'un processus à satisfaire les exigences des clients et autres parties intéressées».

1.3.1. Le modèle de qualité McCall

Né en 1977. Il a été défini lors d'une étude pour l'U.S.Air Force. Il définit 11 facteurs de Qualité regroupés en: Facteurs de l'exploitation, Facteurs de l'évolution de l'application et Facteurs d'adaptabilité de l'application, et 23 Critères de Qualité. Il permet d'évaluer la valeur d'une application informatique.

Un facteur de qualité : une caractéristique de l'application qui contribue à sa qualité, est orienté utilisateur et a un coût. Ils sont regroupés en: Facteurs de l'exploitation, Facteurs de l'évolution de l'application et Facteurs d'adaptabilité de l'application.

Un critère: un attribut de l'application , à travers lequel un facteur de qualité peut être évalué.

1.3.2. La norme de qualité ISO 9126

La norme ISO 9123 est une norme internationale établie en 2001. Son contenu est repris des enrichissements dans la norme ISO25000, appelée aussi SQuaRE (Software Quality Requirement and Evaluation). Il définit les critères de qualité à appliquer lors de la conception et de l'évaluation de logiciels. Elle définit six caractéristiques qualité normalisées pour mesurer la qualité d'un logiciel. Ces caractéristiques sont :

- La capacité fonctionnelle mesure la fonctionnalité et la qualité des exigences du logiciel.

[4] Olivier Englender, Sophie Fernandes. Manager un projet informatique, Troisième édition, EYROLLES, 2012

- La fiabilité mesure la capacité du logiciel à fonctionner sans interruption et à produire des résultats précis.

- La facilité d'utilisation mesure la quantité d'effort et la quantité de temps nécessaires pour apprendre et maîtriser le logiciel.

- Le rendement mesure la quantité de ressources (temps et argent) nécessaires pour obtenir les résultats souhaités à partir du logiciel.

- La maintenabilité mesure la capacité du logiciel à être modifié pour répondre à de nouvelles exigences.

- Enfin, **la portabilité** mesure la capacité du logiciel à être transféré et utilisé sur différents systèmes informatiques.

1.4. Le cycle de vie d´un logiciel[5]

Le cycle de vie du logiciel est une approche globale de gestion d'un projet informatique qui comprend plusieurs étapes essentielles pour la création et la maintenance d'un logiciel. Il contient au minimum les étapes suivantes:

- **Définition des objectifs** : Cette étape consiste à identifier les besoins des utilisateurs et à définir les objectifs du projet.

- **Analyse des besoins et faisabilité** : Cette étape consiste à analyser les besoins des utilisateurs et à déterminer la faisabilité technique, économique et juridique du projet.

- **Conception générale** : Cette étape consiste à concevoir une architecture logicielle globale qui répond aux besoins des utilisateurs.

- **Conception détaillée** : Cette étape consiste à concevoir les détails de la solution, tels que les interfaces utilisateur, les algorithmes, les structures de données, etc.

- **Développement** : Cette étape consiste à coder le logiciel en utilisant les langages de programmation appropriés et les outils de développement.

- **Tests** : Cette étape consiste à tester le logiciel pour s'assurer qu'il fonctionne correctement et répond aux spécifications.

- **Recette** : Cette étape consiste à vérifier que le logiciel est conforme aux exigences définies et que les tests sont satisfaisants.

- **Documentation** : Cette étape consiste à produire une documentation détaillée pour les utilisateurs et les développeurs.

[5] Pascal ROQUES, Franck VALLÉE. UML en action De l'analyse des besoins à la conception en Java, Deuxième édition, EYROLLES, 2003
Valéry Guilhem Frémaux. Le projet informatique de A à Z, Approche pragmatique de la gestion de projet, ellipses, 2006

- **Mise en production** : Cette étape consiste à déployer le logiciel sur le système informatique de l'entreprise.

- **Assistance et maintenance** : Cette étape consiste à assurer le suivi et la maintenance du logiciel pour garantir son bon fonctionnement dans le temps et pour répondre aux éventuels besoins d'évolution.

Le cycle de vie du logiciel est un processus continu qui nécessite une attention constante aux détails et aux besoins des utilisateurs pour assurer la réussite du projet.

1.5. Les modèles de cycle de vie[6]

Il existe plusieurs modèles de cycle de vie du logiciel, chacun avec ses avantages et inconvénients:

1.5.1. Les modèles séquentiels

Les modèles séquentiels de cycle de vie d'un logiciel décrivent les différentes étapes de développement d'un logiciel de manière séquentielle:

-***Modèle en cascade*** : Ce modèle est l'un des plus anciens et des plus simples. Il suit une approche linéaire dans laquelle les différentes étapes (analyse, conception, développement, test, déploiement) sont effectuées dans un ordre fixe. Chaque étape doit être achevée avant de passer à la suivante.

-***Modèle en V*** : Ce modèle est similaire au modèle en cascade, mais il met davantage l'accent sur les tests. Chaque phase de développement est associée à une phase de test correspondante.

-***Le modèle d'intégration***: Il met l'accent sur l'intégration continue des différents modules ou composants du logiciel. Ce modèle est particulièrement utile pour les projets de grande envergure qui impliquent plusieurs équipes de développement travaillant sur des parties distinctes du logiciel. Il consiste en une série de cycles d'intégration, où chaque cycle implique la prise en compte de plusieurs modules, jusqu'à ce que l'ensemble du logiciel soit intégré et testé.

-***Le modèle RAD (Rapid Application Development)***: Le modèle RAD met l'accent sur le développement rapide et itératif de logiciels. Il est particulièrement utile pour les projets qui ont des délais stricts et des exigences changeantes. Le modèle RAD implique une approche itérative, où les développeurs créent des prototypes rapidement, testent et améliorent continuellement le logiciel.

1.5.2. Les modèles itératifs

Les modèles itératifs sont des modèles de développement de logiciel qui impliquent des cycles de développement répétitifs et incrémentaux pour produire des versions

[6] Olivier Englender, Sophie Fernandes. Manager un projet informatique, Troisième édition, EYROLLES, 2012

fonctionnelles d'un produit logiciel. Ils sont souvent utilisés dans les projets de développement de logiciels où les exigences peuvent changer ou évoluer au fil du temps.

- ***Le modèle incrémental*** : Le modèle incrémental est un modèle de développement de logiciel dans lequel les exigences sont divisées en plusieurs étapes, et chaque étape est développée et testée séparément. Il suit un processus linéaire séquentiel, mais chaque étape est une amélioration incrémentale par rapport à la précédente.

- ***Le modèle en spirale de Bohm*** : Le modèle en spirale de Bohm est un modèle itératif de développement de logiciel qui implique la planification, la conception, la mise en œuvre et l'évaluation à chaque étape. Il suit un processus cyclique où chaque cycle se compose de quatre phases : la planification, la définition des objectifs, l'évaluation des risques, et la mise en œuvre. Chaque cycle de développement est basé sur les résultats du cycle précédent.

- ***Le modèle UP (Processus Unifié)*** : Le modèle UP est un modèle itératif de développement de logiciel qui implique des cycles de développement répétitifs et incrémentaux pour produire des versions fonctionnelles d'un produit logiciel. Le modèle UP suit un processus itératif et incrémental où chaque cycle est divisé en quatre phases : l'élaboration, la construction, la transition et la production. Le modèle UP est conçu pour être flexible et peut être adapté à différents types de projets de développement de logiciels (*cf. Chapitre 7*).

1.5.3. Les méthodes agiles

Les méthodes agiles mettent l'accent sur la collaboration, l'adaptabilité et la livraison de fonctionnalités en continu plutôt que sur la planification rigide et la documentation exhaustive. Il existe plusieurs modèles, notamment:

- ***SCRUM***: il s'agit d'un cadre de développement de logiciels itératif et incrémental qui met l'accent sur la livraison de fonctionnalités en continu par l'intermédiaire de sprints de développement courts. Le travail est divisé en petites tâches gérées par un product owner et exécutées par une équipe de développement.

- ***XP (Extreme Programming)***: il met l'accent sur la qualité du code, la communication et la collaboration entre l'équipe de développement et le client. Elle se concentre également sur les pratiques de développement telles que les tests automatisés, la programmation en binôme et la refactorisation.

- ***ASD (Adaptive Software Development)***: il met l'accent sur l'adaptation aux changements imprévus dans les exigences du projet. ASD se concentre également sur la communication et la collaboration entre les membres de l'équipe de développement et le client.

- ***SDM (Dynamic Systems Development Method)***: il met l'accent sur la livraison rapide et efficace de solutions logicielles de haute qualité. Il se concentre également sur la collaboration entre les membres de l'équipe de développement et le client.

- ***FDD (Feature Driven Development)***: il met l'accent sur la livraison de fonctionnalités en continu. Il se concentre également sur la planification et la gestion des fonctionnalités en utilisant des pratiques telles que les modèles de conception orientés objet et la gestion des risques.

- ***Crystal***: il met l'accent sur la collaboration entre les membres de l'équipe de développement, la communication et l'adaptabilité aux changements. Le modèle Crystal est adaptable en fonction de la taille de l'équipe et de la complexité du projet.

Il est important de noter que chaque modèle de cycle de vie du logiciel peut être adapté pour répondre aux besoins spécifiques d'un projet.

Conclusion

Ce chapitre a couvert un certain nombre de points clés dans le domaine du Génie Logiciel. Le Génie Logiciel est un domaine complexe et important qui nécessite une attention particulière à de nombreux aspects différents. En suivant des processus rigoureux et en travaillant de manière méthodique, les développeurs de logiciels peuvent créer des produits de haute qualité qui répondent aux besoins de leurs utilisateurs de manière efficace et efficiente.

Les pionniers du génie logiciel ont également développé des méthodes et des langages pour évaluer et améliorer la qualité et la robustesse des logiciels. Au cours des années, ces méthodes et ces langages se sont perfectionnés et les outils et technologies de génie logiciel sont devenus de plus en plus sophistiqués.

2.Chapitre 02/ Modélisation avec UML

Introduction

UML (Unified Modeling Language) est un langage visuel standard utilisé en génie logiciel pour décrire, spécifier, concevoir et documenter des systèmes logiciels. C'est une collection de notations graphiques et de techniques de modélisation qui aident les développeurs de logiciels et les parties prenantes à visualiser, communiquer et comprendre les systèmes logiciels.

UML est le résultat de la fusion de trois méthodes d'analyse orientées objet :

1. La méthode OOD, *Object Oriented Design*, de G.Booch. Elle a été conçue à la demande du Ministère de la Défense des USA. Elle définit *7 types de diagrammes* pour représenter un système orienté objet,
2. La méthode OMT, *Object Modeling Technique*, a été mise au point à General Electric. Elle décrit le système sous forme *de vues statiques, dynamiques et fonctionnelles* ;
3. La méthode OOSE, *Object Oriented Software Engineering*, (Origine : universitaire (informatique temps réel) et industrielle (Ericsson)). Elle Met l'accent sur l'analyse fondée *sur les cas d'utilisation*.

UML est largement utilisé dans les projets de développement de logiciels comme moyen de communication entre les développeurs, les concepteurs et les parties prenantes. Il aide à identifier et à clarifier les exigences du système logiciel, ainsi qu'à concevoir et documenter son architecture et son comportement.

2.1. Notions : Méthode, Modèle, Langage et Diagramme[7]

La construction d'un système d'information, d'un réseau, d'un *logiciel complexe*, de taille importante et par de nombreuses personnes oblige à *modéliser*.

La modélisation est une technique d'ingénierie qui permet de comprendre un système par l'établissement de *modèles* pour mettre au point une solution à un problème.

Un modèle est une abstraction permettant de mieux comprendre un objet complexe (bâtiment, économie, atmosphère, cellule, logiciel, etc.).

En informatique, un modèle sert :

- De document d'échange entre clients et développeurs
- D'outil de conception

[7] https://home.mis.u-picardie.fr/~furst/docs/1-UML_Besoins.pdf

- De référence pour le développement
- De référence pour la maintenance et l'évolution

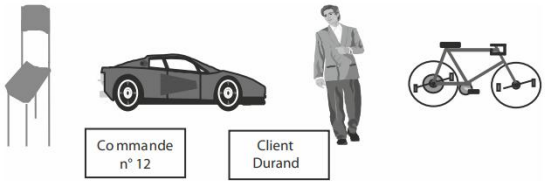
Langage de modélisation : une syntaxe commune, graphique, pour modéliser.

Méthode : procédé permettant de construire un modèle aussi correct que possible et aussi efficacement que possible (MERISE, …).

Un diagramme est la représentation graphique d'un modèle.

2.2. Les éléments et mécanismes généraux

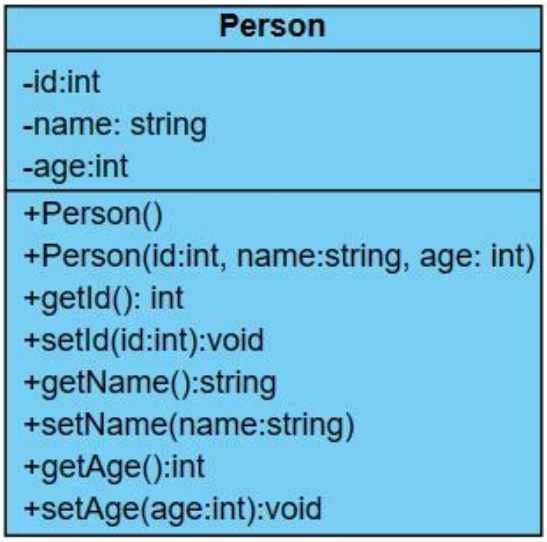
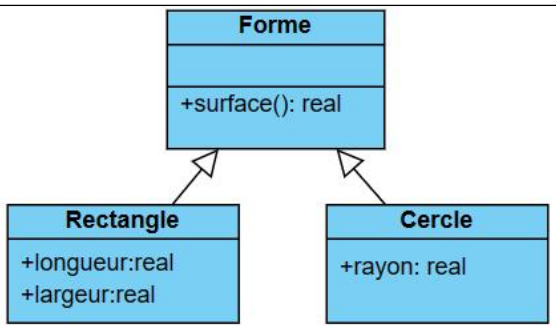
La modélisation orientée objet est une approche qui permet de structurer le code en utilisant des concepts issus du monde réel, tels que les objets, les classes, les attributs et les méthodes. Voici les concepts de base de la modélisation orientée objet :

<table>
<tr><th colspan="3">Concepts de bases</th></tr>
<tr><th>Concept</th><th>Description</th><th>Exemple</th></tr>
<tr><td>Objet</td><td>Un objet est une entité identifiable du monde réel.
Il peut avoir une existence physique (une chaise, une voiture, un client, etc.) ou ne pas en avoir (une commande).
Identifiable signifie que l´objet peut être désigné.
Tout objet a un ensemble d´attributs et un ensemble de méthodes.</td><td> 

Source: [8]</td></tr>
<tr><td>Classe</td><td>Une classe décrit un groupe d'objets ayant :
les mêmes propriétés (attributs),
un même comportement (opérations), et
une sémantique commune (domaine de définition).</td><td rowspan="3">Cette classe s'appelle "Person" et a trois attributs : "id", "name" et "age".

La classe possède également plusieurs méthodes pour accéder et modifier ces attributs.</td></tr>
<tr><td>Attribut</td><td>Un attribut est une propriété élémentaire d'une classe.
Pour chaque objet d'une classe, l'attribut prend une valeur (sauf cas d'attributs multi-valués)</td></tr>
<tr><td>Opération</td><td>Une opération est une fonction</td></tr>
</table>

[8] Gabay, Joseph, and David Gabay. UML 2 Analyse et conception: Mise en œuvre guidée avec études de cas. Dunod, 2008

	applicable aux objets d'une classe. Elle permet de décrire le comportement d'un objet. Une méthode est l'implémentation d'une opération.	
Mécanismes généraux		
Mécanisme	**description**	**Exemple**
Encapsulation	L'encapsulation est un principe de l´approche orientée objet qui consiste à regrouper les données (attributs) et les méthodes qui les manipulent au sein d'une même classe, afin de les protéger contre les accès non autorisés.	Les attributs sont représentés avec un tiret devant leur nom pour indiquer qu'ils sont privés. Les méthodes commençant par un "+" sont des méthodes publiques, c'est-à-dire qu'elles peuvent être appelées à partir de l'extérieur de la classe. Le constructeur de la classe prend trois arguments : id, name et age. Les méthodes setId(), setName() et setAge() sont utilisées pour définir les valeurs des attributs privés.
Héritage	L'héritage est un mécanisme de l´approche objet qui permet de créer une nouvelle classe en utilisant une classe existante comme modèle. La nouvelle classe hérite des attributs et des méthodes de la classe parente.	 Dans cet exemple, la classe Forme est la classe parente qui contient une méthode surface() qui calcule la surface de la forme.

		Les classes Rectangle et Cercle héritent de la classe Forme et ajoutent des attributs spécifiques : longueur et largeur pour Rectangle, et rayon pour Cercle.
Polymorphisme	Le polymorphisme signifie que la même méthode peut avoir des comportements différents dans des classes différentes.	Dans l´exemple précédent, les classes Rectangle et Cercle peuvent également redéfinir la méthode calculerSurface() pour calculer la surface spécifique à leur forme. La flèche creuse indique que Rectangle et Cercle héritent de la classe Forme.
Abstraction	L'abstraction est un principe de l´approche objet qui consiste à définir des interfaces claires et simples pour les classes et les objets, afin de masquer la complexité interne et de faciliter l'utilisation du code.	

2.3. Diagrammes d'UML

UML fournit un ensemble de diagrammes normalisés qui peuvent être utilisés pour représenter différents aspects d'un système logiciel, tels que les exigences, la structure, le comportement et les interactions entre les composants. Les créateurs d'UML ont organisé les diagrammes en fonction de trois axes pour orienter leurs modélisations (Figure suivante[9]) :

- L'axe fonctionnel représente les actions effectuées par le système à construire.

- L'axe structurel et statique décrit la configuration du système.

- L'axe dynamique se concentre sur la construction des fonctionnalités du système.

[9] Roques, Pascal. UML 2 par la pratique: Etudes de cas et exercices corrigés. Editions Eyrolles, 2008.

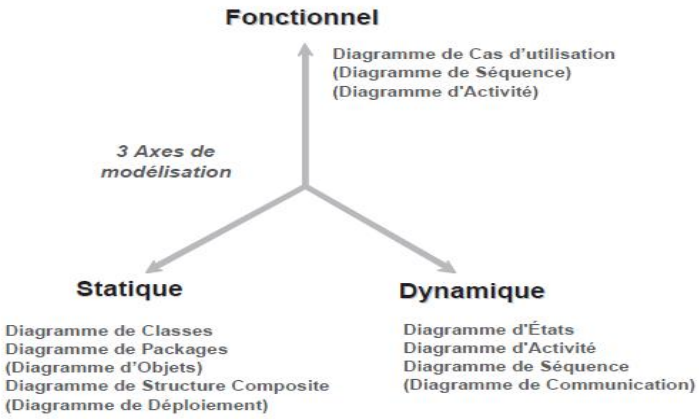


Figure 1: Les axes d´UML

UML 2.0 offre divers types de diagrammes permettant de représenter graphiquement et de gérer les différents éléments de la modélisation:

- ***Le diagramme des cas d'utilisation :*** représente les fonctionnalités (ou dit cas d'utilisation) nécessaires aux utilisateurs.

- ***Diagramme de classes :*** dans la phase d'analyse, ce diagramme représente les entités (des informations) manipulées par les utilisateurs. Dans la phase de conception, il représente la structure objet d'un développement orienté objet.

- ***Le diagramme d'objet*** **:** sert à illustrer les classes complexes en utilisant des exemples d'instances. Une instance est un exemple concret de contenu d'une classe. En illustrant une partie des classes avec des exemples (grâce à un diagramme d'objets), on arrive à voir un peu plus clairement les liens nécessaires.

- *Le diagramme de séquences* **:** permet de décrire les différents scénarios d'utilisation du système.

- *Le diagramme d'activités :* représente le déroulement des actions, sans utiliser les objets. En phase d'analyse, il est utilisé pour consolider les spécifications d'un cas d'utilisation.

- *Le diagramme de collaboration* (appelé également diagramme de communication) : Il permet de mettre en évidence les échanges de messages entre objets. Cela nous aide à voir clair dans les actions qui sont nécessaires pour produire ces échanges de messages. Et donc de compléter, si besoin, les diagrammes de séquence et de classes.

- *Le diagramme état-transition* *:* Il permet de décrire le cycle de vie des objets d'une classe.

- ***Le diagramme global d'interaction*** : permet de donner une vue d'ensemble des interactions du système. Il est réalisé avec le même graphisme que le diagramme

d'activité. Chaque élément du diagramme peut ensuite être détaillé à l'aide d'un diagramme de séquence ou d'un diagramme d'activité.

- Le diagramme de temps : est destiné à l'analyse et la conception de systèmes ayant des contraintes temps-réel. Il s'agit là de décrire les interactions entre objets avec des contraintes temporelles fortes.

- Le diagramme de structure composite : décrit un objet complexe lors de son exécution.

- Le diagramme de composants : décrit tous les composants utiles à l'exécution du système (applications, librairies, instances de base de données, exécutables, etc.).

- Le diagramme de déploiement : correspond à la description de l'environnement d'exécution du système (matériel, réseau…) et de la façon dont les composants y sont installés.

2.4. Paquetage ou Packages

Un Paquetage (sous modèle) : c'est une notion qui peut apparaître dans beaucoup de diagrammes pour spécifier le regroupement d'éléments au sein d'un sous-système (cas, classes, objets, composants, autres paquetages, ...) et aussi pour encapsuler ces éléments.

Caractéristiques d'un paquetage :

- Les packages peuvent contenir des éléments tels que des classes, des interfaces, des diagrammes, des sous-packages, des commentaires et des contraintes.

- Les packages peuvent être organisés en une hiérarchie, ce qui permet de définir des relations d'importation et de dépendance entre eux.

- Les packages peuvent être nommés et utilisent une notation de nommage en point pour refléter leur position dans la hiérarchie.

- Les packages peuvent être utilisés pour encapsuler des éléments, ce qui permet de définir leur portée et leur visibilité dans le modèle.

- Les packages peuvent également être utilisés pour représenter des groupes fonctionnels ou des sous-systèmes, ce qui permet de mieux comprendre la structure et les interactions du système dans son ensemble.

- 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 un seul paquetage racine qui détient la totalité du modèle d'un système.

les packages sont un élément important de la modélisation UML, car ils permettent de gérer la complexité des modèles en les organisant en unités logiques plus grandes et plus cohérentes.

Complément du chapitre 2[10]

Q1) Une classe décrit uniquement les attributs de type objet.

Non. Elle décrit aussi son comportement sous forme d'opérations.

Q2) Quelle phrase détermine une relation d'héritage ?

- Un cheval est un type d'animal

- Un animal est un type de cheval

Un cheval est un type d'animal,

Q3) Quelle est la différence entre une classe abstraite et une classe concrète ?

Une classe concrète possède des instances tandis qu'une abstraite ne peut pas en posséder.

Q4) A quoi sert de définir des classes abstraites dans un modèle ?

Une classe abstraite a pour vocation de posséder des sous-classes concrètes et sert à factoriser des attributs et méthodes communs à ses classes.

Q5) Quel type de contraintes peut-on appliquer à une relation d'héritage ?

UML offre quatre contraintes pouvant être appliquées à la relation d'héritage :

{incomplete} signifiant que l'ensemble des sous-classes est incomplet et qu'il ne couvre pas la surclasse.

{complete} signifiant que l'ensemble des sous-classes est complet et qu'il couvre la surclasse.

{disjoint} signifiant que les sous-classes n'ont aucune instance en commun.

{overlapping} signifiant que les sous-classes peuvent avoir au moins une instance commune.

Q6) Quelle notion de l'approche objet permet de masquer des attributs et des méthodes d'un objet vis-à-vis des autres ?

[10] Les compléments de cours sont extraits de : Laurent DEBRAUWER. Fien VAN DER HEYDE. UML 2.5. Initiation, exemples et exercices corrigés. 5e édition

Cette notion est l'encapsulation. En effet, certains attributs et méthodes ont pour but des traitements internes à l'objet et ne doivent pas être accessibles en dehors de celui-ci. Ces attributs et méthodes encapsulés sont dits privés.

Q7) Qu'est-ce que le polymorphisme ?

Le polymorphisme signifie que la même méthode peut avoir des comportements différents dans des classes différentes.

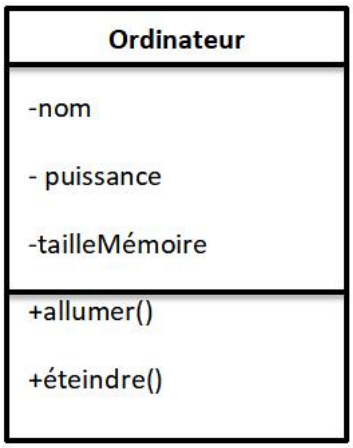
Q8) Qu'est-ce que les stéréotypes UML ?

Les stéréotypes permettent de définir de nouveaux éléments de modélisation par extension d'éléments existants d'UML. Certains stéréotypes sont prédéfinis dans UML, d'autres peuvent être définis par l'utilisateur.

Exercices corrigés[11]

Exercice N°1 :

Représenter une classe qui modélise des ordinateurs et qui contient les attributs suivants : nom, puissance et taille de mémoire. La classe doit également définir les méthodes permettant d'allumer et d'éteindre l'ordinateur.



Exercice N°2 :

L´objectif est de déterminer les liens d'héritage entre des concepts ainsi que les contraintes s'appliquant à ces liens. De ce fait, considérons les deux phrases suivantes :

- Les périphériques de stockage sont deux types amovibles ou non amovibles.

[11]Ces exercices sont extraits de : Laurent DEBRAUWER. Fien VAN DER HEYDE. UML 2.5. Initiation, exemples et exercices corrigés. 5e édition

- Les disquettes, les clés USB, et les disques durs sont des périphériques de stockage.

a) Modélisez ces deux phrases.

b) Complétez les modèles en introduisant des contraintes de couverture (complet ou incomplet) et de recouvrement (disjoint ou non) entre les sous-classes.

Exercice N°3 :

En utilisant le diagramme de classe ci-dessous qui représente des objets graphiques, répondez aux questions et proposez une extension du modèle :

a) Est-ce que l'attribut diamètre peut être accessible pour un objet Carré ?

b) Est-ce que l'objet Cercle possède un attribut couleur ?

c) Est-ce qu'on peut appliquer la méthode déplacer à un objet Point ?

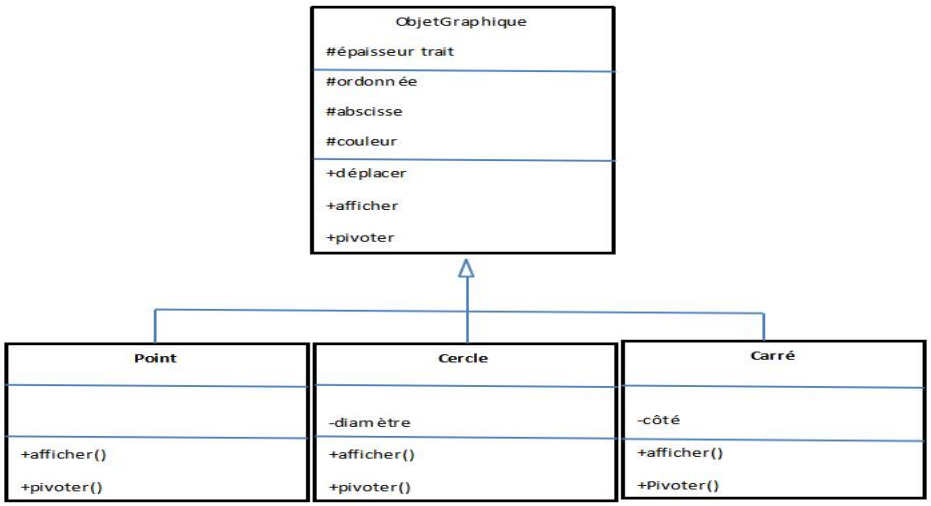
d) Comment la notion de l'approche objet permet-elle d'avoir la méthode pivoter dans toutes les classes du diagramme et quel est son objectif ?

e) Quelle hypothèse est nécessaire pour que la classe ObjetGraphique devienne abstraite ?

f) Pourquoi est-il avantageux que cette classe soit abstraite plutôt que concrète ?

g) Quelle modification du diagramme de classe est nécessaire ?

h) Proposez une extension du modèle pour permettre la création d'un objet graphique composé de plusieurs autres objets graphiques.



Corrigé (Héritage, encapsulation et polymorphisme)

1. L'attribut **Diamètre** est un attribut privé de la classe **Cercle**. Grâce au principe d'encapsulation, il n'est accessible qu'à des objets de sa classe. Un objet **Carré** n'a donc pas accès à cet attribut.
2. La classe **Cercle** est une sous-classe de la classe **ObjetGraphique**. Elle hérite par conséquent de tous les attributs de sa surclasse. L'attribut **couleur** est donc un attribut de **Cercle** hérité de la classe **ObjetGraphique**.
3. De la même manière que pour la question précédente, une méthode est héritée par les sous-classes d'une classe. La méthode **déplacer** est donc héritée dans la classe **Point**.
4. La méthode **pivoter** est présente dans toutes les classes du diagramme et y possède un comportement différent grâce à la notion de polymorphisme.
5. **A.** une classe abstraite est une classe qui ne peut pas posséder d'instances directes. Afin de rendre la classe **ObjetGraphique** abstraite, il faut considérer que cette classe ne décrit que partiellement des objets graphiques.la solution consiste à rendre abstraites les méthodes afficher et pivoter. En effet, il n'est pas possible de dessiner ou de pivoter un objet graphique sans connaître sa nature précise, information qui est fournie par l'une des sous-classes d'**ObjetGraphique**. Par la suite, il est possible de créer de nouvelles sous-classes d'**ObjetGraphique** pour obtenir de nouveaux objets en plus des points, des cercles et des carrés.

   B. il est préférable d'éviter que la classe **ObetGraphique** soit concrète sinon il est obligatoire de devoir spécifier le comportement des méthodes **afficher** et **pivoter**, en donnant alors un comportement par défaut, ce qui ne constitue pas une solution satisfaisante.

   C. En UML, le nom d'une classe abstraite est mis en italique ou en conférant le stéréotype « abstract » à la classe. Les méthodes **afficher** et **pivoter** deviennent aussi abstraites puisqu'elles sont définies dans les sous-classes correspondantes.

6. Les objets graphiques composés sont représentés par une classe **GraphiqueComposé**. Chaque objet graphique composé est alors typé comme un objet graphique. La classe **GraphiqueComposé** est donc liée à la classe **ObjetGraphique** par une relation d'héritage. Un objet graphique composé est composé d'autres objets graphiques. La suppression de celui-ci entraîne la suppression de ses composants. Il y a donc d'une relation de composition entre les classes **GraphiqueComposé** et **ObjetGraphique**.

3.Chapitre 03/ Diagramme UML de cas d'utilisation : Vue fonctionnelle

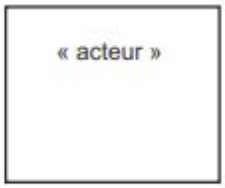
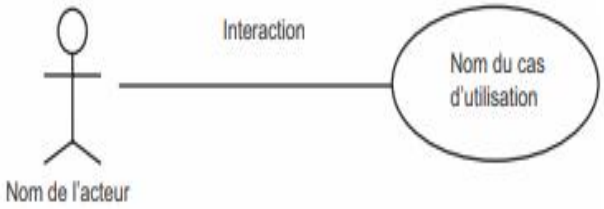
3.1. Diagramme de cas d'utilisation

Tout système peut être décrit par un certain nombre de *cas d'utilisation* correspondant aux besoins exprimés par l'ensemble des utilisateurs. À chaque utilisateur, vu comme *acteur,* correspondra un certain nombre de cas d'utilisation du système. L'ensemble de ces cas d'utilisation se représente sous forme d'un diagramme de cas d'utilisation.

3.1.1. Objectif : Les cas d'utilisation constituent un moyen de recueillir et de décrire les besoins des acteurs du système.

Les cas d'utilisation ont été définis initialement par Ivar Jacobson en 1992 dans sa méthode OOSE.

3.1.2. Concepts de base

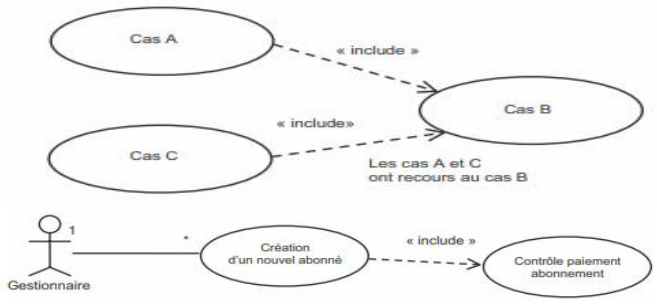
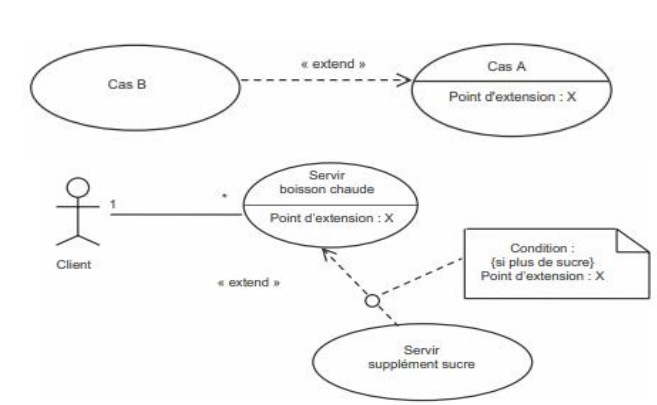
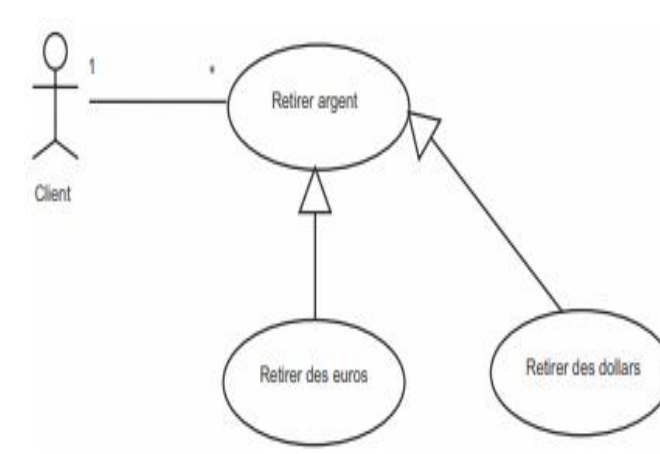
Concept	Description	Formalisme
Acteur	-Un acteur désigne un rôle rempli par une entité externe (tel qu'un utilisateur humain, un appareil physique ou un autre système) qui interagit directement avec le système examiné. - ***Un acteur est extérieur au système.***	 
Cas d'utilisation et interaction	Un cas d'utilisation (use case) représente un ensemble de séquences d'actions qui sont réalisées par le système et qui produisent un résultat observable intéressant pour un acteur particulier. Un cas d'utilisation modélise un service rendu par le système. Il exprime les interactions acteurs/système et apporte une valeur ajoutée	 La présence de cette relation (acteur-cas d´utilisation) indique la présence d'une ***communication*** sous la forme d'une interaction entre l'utilisateur et le système.

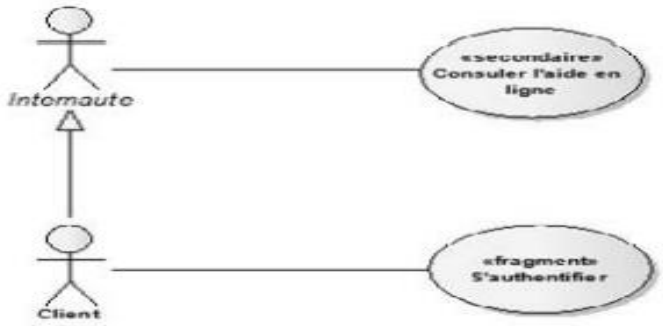
	« notable » à l'acteur concerné. Une interaction permet de décrire les échanges entre un acteur et un cas d'utilisation.	

- Il existe deux types d´acteurs: Primaire et Secondaire

- Un cas d'utilisation correspond à un objectif fonctionnel de l'acteur primaire du cas d'utilisation.

- L'objectif d'un cas d'utilisation ne constitue pas un résultat nécessaire pour un acteur secondaire. Un acteur secondaire interagit avec le cas d'utilisation pour la réalisation de celui-ci mais n'y trouve pas un intérêt direct.

Relations entre cas d'utilisation		
Relation d'inclusion	formalisée par le stéréotype <include> : le cas d'utilisation de base en incorpore explicitement un autre, de façon *obligatoire.*	
Relation d'extension	formalisée par le stéréotype <extend> : le cas d'utilisation de base en incorpore implicitement un autre, de façon *optionnelle.*	
Relation de généralisation entre cas d'utilisation	La relation de généralisation/ spécialisation est une relation liant deux cas d'utilisation. Si un premier cas d'utilisation spécialise un second, alors il en hérite la description qu'il peut alors enrichir. Il est appelé un sous-cas d'utilisation (comme une sous-classe spécialise une autre classe).	

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

3.2. Description textuelle de cas d'utilisation

Chaque cas d'utilisation doit être décrit sous forme textuelle afin de bien identifier les traitements à réaliser par le système en vue de la satisfaction du besoin exprimé par l'acteur.

Points	**Description**	**Exemple**
Objectif	Décrire succinctement le contexte et les résultats attendus du cas d'utilisation.	Cas d'utilisation : effectuer une commande
Acteurs concernés	Le ou les acteurs concernés par le cas doivent être identifiés en précisant globalement leur rôle.	Acteurs : Client
Préconditions	Si certaines conditions particulières sont requises avant l'exécution du cas, elles sont à exprimer à ce niveau.	Préconditions : Le client doit s'authentifier au site Ce cas d'utilisation commence lorsqu'un client sélectionne des produits et clique sur le bouton « commander »
Post-conditions	Par symétrie, si certaines conditions particulières doivent être réunies après l'exécution du cas, elles sont à exprimer à ce niveau.	Post-condition : Commande ajoutée
Scénario nominal	Il s'agit là du scénario principal qui doit se dérouler sans incident et qui permet d'aboutir au résultat souhaité.	Scénario nominal : - Enchaînement (1) : Le système demande au client de saisir son pseudonyme et son mot de passe, - Le client saisit son pseudonyme et son mot de passe et valide × --

		-Enchaînement (2) : Le client effectuer une commande - Le système affiche la commande et indique le montant total - Le client confirme la commande -Enchaînement (3) : Le système demande au client de choisir un mode de paiement - Le client saisit ses informations de paiement et valide -Enchaînement (3) : Le système demande au client de choisir une adresse de livraison - Le client saisit une adresse pour recevoir la commande et valide
Scénario alternatifs	Les autres scénarios, secondaires ou correspondants à la résolution d'anomalies, sont à décrire à ce niveau. Le lien avec le scénario principal se fait à l'aide d'une numérotation hiérarchisée (1.1a, 1.1b…) rappelant le numéro de l'action concernée.	- Enchaînement (1-a) : - Si les informations sont incorrectes [Exception 1 : MP/PseudoIncorrestes] - Enchaînement (2-a) : Le client ajoute un produit au panier - Enchaînement (2-b) : Supprimer un produit du panier – - Enchaînement (3-a) : - Si les informations de paiement sont incorrectes [Exception 2 : InfoPaiementIncorectes]
Exceptions		Exceptions -[Exception 1 : MP/PseudoIncorrestes] : un message d'erreur est affiché sur l'écran avisant le client que le nom d'utilisateur ou le mot de passe sont incorrectes. -[Exception 2 : InfoPaiementIncorectes] : un message d'erreur est affiché sur l'écran avisant le client que les informations de paiement sont incorrectes.

Complément du chapitre 3

Q1) Que décrivent les cas d'utilisation ?

Les cas d'utilisation décrivent les exigences fonctionnelles d'un système à réaliser.

Q2) Les cas d'utilisation correspondent à un ensemble d'interaction entre un utilisateur et le système.

Oui, un cas d'utilisation décrit les interactions entre un utilisateur et le système. En UML, la description est graphique et se limite à indiquer le nom du cas d'utilisation sans détailler les interactions.

Q3) Dans un cas d'utilisation, un acteur représente un utilisateur jouant un rôle précis dans l'utilisation du système.

Oui, le terme acteur est préféré à celui d'utilisateur pour bien distinguer la personne du rôle qu'elle joue dans le cadre d'un cas d'utilisation. En effet, la même personne (donc le même utilisateur) peut intervenir dans un même cas pour des rôles différents. Cet utilisateur correspond alors à plusieurs acteurs distincts.

Q4) Un acteur est obligatoirement une personne physique.

Non, un acteur peut être un autre système, Par exemple, le système informatique d'entreprise de distribution en gros peut être sollicité par le système d'un détaillant pour connaître les délais de réapprovisionnement ou les prix des articles. Le système d'un détaillant constitue un acteur pour ces cas d'utilisation.

Q5) Tous les cas d'utilisation ont une relation de communication directe avec un acteur.

Non, il existe des cas d'utilisation qui n'ont pas de relations de communication directe avec un acteur primaire. Ces cas d'utilisation sont destinés à être liés à d'autres cas d'utilisation avec une ou plusieurs relations d'inclusion ou d'exclusion.

Q6) Qu'est-ce qu'un cas d'utilisation abstrait ?

Un cas d'utilisation qui n'est pas mis en œuvre directement mais dont le but est d'être spécialisé est appelé un cas d'utilisation abstrait.

Q7) Lors du déroulement d'un projet, à quels moments les cas d'utilisation sont-ils utilisés ?

La phase de validation vérifie que le système conçu répondre aux sollicitations des utilisateurs conformément aux différents cas d'utilisation.

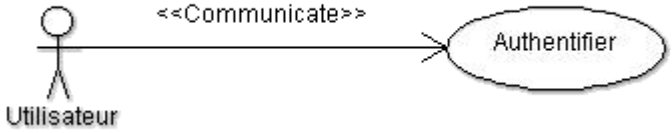
Exercices corrigés[12]

Exercice N°1 :

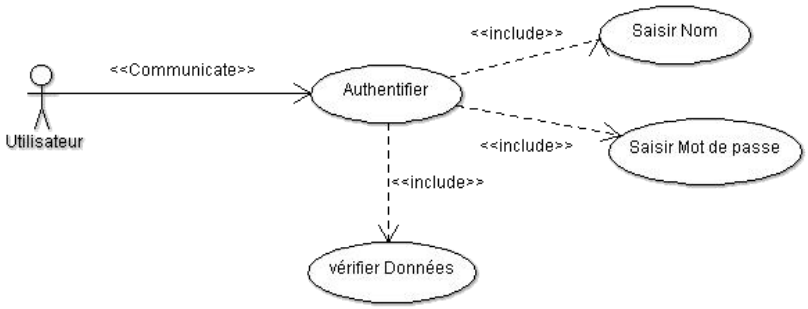
L´objectif est de décrire le cas d'utilisation de l'identification d'un utilisateur à un système informatique. Cette identification s'effectue de façon simple par la saisie d'un nom et d'un mot de passe.

- Modélisez, en UML, le cas d'utilisation de l'identification d´un acteur qui interagit avec ce dernier, sans spécifier les détails.

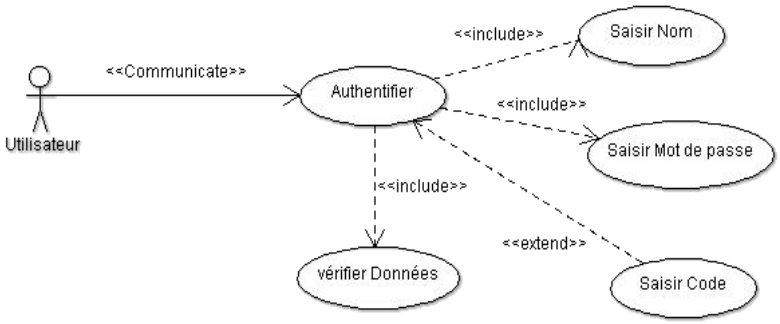
[12] Ces exercices sont extraits de : Laurent DEBRAUWER. Fien VAN DER HEYDE. UML 2.5. Initiation, exemples et exercices corrigés. 5e édition



- Introduisez, dans le premier diagramme du cas d'utilisation, la saisie du nom et celle du mot de passe ainsi que la vérification de ces données.



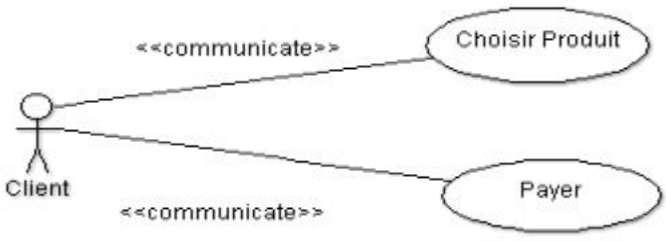
- Ajoutez la saisie optionnelle d'un code complémentaire après celle du mot de passe.



Exercice N°2 :

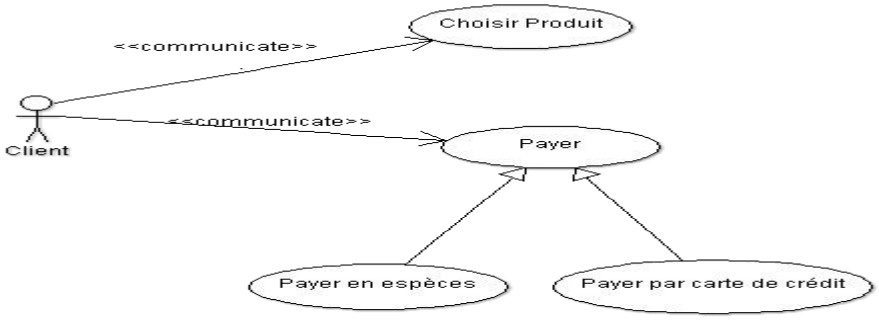
L´objectif est de modéliser le fonctionnement d´un distributeur de produits courants où le client du distributeur peut y trouver des produits alimentaires (pains, conserves, boissons, etc.) ainsi que d'autres types de produits courants (lessives, savons, etc.). Une fois qu'il a choisi les produits qu'il désire acheter, il doit ensuite payer ses achats.

La figure suivante illustre le cas d'utilisation correspondant à cette description.

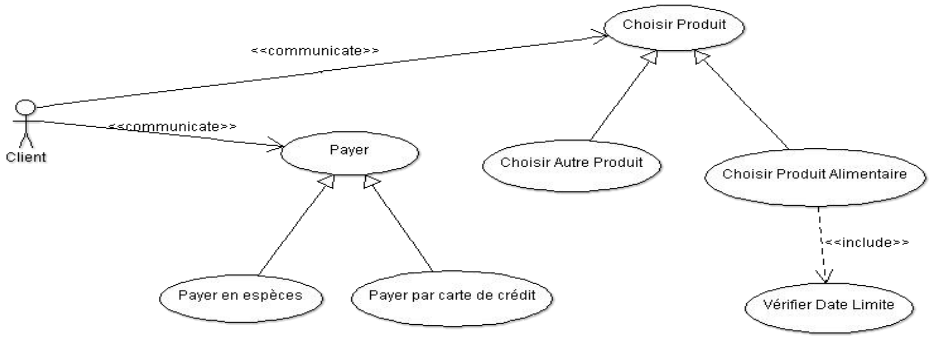


Représentez, en complétant le diagramme précédent:

- Il existe deux façons de payer les produits : soit en espèces soit par carte de crédit.



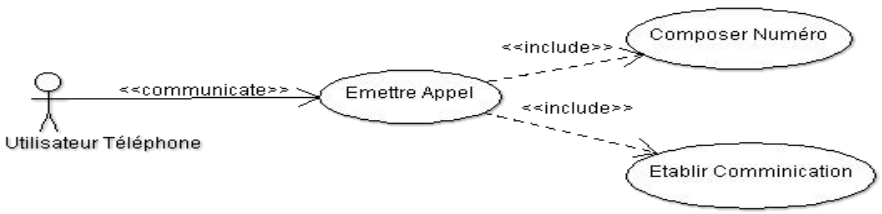
- Lors de l'achat d'un produit alimentaire et uniquement dans ce cas, le client vérifie la date de limite de consommation du produit.



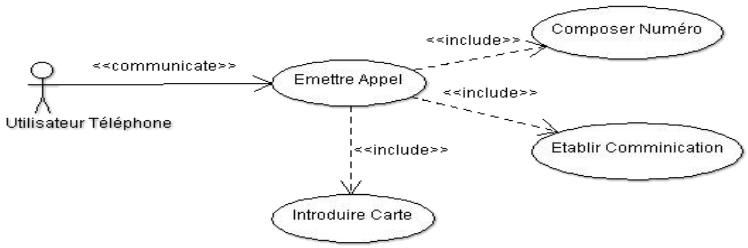
Exercice N°3 :

Le but de cet exercice est de décrire et de détailler le cas d'utilisation d'émission d'un appel téléphonique. Il convient de préciser qu'une phase de numérotation est nécessaire avant l'établissement de la communication avec le correspondant. Cette phase d'établissement de la communication a toujours lieu.

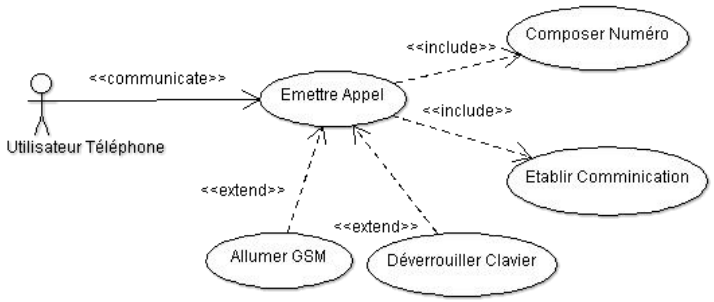
- Représentez en UML le cas d'utilisation correspondant à l'émission d'un appel.



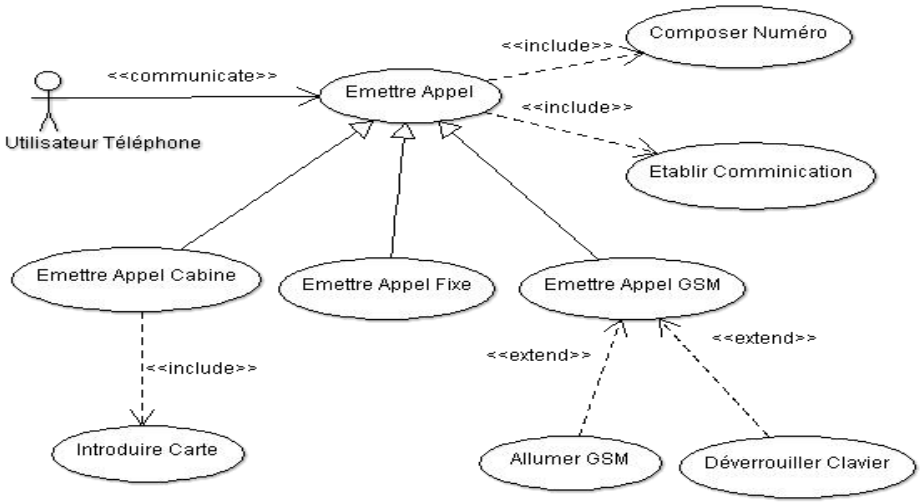
- Le cas d'utilisation précédent s'applique bien à une ligne téléphone fixe. Adaptez-le pour une cabine téléphonique, où il est nécessaire d'introduire une carte téléphonique avant de pouvoir émettre un appel.



- Modifiez également le cas d'utilisation pour un GSM. Il peut être alors nécessaire d'allumer au préalable celui-ci ou d'en déverrouiller le clavier.



- Reprenez ces trois cas et généralisez-les par un cas d'utilisation abstrait.



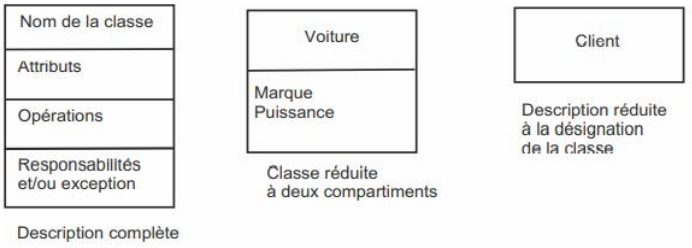
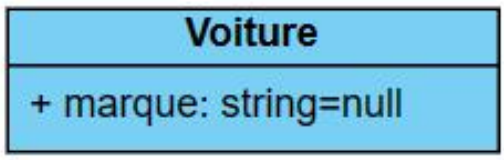
4.Chapitre 04/ Diagrammes UML de classe et d'objet : Vue statique

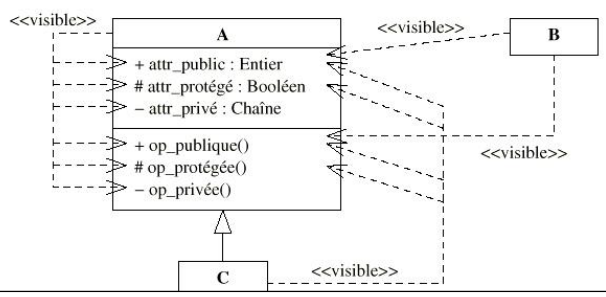
4.1. Diagramme de classes

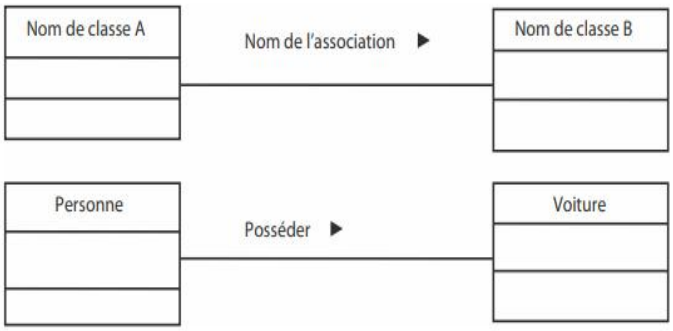
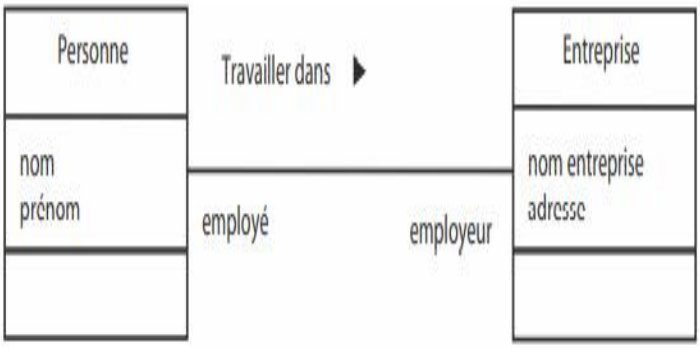
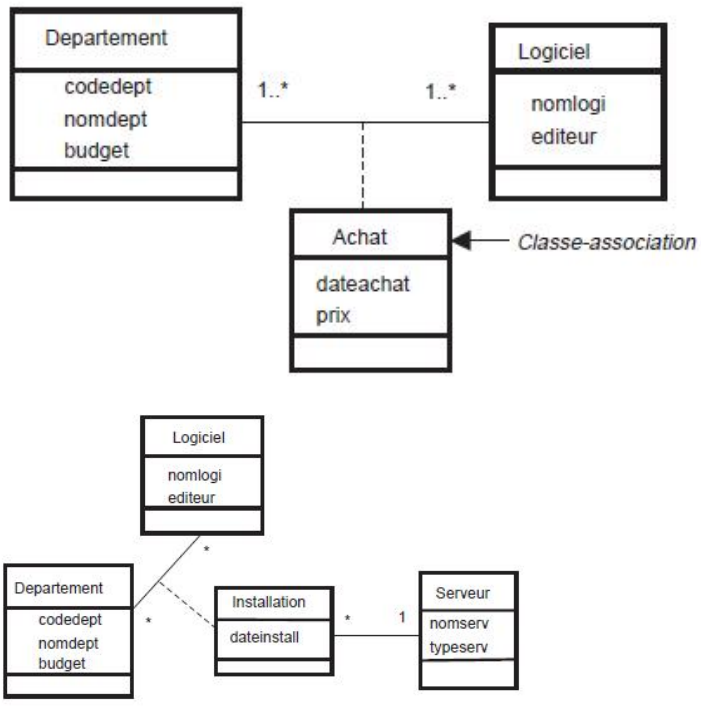
Le diagramme de classe constitue l'un des pivots essentiels de la modélisation avec UML.

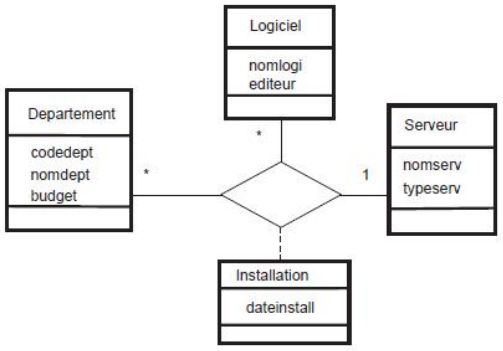
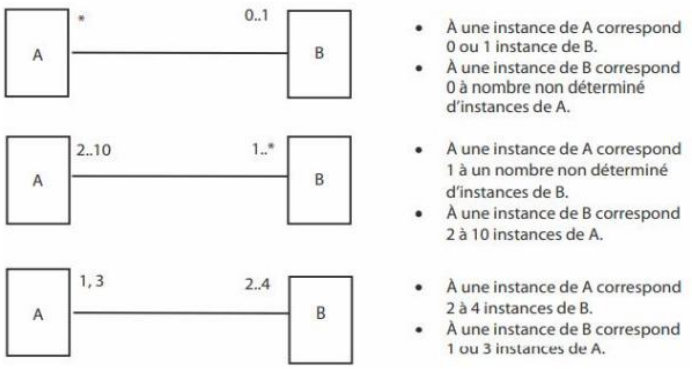
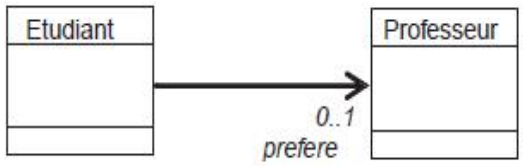
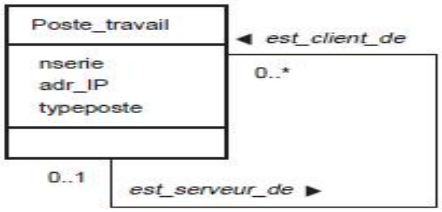
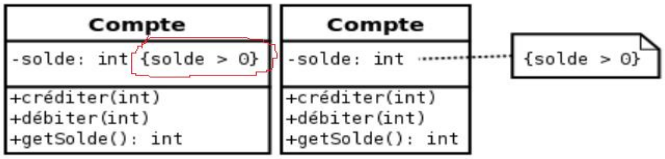
4.1. Objectif : il permet de donner la représentation statique du système à développer.

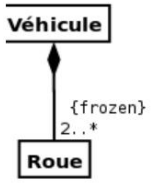
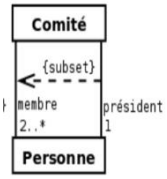
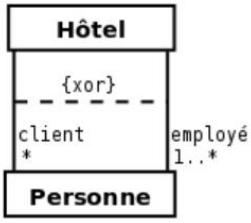
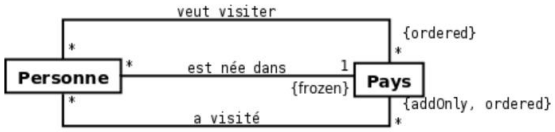
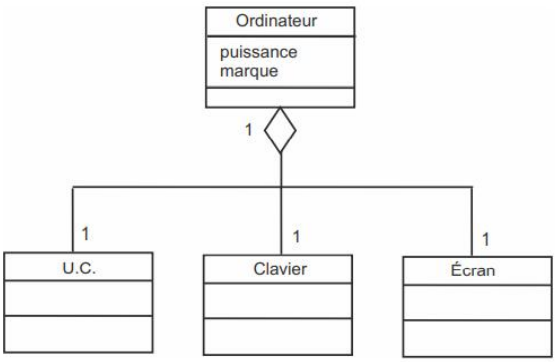
4.2. Les concepts de base

Concept	Description	Formalisme /Exemple
Classe	Une classe décrit un groupe d'objets ayant : les mêmes propriétés (attributs), un même comportement (opérations), et une sémantique commune (domaine de définition).	
Attribut	**Caractéristiques d´un attribut** **Visibilité/Nom attribut : type [= valeur initiale {propriétés}]** – Visibilité : se reporter aux explications données plus loin sur ce point. – Nom d'attribut : nom unique dans sa classe. – Type : type primitif (entier, chaîne de caractères…) dépendant des types disponibles dans le langage d'implémentation ou type classe matérialisant un lien avec une autre classe. – Valeur initiale : valeur facultative donnée à l'initialisation d'un objet de la classe. – {propriétés} : valeurs marquées facultatives (ex. :	 Dans cet exemple, le signe "+" devant le nom de l'attribut indique que celui-ci est public, tandis que le signe "-" indiquerait une visibilité privée. Le type de données est indiqué après le nom de l'attribut, suivi éventuellement d'une valeur par défaut. La multiplicité peut être représentée sous la forme d'un intervalle (par exemple, 0..* pour indiquer que l'attribut peut avoir plusieurs valeurs) mais dans notre exemple, la multiplicité est 1 car chaque voiture ne peut avoir qu'une seule marque.

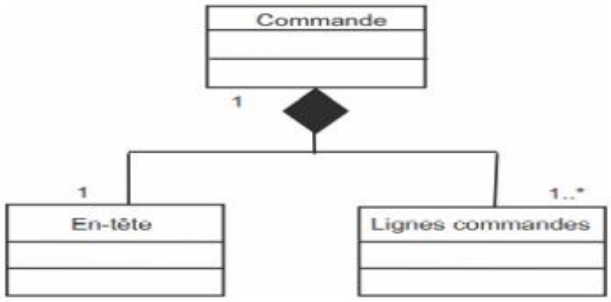
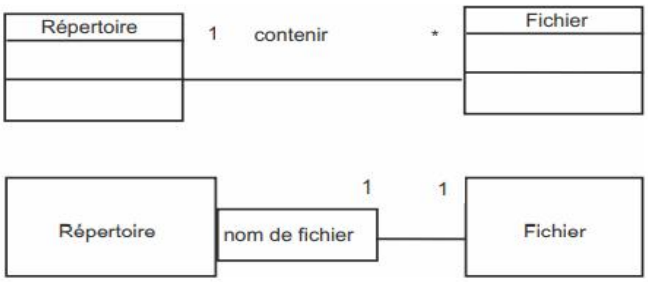
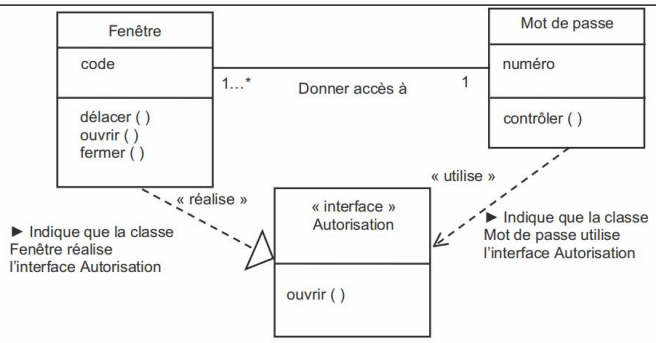
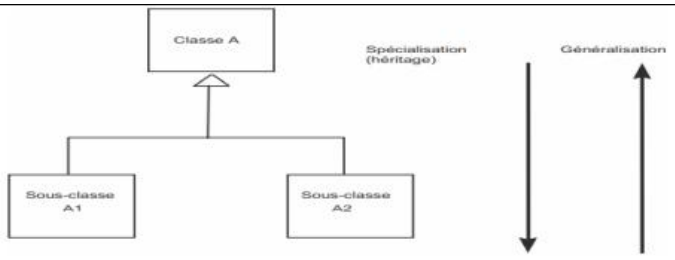
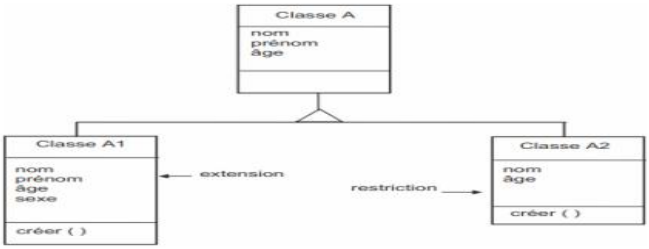
	« interdit » pour mise à jour interdite).	
Opération	**Caractéristiques d´une opération** **Visibilité Nom d'opération (paramètres) [:[type résultat] {propriétés}]** – Visibilité : se reporter aux explications données plus loin sur ce point. – Nom d'opération : utiliser un verbe représentant l'action à réaliser. – Paramètres : liste de paramètres (chaque paramètre peut être décrit, en plus de son nom, par son type et sa valeur par défaut). L'absence de paramètre est indiquée par ( ). – Type résultat : type de (s) valeur(s) retourné(s) dépendant des types disponibles dans le langage d'implémentation. Par défaut, une opération ne retourne pas de valeur, ceci est indiqué par exemple par le mot réservé « void » dans le langage C++ ou Java. – {propriétés} : valeurs facultatives applicables (ex. : {query} pour un comportement sans influence sur l'état du système).	 Dans cet exemple, le signe "+" devant le nom de l'opération indique que celle-ci est publique, tandis que le signe "-" indiquerait une visibilité privée. Les paramètres de l'opération peuvent être indiqués entre les parenthèses, et le type de retour est indiqué après le nom de l'opération, suivi éventuellement d'une liste de paramètres entre parenthèses. L´opération "Demarrer" ne nécessite pas de paramètres et ne renvoie aucune valeur, donc nous indiquons simplement les parenthèses vides et le type de retour "Void".
Visibilité des attributs et opérations		
Public (+)	Attribut ou opération visible par tous.	
Protégé (#)	Attribut ou opération visible seulement à l'intérieur de la classe et pour toutes les sous-classes de la classe.	
Privé (-)	Attribut ou opération seulement visible à l'intérieur de la classe.	

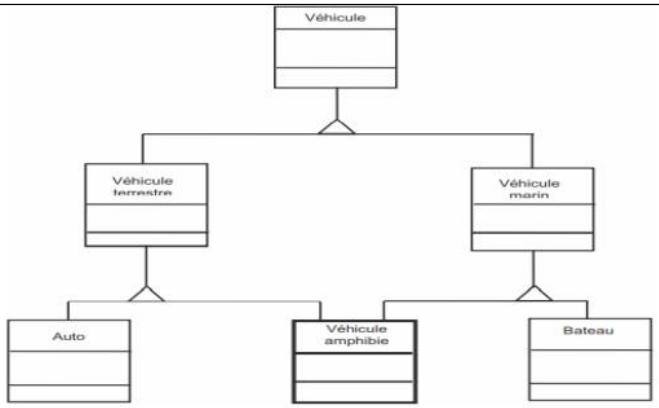
Paquetage (~)	Attribut ou opération ou classe seulement visible à l'intérieur du paquetage où se trouve la classe.	
Lien, Association, Multiplicité et Navigabilité		
Lien et association	Un lien est une connexion physique ou conceptuelle entre objets. Une association décrit *un groupe de liens* ayant *une même structure et une même sémantique.* Un lien est une instance d'une association. Chaque association peut être identifiée par son nom.	
Rôle d'association	Les noms spécifiés pour les extrémités d'une association représentent le rôle joué par les instances des classes dans l'association. Un rôle a la même nature qu'un attribut dont le type serait la classe située à l'autre extrémité de l'association.	
Classe-association	Les classes associations permettent de décrire les liens entre classes sous forme d'une classe. Une classe-association a le statut d'une classe décrivant les occurrences d'une association. Elle peut être dotée d'attributs, d'opérations, et être reliée à d'autres classes par des associations.	

Association n-aire	Une association n-aire connecte n classes. Une association n-aire peut également posséder ou non des attributs.	
Multiplicité	Multiplicité (multiplicity) : chaque extrémité d'une association porte une indication de multiplicité. Elle exprime le nombre minimum et maximum d'objets d'une classe qui peuvent être reliés à des objets d'une autre classe.	
Navigabilité	Par défaut, une association UML est navigable dans les deux sens. UML 2 permet d'exprimer dans un diagramme que les instances d'une classe ne connaissent pas les instances d'une autre bien qu'étant reliées entre elles dans le cadre de l'association.	 La flèche représente le sens de navigation de l'association.
Association réflexive	Une association réflexive est une association qui fait intervenir au moins deux fois la même classe.	
Contraintes		
Contrainte	Une contrainte est une exigence ou une limite sémantique formulée sous forme d'instruction dans un langage textuel, qu'il soit naturel ou formel. Elle peut être appliquée à tout élément ou liste d'éléments d'un modèle et doit être respectée pour une	 - La contrainte **{solde>0}** précise que l´attribut solde doit être positif.

	mise en œuvre correcte du système.	 - **{frozen}** précise que le nombre de roues d'un véhicule ne peut pas varier.  - **{subset}** précise que le président est également un membre du comité.  - **{xor}** (ou exclusif) précise que les employés de l'hôtel n'ont pas le droit de prendre une chambre dans ce même hôtel.  - **{ordered}** exprime qu´une personne a visité un certain nombre de pays, dans un ordre donné, - **{addOnly}** exprime que le nombre de pays visités ne peut que croître ; Source: [UML 2 - de l'apprentissage à la pratique (developpez.com)]
Agrégation et composition entre classes[13]		
Agrégation	l'agrégation est un cas particulier d'association associant un objet complexe aux objets qui le constituent.	

[13] Les figures de ces sections sont extraites de : Gabay, Joseph, and David Gabay. UML 2 Analyse et conception: Mise en œuvre guidée avec études de cas. Dunod, 2008.

Composition	La composition est une forme d'agrégation impliquant deux contraintes supplémentaires : -Un composant n'appartient qu'à un seul objet composé, -La destruction d'un objet composé entraîne celle de ses composants	
Association qualifiée, dépendance et classe d'interface[8]		
Association qualifiée	une association qualifiée est une association qui permet de restreindre les objets référencés dans une association grâce à une clé.	
Dépendance	La dépendance entre deux classes est utilisée pour illustrer l'existence d'un lien sémantique entre elles.	 On dit qu'une classe B dépend d'une classe A lorsque des éléments de la classe A sont requis pour construire la classe B.
Classe d'interface	Une classe d'interface sert à définir l'apparence externe d'une classe en spécifiant les opérations accessibles aux autres classes. Cette classe, qui est nommée, ne comprend pas de description des attributs.	
Généralisation et spécialisation	La généralisation est la relation entre une classe et deux autres classes ou plus partageant un sous-ensemble commun d'attributs et/ou d'opérations.	
Extension et restriction de classe	L´extension d´une classe consiste à ajouter des propriétés dans sa sous-classe, La restriction d´une classe correspond à masquer des propriétés dans la sous-	

	classe.	
L'héritage multiple	L'héritage multiple est une situation dans laquelle une même classe hérite de deux ou plusieurs classes parentes distinctes.	

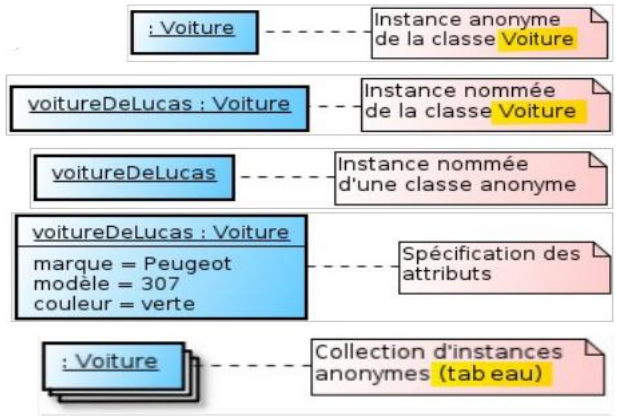
4.2. Diagramme d´objets

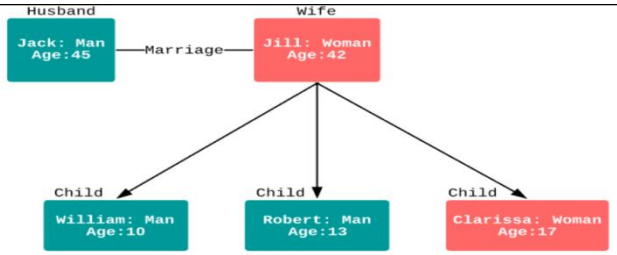
Le diagramme d'objets fait parties des diagrammes structurels (statique).

4.2.1. Objectif

Un diagramme d'objets UML représente une instance spécifique d'un diagramme de classes à un moment précis. Il permet d´affiner le diagramme de classes.

4.2.2. Concepts de base

Concept	Description	Formalisme /Exemple
Objet	Un objet est une instance d'une et une seule classe.	 Source: coursUML4.pdf (remy-manu.no-ip.biz)

Lien	Un lien est une instance d´une association	 Source: Qu'est-ce qu'un diagramme d'objets UML | Lucidchart

Complément du chapitre 4

Q1) Quelle est la différence entre un diagramme de classes et un diagramme d'objets ?

Le diagramme de classes représente l'aspect statique d'un système tandis que le diagramme d'objet montre son aspect dynamique. Ce dernier représente les instances créées et leurs liens lorsque le système est actif. A un diagramme de classes peut correspondre une infinité de diagramme d'objets.

Q2) Les contraintes sont représentées dans un diagramme de classes par :

a. Nom de la contrainte entre parenthèses (nom de la contrainte)

b. Nom de la contrainte entre accolades {nom de la contrainte}

c. Nom de la contrainte précédé par le mot clé constraint : constraint nom de la contrainte.

Q3) Quelle contrainte permet d'exprimer l'exclusion entre les ensembles d'occurrences de deux associations ?

L'exclusion entre deux associations est exprimée par la contrainte {xor}. Elle exprime le fait qu'un objet ne peut être lié que par l'une ou l'autre des associations référencées par la contrainte.

Q4) La notion d'inclusion entre occurrences de deux associations peut-elle être exprimée à l'aide d'une contrainte ?

La notion d'inclusion entre les occurrences de deux associations peut être exprimée grâce à la contrainte {subset}. Elle exprime le fait que les occurrences d'une association sont automatiquement occurrences d'une autre.

Q5) Parmi les contraintes suivantes, laquelle est utilisée pour exprimer un ordre dans une association :

a. {arranged}

b. {ordred}

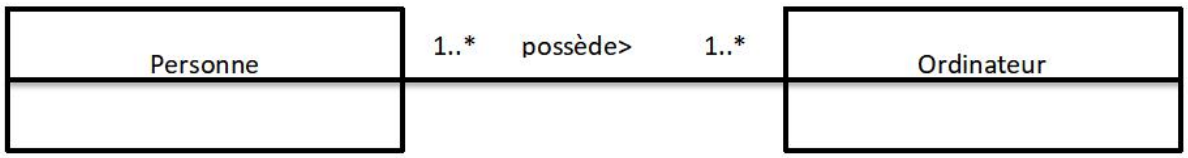
c. {numbred}

La contrainte {ordred}, lorsqu'utilisée à l'extrémité d'une association, désigne un ordre exprimé sur les instances de la classe concernée.

Exercices[14]

Exercice N°1 (avec corrigé) :

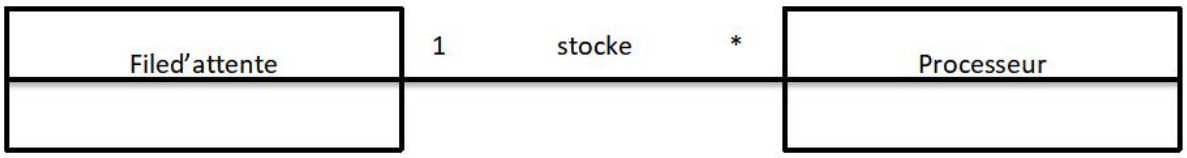
Décrire en UML la phrase suivante : une personne possède un ou plusieurs ordinateurs.



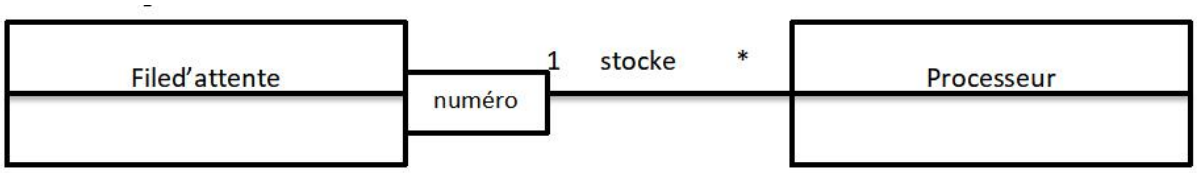
Exercice N°2 (avec corrigé):

Considérons des processus qui s'exécutent dans un ordinateur. Le système de l'ordinateur gère la liste des processus faisant appel à une ressource (comme une imprimante) grâce à une file d'attente. Cette file stocke les processus en attente de la ressource.

- Proposez une modélisation de la file.



- Considérons maintenant que chaque processus dans la file d'attente est identifié de façon unique par un numéro. Proposez une modélisation de la file.

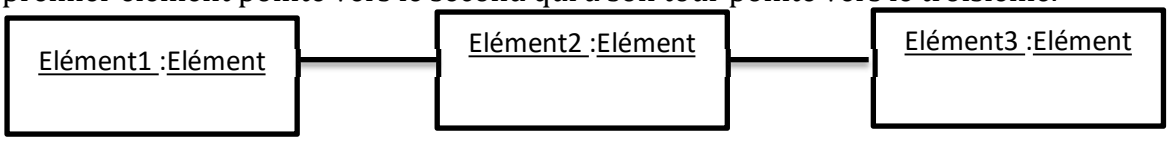


Exercice N°3 (avec corrigé):

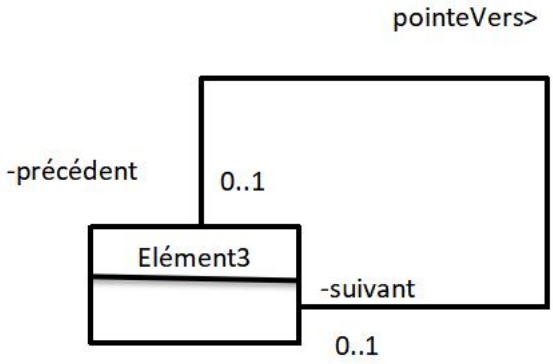
[14] Extraits de : Laurent DEBRAUWER. Fien VAN DER HEYDE. UML 2.5. Initiation, exemples et exercices corrigés. 5e édition

Une liste chaînée est une structure informatique représentant une collection ordonnée d'élément. L'accès aux éléments d'une liste s'effectue de manière séquentielle. Chaque élément de la liste permet l'accès au suivant grâce à un pointeur.

Un exemple de liste composé de trois éléments est représenté dans la figure suivante. Le premier élément pointe vers le second qui à son tour pointe vers le troisième.



- Proposez une modélisation de cette structure.



Exercice N°4 :

L´objectif est de décrire la gestion d'une connexion utilisateur à un serveur de bases de données. Un utilisateur de serveurs de bases de données se connecte avec un accès constitué d'un nom de compte et un mot de passe. La date et l'heure de la connexion doivent être stockées.

- Proposez une représentation sous forme d'un diagramme de classes.

- Complétez le modèle afin que chaque accès puisse disposer d'un espace de travail de taille fixe.

Exercice N°5 :

Un site Web est un ensemble de pages Web liées entre elles par des références appelées hyperliens. Ces pages sont conçues pour être consultées avec un navigateur. Chaque page possède sa propre configuration. La consultation d'une page s'effectue grâce à une URL qui correspond à l'adresse de la page.

Un utilisateur peut, à tout moment, consulter une page Web en utilisant un navigateur. Cette requête est transmise au site Web correspondant. La date et l'heure de chaque connexion doivent être stockées.

- Élaborez le diagramme de classes correspondant à la description.

Exercice N°6 :

Il s'agit de modéliser un diagramme de classes pour la gestion d'une conférence scientifique. Les connaissances du domaine sont résumées sous forme de phrases comme suit :

- La conférence se compose de plusieurs sessions.

- Chaque session possède une date et une heure de début de session,

- Les participants participent à une session soit en tant qu'opérateur soit en tant que public.

- Tous les participants doivent s'inscrire à la conférence. Une inscription peut être annulée ou confirmée.

- Un article scientifique est présenté à une session.

- Un article est soit un article long, soit un article court. Il est composé de sections numérotées et concerne un sujet donné.

- Un auteur peut avoir un ou plusieurs articles présentés à la conférence.

Exercice N°7 :

L´objectif est de modéliser un système de commerce électronique qui a la description suivante:

Un utilisateur (client) qui se connecte à un site de commerce électronique choisit parmi les produits proposés. Il remplit au fur et à mesure un panier virtuel comportant les produits à acheter. A tout moment, il peut ajouter ou supprimer un produit au panier.

Le paiement s'effectue à l'aide d'une carte de crédit. Celle-ci n'appartient pas nécessairement à l'utilisateur. A la fin de la transaction, l'utilisateur peut soit valider sa commande ou l'annuler.

- Décrivez ce système à l'aide d'un diagramme de classes.

5.Chapitre 5/ Chapitre 5. Diagrammes UML : vue dynamique

5.1. Diagramme d'interaction

Les échanges de messages entre objets peuvent être représentés en UML dans deux sortes de diagrammes complémentaires :

Le diagramme de séquence, qui met l'accent sur la *chronologie des messages* ;

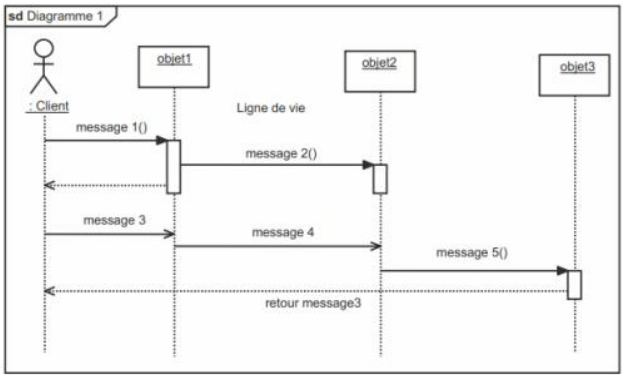
Le diagramme de communication (appelé collaboration en UML 1.x), qui *souligne les relations structurelles entre les participants* qui échangent les messages.

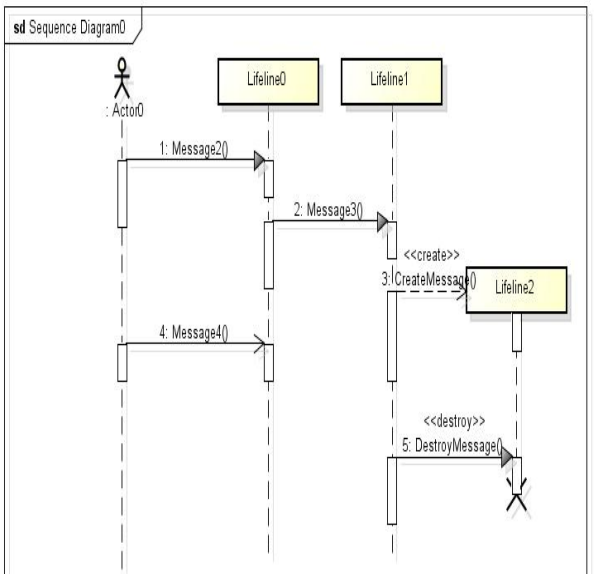
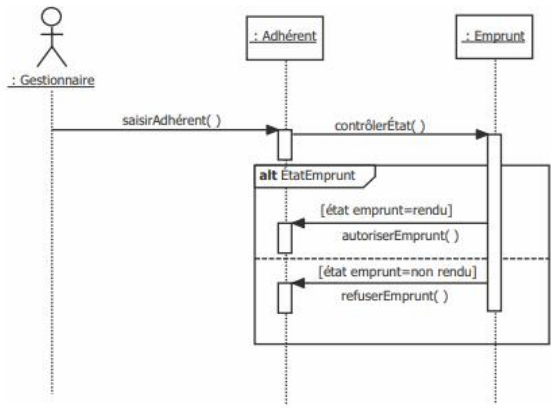
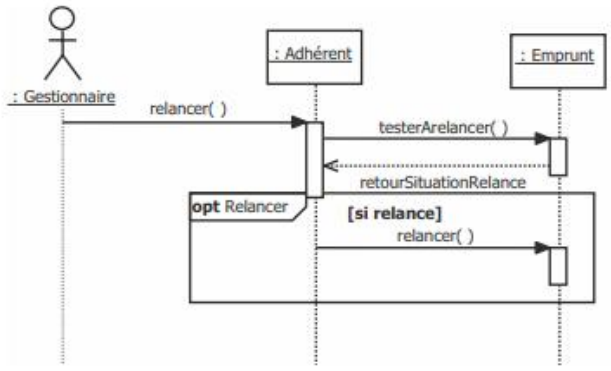
5.1.1. Diagramme de séquence

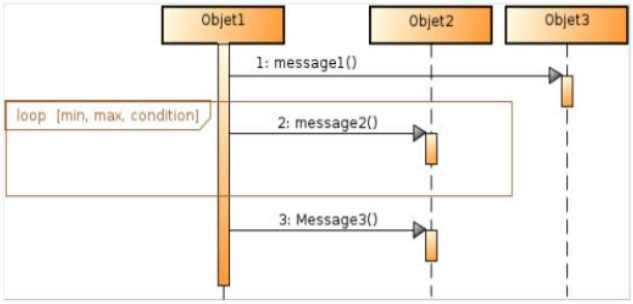
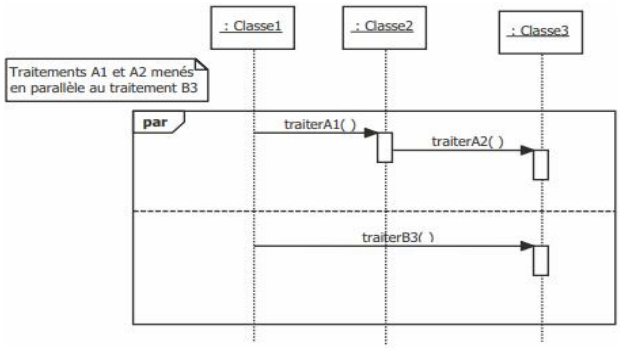
Objectif : représenter les interactions entre objets en indiquant la chronologie des échanges.

Cette représentation peut se réaliser par cas d'utilisation en considérant les différents scénarios associés.

Concepts de base

Concept	Description	Formalisme/Exemple
Ligne de vie	Représente l'ensemble des opérations exécutées par un objet.	
Message synchrone	Dans ce cas l'émetteur reste en attente de la réponse à son message avant de poursuivre ses actions.	
Message asynchrone	Dans ce cas, l'émetteur n'attend pas la réponse à son message, il poursuit l'exécution de ses opérations.	
Opération particulières		
Création d'un objet	La création d'un objet est matérialisée par un message spécifique, appel un constructeur, généralement accompagné du stéréotype *« create »* qui pointe sur le début (le sommet) de la	

	ligne de vie de l'objet créé (Le rectangle de l'instance de la classe est alors surbaissée).	
Destruction d'un objet	La destruction d'un objet est représentée par une croix à la fin de sa ligne de vie. Souvent l'objet est détruit suite à la réception d'un message mais ce n'est pas obligatoire. Dans ce cas-là, il porte le stéréotype *«destroy ».*	
Fragment d'interaction		
Treize opérateurs ont été définis dans UML : alt, opt, loop, par, strict/weak, break, ignore/ consider, critical, negative, assertion et ref.		
Opérateur alt	L'opérateur alt correspond à une instruction de test avec une ou plusieurs alternatives possibles. Il est aussi permis d'utiliser les clauses de type sinon.	
Opérateur opt	L'opérateur opt (optionnel) correspond à une instruction de test sans alternative (sinon).	

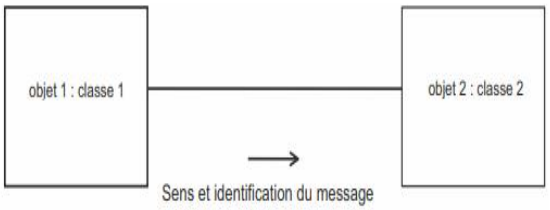
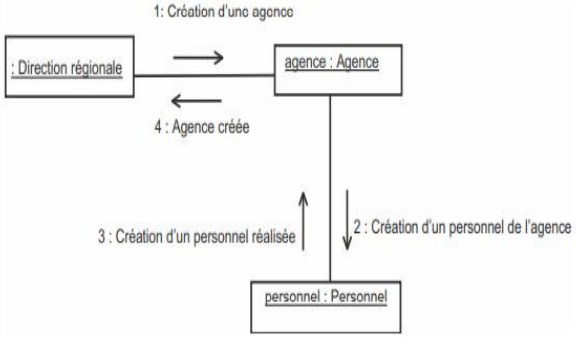
Opérateur loop	L'opérateur loop correspond à une instruction de boucle qui permet d'exécuter une séquence d'interaction tant qu'une condition est satisfaite. Il est possible aussi d'utiliser une condition portant sur un nombre minimum et maximum d'exécution de la boucle en écrivant : loop min, max.	 Source: coursUML5.pdf (remy-manu.no-ip.biz)
Opérateur par	Un fragment d'interaction avec l'opérateur *de traitements parallèles* (par) contient au moins deux sous fragments (*opérandes*) séparés par des pointillés qui s'exécutent simultanément (*traitements concurrents*).	

- Opérateurs ignore et consider : pour les fragments facultatifs ou obligatoires.
- Opérateur critical : Un fragment doit se dérouler sans être interrompu.
- Opérateur break : pour les fragments représentants des scenarii exceptionnels ou de ruptures (ex appui sur la touche *« Esc »*). Le scénario de rupture est exécuté si une condition de garde est satisfaite.
- Opérateur assert : Pour les fragments dont on connaît à l'avance les paramètres du message (exemple : après la saisie des 4 chiffres d'un code, la saisie suivante sera obligatoirement la touche *« Entrée »*).
- Opérateur seq : Le fragment est composé de plusieurs sous fragments qui peuvent s'exécuter dans n'importe quel ordre (mais pas en même temps).
- Opérateur strict : Les messages d'un fragment doivent se dérouler dans un ordre bien précis.
- Opérateur neg : La séquence à l'intérieur du fragment n'est pas valide.
- Opérateur ref : permet d'appeler à un autre diagramme de séquence.

5.1.2. Diagramme de communication

Objectif : permet de représenter des interactions entre objets. Le diagramme de communication met plus l'accent sur l'aspect spatial des échanges que l'aspect temporel.

Concepts de base :

<table>
<tr><th>Concept</th><th>Description</th><th>Formalisme/Exemple</th></tr>
<tr><td>Rôle</td><td>Chaque participant à un échange de message correspondant à une ligne de vie dans le diagramme de séquence se représente sous forme d'un rôle dans le diagramme de communication.
Un rôle est identifié par <nom de rôle>:<nom du type>
Le nom du rôle correspond au nom de l'objet dans le cas où l'acteur ou la classe ont un rôle unique par rapport au système.
Le nom du type correspond au nom de la classe lorsque l'on manipule des objets.</td><td> 
</td></tr>
<tr><td>Massage</td><td>Un message correspond à un appel d'opération effectué par un rôle émetteur vers un rôle récepteur.
Le sens du message est donné par une flèche portée au-dessus du lien reliant les participants au message (origine et destinataire).
Chaque message est identifié par : <numéro> : nom ( )</td><td> 
</td></tr>
<tr><td colspan="3">La syntaxe d'un message :
[n° du message préc. reçu] «.» n° du message [clause d'itération] [condition] «:» nom du message.</td></tr>
<tr><td>Numéro du message précédent reçu</td><td>Permet d'indiquer la chronologie des messages..</td><td rowspan="3">1.2.1 * [3 fois] pour un message à adresser trois fois de suite.
1.2a et 1.2b pour deux messages envoyés en même temps.
Exemple récapitulatif de désignation de message :
1.2a.1.1[si t > 100] : lancer( )
Ce message signifie :
• 1.2a : numéro du message reçu avant l'envoi du message courant.
• 1.1 : numéro de message courant à envoyer.
• [si t > 100] : message à envoyer si t > 100.
• lancer( ) : nom du message à envoyer.</td></tr>
<tr><td>Numéro du message</td><td>Numéro hiérarchique du message de type 1.1, 1.2… avec utilisation de lettre pour indiquer la simultanéité d'envoi de message</td></tr>
<tr><td>Clause d'itération</td><td>Indique si l'envoi du message est répété. La syntaxe est * [spécification</td></tr>
</table>

	de l'itération]	
Condition	Indique si l'envoi du message est soumis à une condition à satisfaire.	

Complément de cours (diagrammes d´interaction)

Q1) Un diagramme de séquence est basé sur une représentation temporelle ?
Oui, un diagramme de séquence illustre les interactions entre les objets du système en montrant, de façon séquentielle, les envois de messages qui interviennent entre ces objets. A chaque objet est associée une ligne de vie qui montre les massages envoyés et reçus par l'objet. Elle montre également les périodes d'activité de l'objet.

Q2) Un diagramme de séquence fait intervenir :

a. Des classes
b. Des objets

Un diagramme de séquence fait intervenir que des objets, instances des classes et non pas les classes.

Q3) Un message peut contenir des paramètres ?

Oui un message permet de transmettre des informations entre les objets. C'est le but des paramètres.

Q4) Comment est représenté le message de création d'un objet ?

La création d'un objet est représentée par un message spécifique qui conduit à la tête de la ligne de vie du nouvel objet.

Q5) Quelle est la différence entre un message synchrone, asynchrone et de retour ?

Lors de l'envoi d'un message synchrone, l'émetteur du message attend le retour du destinataire avant de continuer sa propre activité, c'est-à-dire qu'il attend le message de retour émis par le destinataire. Dans le cas d'un message asynchrone, cette attente n'existe pas. Il n'y a alors pas message de retour.

Q6) Un objet peut-il envoyer un message à lui-même ?

Oui, un objet peut envoyer un message à lui-même.

Q7) A quoi servent les cadres d'interaction ?

Les cadres d'interaction permettent de décrire des alternatives, des boucles, des messages à envoyer en parallèle, etc.

Q8) Une condition de garde renvoie une valeur booléenne?

Oui, une condition de garde est une expression booléenne qui renvoie une valeur vraie ou fausse.

Q9) Un diagramme de communication utilise également une représentation temporelle?
Non, un diagramme de communication est basé sur une représentation spatiale des objets. Chaque objet est lié graphiquement aux objets avec lesquels il interagit.

Exercices

Exercice N°1 (avec corrigé) :

Le but de l'exercice est de décrire les interactions entre un utilisateur et le site Web d'une société de chemins de fer.

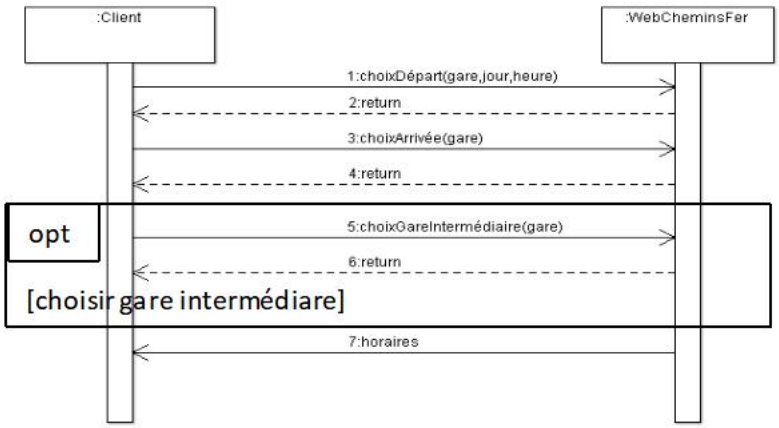
a) Représentez à l'aide d'un diagramme de séquence, les interactions suivantes :

- Le choix de la gare, de la date et de l'heure du départ,

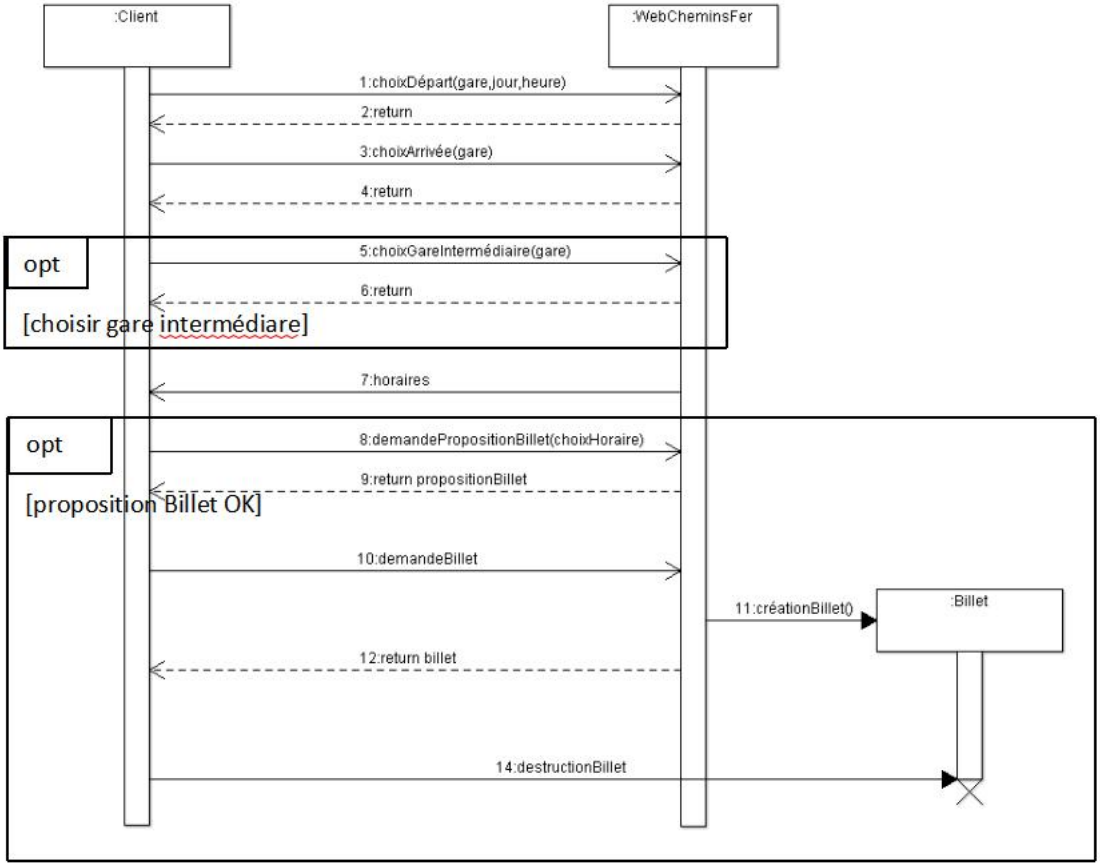
- Le choix de la gare d'arrivée,

- Choix optionnel d'une gare intermédiaire,

- Obtention des horaires.

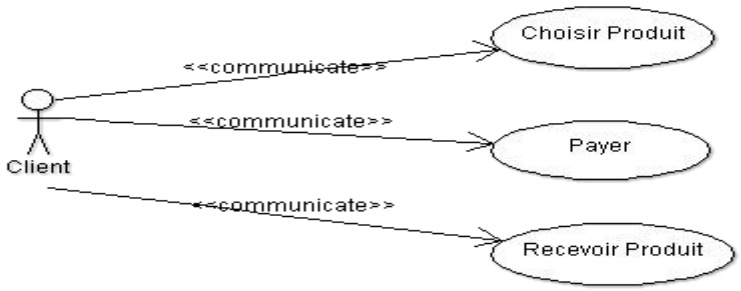


b) Ajoutez la possibilité d'acheter un billet correspondant à l'horaire préalablement sélectionné parmi ceux visualisés. L'utilisateur confirme ensuite l'acquisition définitive avant de recevoir le billet pour impression. Aucune trace informatique du billet n'est conservée.



Exercice N°2 :

Le but de l'exercice est de montrer comment le diagramme de séquence constitue un support pour décrire les objets composant un système. Le système considéré est un distributeur automatique de produits alimentaires. L'utilisateur communique avec le système au travers de trois cas d'utilisation : Choisir Produit, Payer et Recevoir Produit. Le diagramme de cas d'utilisation ci-après montre ces trois cas :



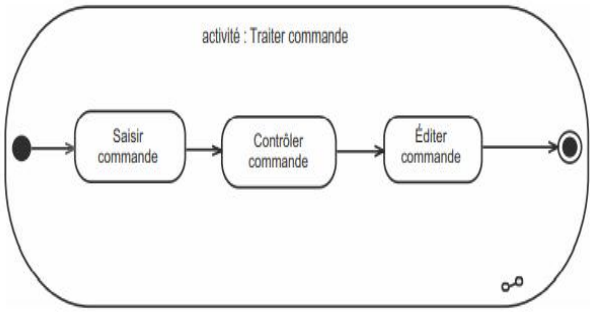
- Elaborez un diagramme de séquences qui illustre les cas d'utilisation, en respectant la séquence globale basée sur l'achat d'un produit : choix, paiement puis réception du produit.

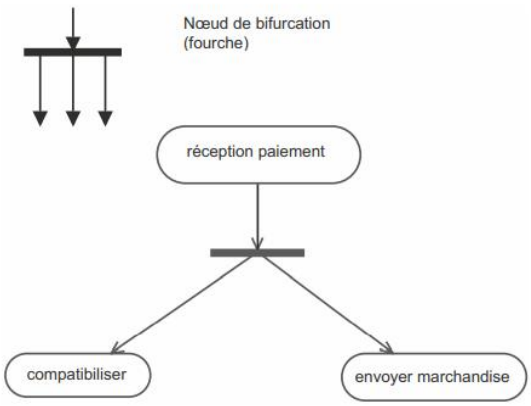
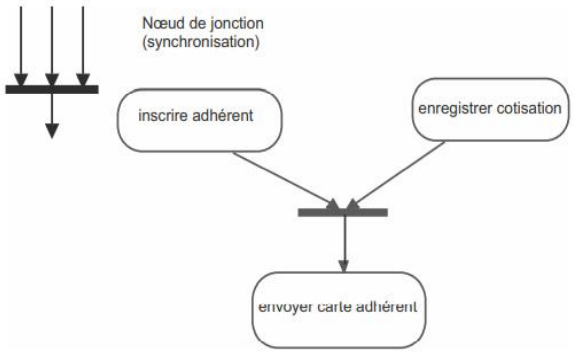
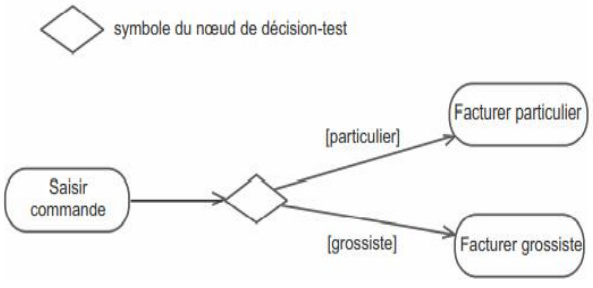
- Décomposez les messages de chaque diagramme de séquences pour découvrir des objets du système. Par exemple, le choix du produit s'effectue en tapant sur un clavier.

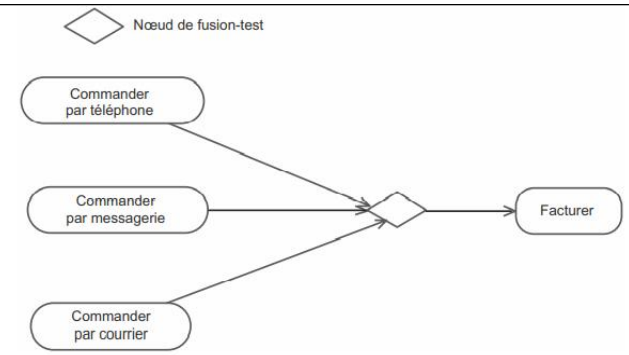
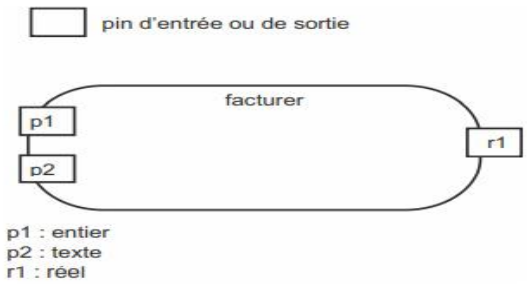
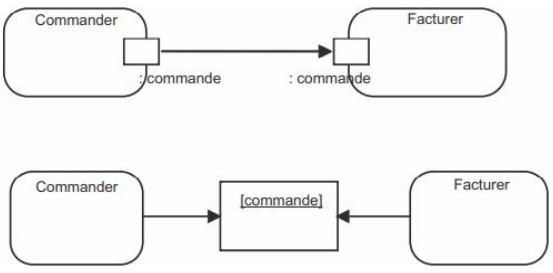
5.2. Diagramme d'activité

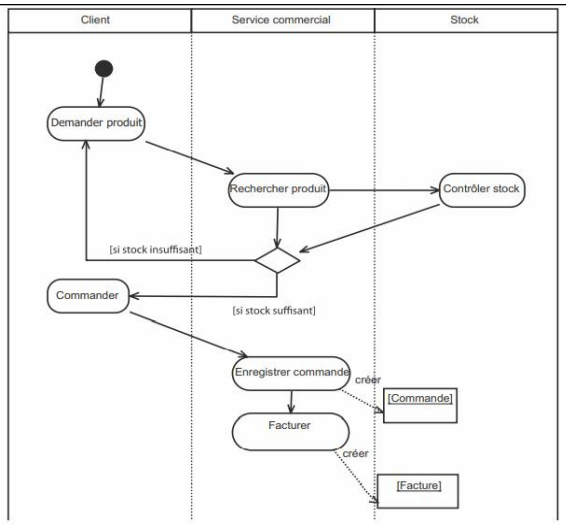
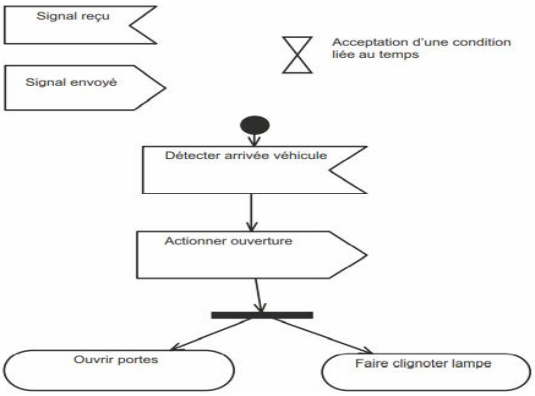
5.2.1. Objectif : il présente le comportement interne des opérations ou des cas d'utilisation. Dans ce diagramme le comportement s'applique aux flots de contrôle et aux flots de données propres à un ensemble d'activité.

5.2.2. Concepts de base

Concept	Description	Formalisme / Exemple
Une activité	Une activité représente le comportement d'une partie du système en termes d'actions et de transitions. Une activité est composée de trois types de nœuds : Nœud d'exécution (action, transition), Nœud de contrôle et Nœud d'objet.	
Les nœuds d'exécution		
Action	Une action correspond à un traitement qui modifie l'état du système.	
Transition et flot de contrôle	Dès qu'une action est achevée, une transition automatique est déclenchée vers l'action suivante.	

	Il n'y a donc pas d'événement associé à la transition. L'enchaînement des actions constitue le flot de contrôle.	
Les nœuds de contrôle		
Un nœud de bifurcation (fourche)	Un nœud de bifurcation (fourche) permet à partir d'un flot unique entrant de créer plusieurs flots concurrents en sortie de la barre de synchronisation.	
Un nœud de jonction (synchronisation)	Un nœud de jonction (synchronisation) permet, à partir de plusieurs flots concurrents en entrée de la synchronisation, de produire un flot unique sortant. Le nœud de jonction est le symétrique du nœud de bifurcation.	
Un nœud de test-décision	Un nœud de test-décision permet de faire un choix entre plusieurs flots sortants en fonction des conditions de garde de chaque flot.	

Nœud de fusion-test	Un nœud de fusion-test permet d'avoir plusieurs flots entrants possibles et un seul flot sortant. Le flot sortant est donc exécuté dès qu'un des flots entrants est activé.	
Pin d'entrée et de sortie	Un pin d'entrée ou de sortie représente un paramètre que l'on peut spécifier en entrée ou en sortie d'une action. Un nom de donnée et un type de donnée peuvent être associés au pin. Un paramètre peut être de type objet.	
Les nœuds d'objet		
Flot de données et nœud d'objet	Un nœud d'objet permet de représenter le flot de données véhiculé entre les actions. Les objets peuvent se représenter de deux manières différentes : soit en utilisant le pin d'objet soit en représentant explicitement un objet.	

Partition	UML permet aussi d'organiser la présentation du diagramme d'activité en couloir d'activités. Chaque couloir correspond à un domaine de responsabilité d'un certain nombre d'actions.	
Représentation d'actions de communication		
- Signal - Ecoulement du temps.	Les interactions de communication liées à certains types d'événement peuvent se représenter.	

Complément de cours (diagramme d´activités)

Q1) Un diagramme d'activité peut être utilisé pour décrire un workflow ?

Oui, un diagramme d'activité est un bon candidat pour décrire les activités et les enchaînements d'activités constituant un workflow.

Q2) Un diagramme d'activité peut servir à découvrir des activités parallèles ?

Oui, il est possible que les activités soient exécutées simultanément.

Q3) Une activité représente une tâche automatisée ou manuelle ?

Oui, une activité est constituée d'actions élémentaires. Une action consiste à affecter une valeur à un attribut, créer ou détruire un objet, effectuer une opération, envoyer un signal à un autre objet ou à soi-même, etc. dans le cas d'un workflow, il peut aussi s'agir d'une tâche manuelle.

Q4) Une activité pet être implantée par une méthode d'un objet?
Oui, une méthode peut tout à fait décrire le comportement d'une activité.

Q5) Les éléments suivants peuvent apparaître dans un diagramme d'activités : activité, enchaînement, garde, alternative, fourche, synchronisation, activité initiale et finale?
Oui, activité, enchaînement, garde alternative, fourche, synchronisation, activité initiale et finale sont les principaux éléments constituant un diagramme d'activités.

Q6) Un diagramme d'activités peut décrire des activités parallèles en fixant des priorités.?
Non, un diagramme d'activité peut décrire des activités enclenchées en parallèle mais sans fixer ni ordre ni priorité.

Q7) Un enchaînement peut être lié à un événement?

Non, un enchaînement peut simplement être lié à une condition de garde mais pas à un événement comme dans le cas d'une transition du diagramme d'états-transitions.

Q8) A quoi sert un enchaînement de type fourche ainsi qu'un enchaînement de type synchronisation ?

Un enchaînement d'activités de type fourche possède plusieurs activités de destination. Après son franchissement, toutes les activités de destination sont enclenchées en parallèle. Un enchaînement d'activités de type synchronisation possède plusieurs activités d'origine. L'exécution de toutes les activités d'origine doit être terminée pour que l'enchaînement soit franchi. Un tel enchaînement permet de synchroniser plusieurs activités enclenchées en parallèle.

Q9) L'objet associé à une travée (ou un couloir) est responsable des activités contenues dans cette travée.?
Oui, une travée contient toutes les activités du diagramme dont l'objet associé est responsable.

Q10) A quoi sert une activité complexe ?

Une activité complexe est composée d'un ensemble de sous-activités.

Exercices Corrigés

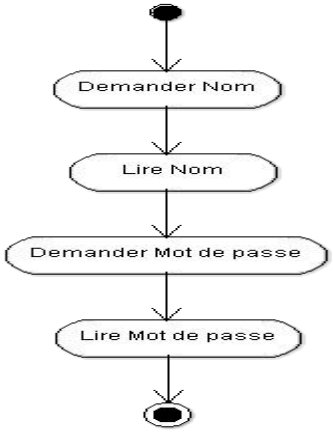
Exercice N°1 :

L'authentification d'un utilisateur à un système informatique consiste en deux étapes :

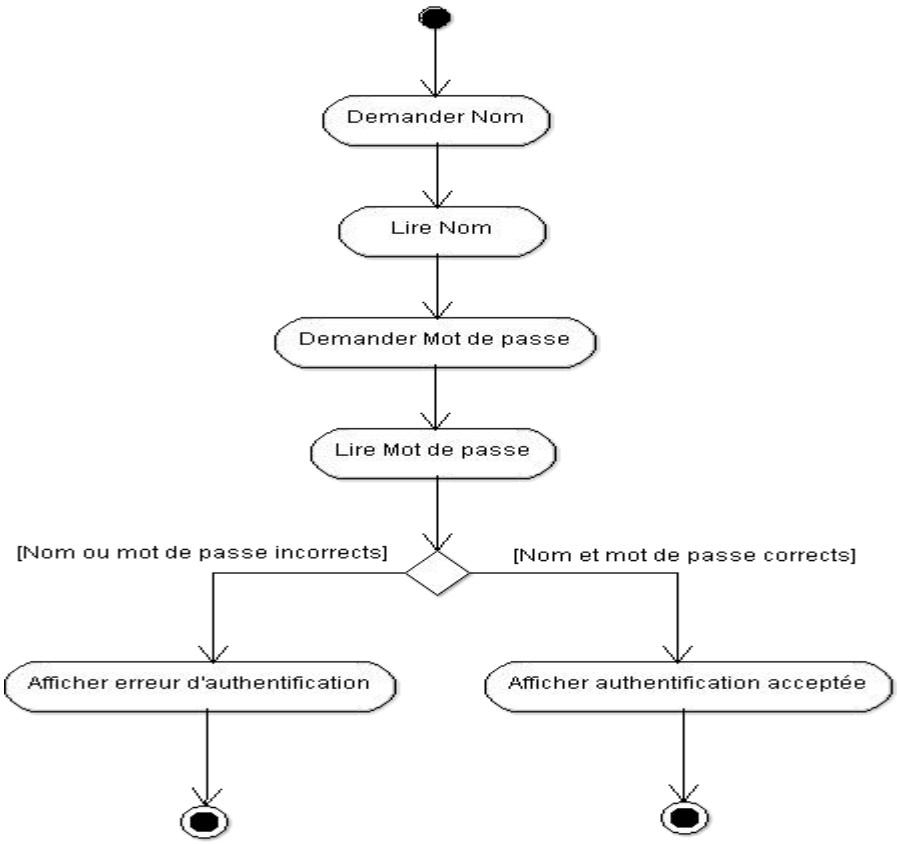
- La saisie du nom
- La saisie du mot de passe

1) Représentez en UML les activités nécessaires que doit réaliser le système d'information pour authentifier un utilisateur en faisant l'hypothèse que

l'utilisateur saisisse correctement son nom et son mot de passe. On suppose que ces deux étapes sont réalisées chronologiquement.



2) Ajoutez une alternative pour tester si le nom et le mot de passe sont corrects.



Exercice N°2 :

Un virement international dans l'Union Européenne requiert les indications suivantes :

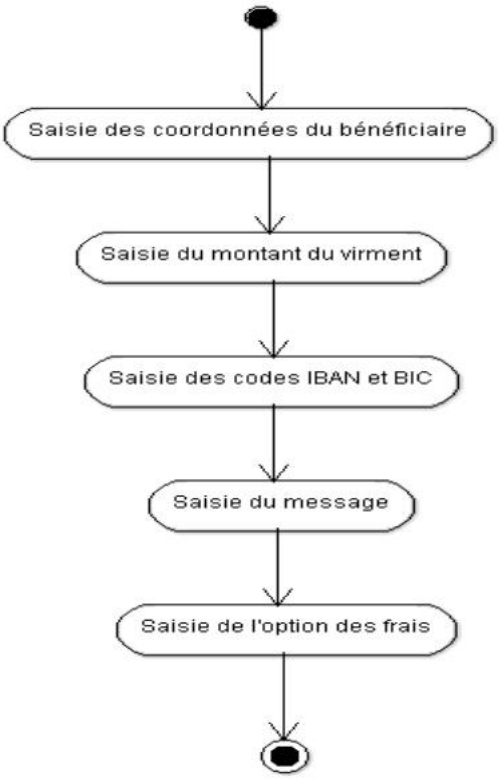
- Nom et adresse du bénéficiaire,
- Montant du virement,
- Code BIC de a banque qui est un identifiant unique de 8 caractères,
- Numéro du compte bancaire au format IBAN qui est une représentation unifié,
- Message à l'attention du bénéficiaire,

Par ailleurs, il existe trois options distinctes laissées au choix du donneur d'ordre du virement :

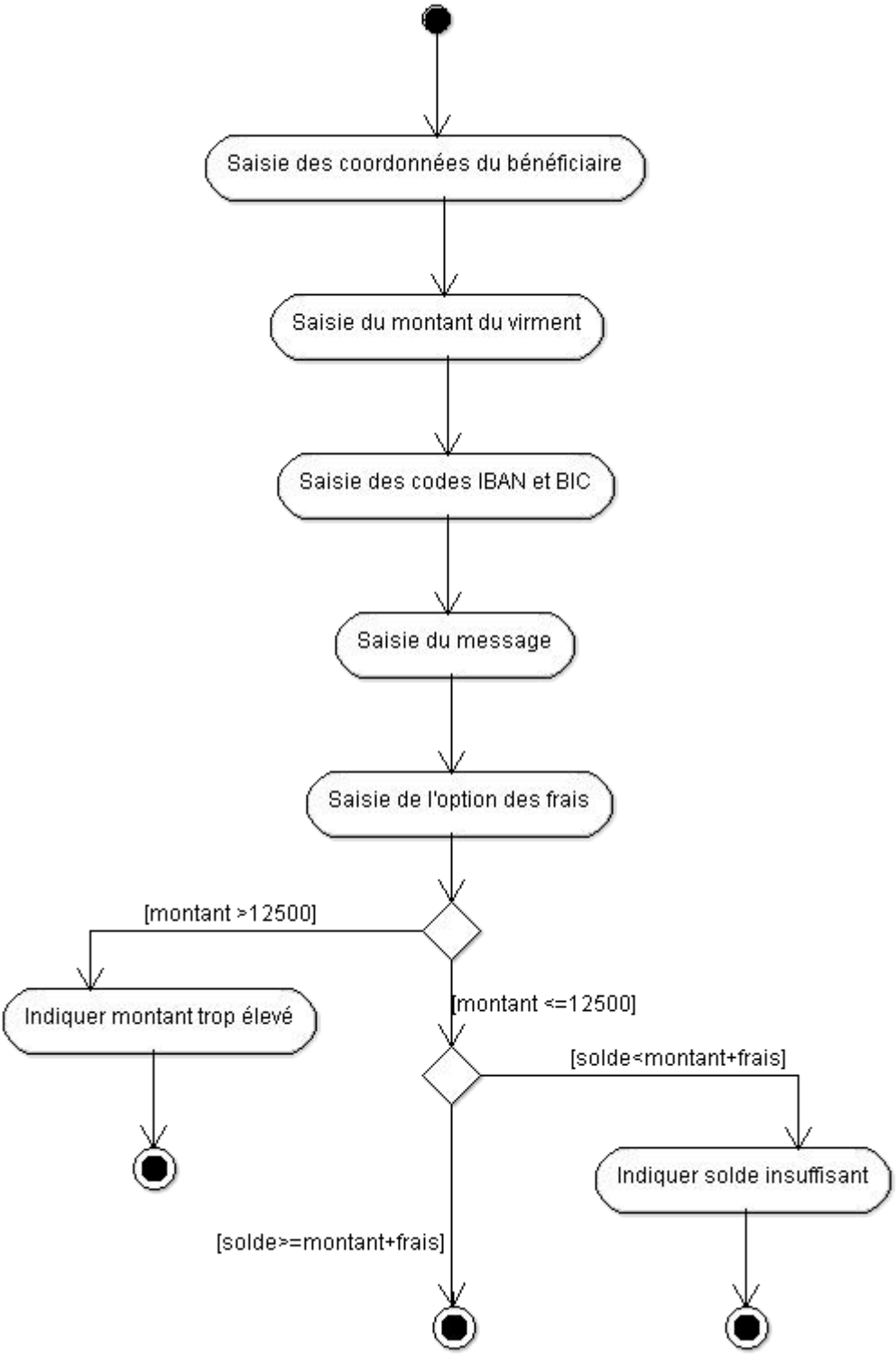
- L'option partagée où le donneur d'ordre et le destinataire paient respectivement les frais de leur banque respective,
- L'option frais à charge du donneur d'ordre où le donneur d'ordre paie l'ensemble des frais,
- L'option frais à charge du bénéficiaire où le bénéficiaire paie l'ensemble des frais.

Le système pris en compte est le site Web de la banque qui offre la possibilité à ses utilisateurs de saisir un virement européen en ligne.

1) Représentez en UML les différentes saisies que doit réaliser le système auprès de l'utilisateur pour préparer un virement européen.

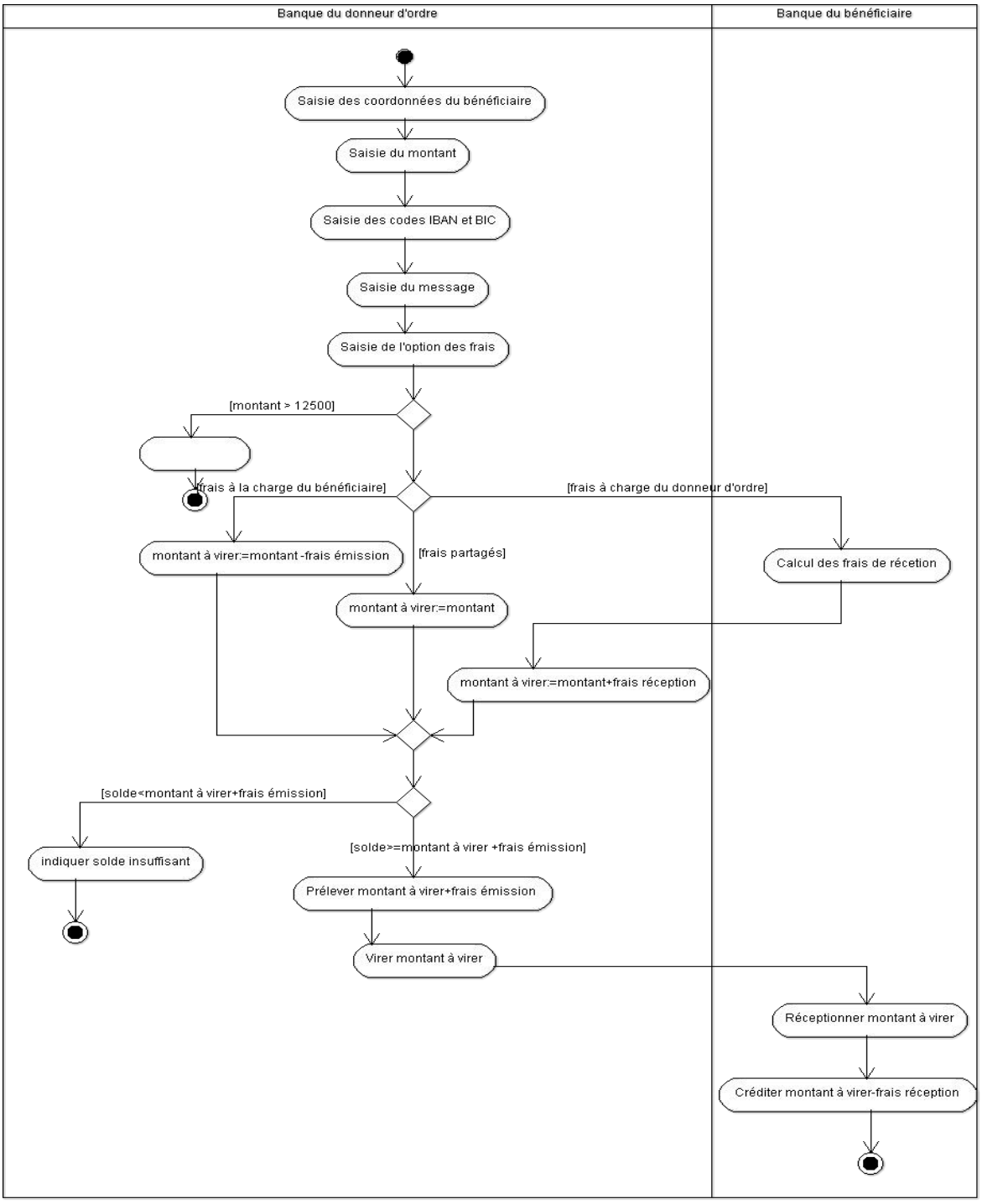


2) Le montant d'un virement européen ne peut pas dépasser 12 500 Euros, sinon il convient d'utiliser une autre procédure dite du virement international qui demande d'autres informations et dont le coût est plus élevé. Modifiez le diagramme pour que le système prévienne l'utilisateur si le montant dépasse 12 500 Euros. Prenez également en compte le cas où le total du montant et des frais dépasse le solde de son compte.



3) Complétez le diagramme précédent en incluant :
    a. Le calcul des frais (en interrogeant la banque du bénéficiaire si nécessaire).

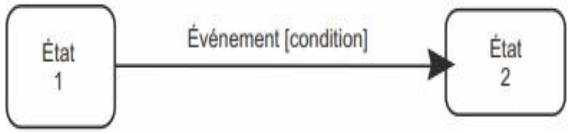
b. Le virement en lui-même.
c. La mise à jour du solde du compte du donneur d'ordre et du bénéficiaire.



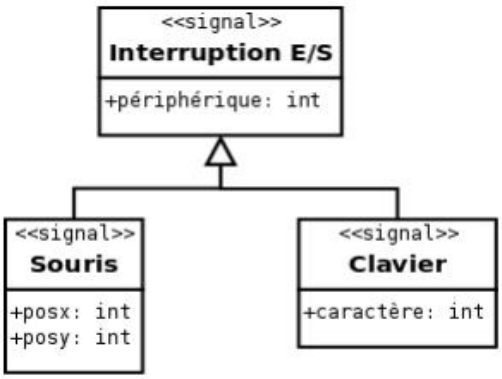
5.3. Diagramme état/transition

5.3.1. Objectif : Permet de décrire le fonctionnement d'une machine (ou d'un objet) ayant un comportement séquentiel. Il représente les différents états (situations) dans lesquels peut se trouver la machine (ou l'objet) ainsi que la façon dont cette machine (ou l'objet) passe d'un état à l'autre en réponse à des événements.

5.3.2. Concepts de base

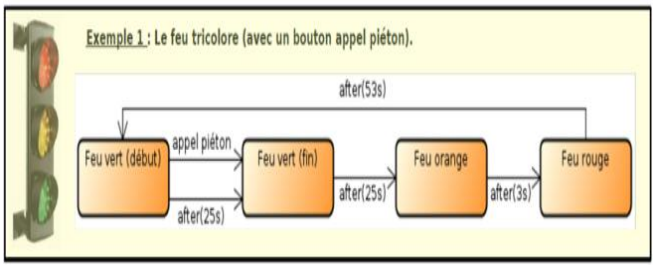
<table>
<tr><th>Concept</th><th>Description</th><th>Formalisme / Exemple</th></tr>
<tr><td>Etat</td><td>L'état d'un objet est défini, à un instant donné, par l'ensemble des valeurs de ses propriétés.
Un état représente une situation durant la vie d'un objet pendant laquelle : il satisfait une certaine condition, il exécute une certaine activité ou bien il attend un certain événement.</td><td rowspan="3"> 
</td></tr>
<tr><td>Transition</td><td>Le passage d'un état à un autre état s'appelle transition.</td></tr>
<tr><td>Événement</td><td>Un événement est un fait survenu qui déclenche une transition.</td></tr>
<tr><td colspan="3">Etat initial et état final</td></tr>
<tr><td>L'état initial</td><td>L'état initial est un pseudo état qui indique 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 initia.</td><td>  </td></tr>
<tr><td>L'état final</td><td>L'état final est un pseudo état qui indique que le diagramme d'états-transitions, ou l'état enveloppant, est terminé.</td><td>  </td></tr>
<tr><td colspan="3">Il existe quatre types d'événements :</td></tr>
<tr><td>Type appel de méthode (call)</td><td colspan="2">Appel d'une méthode de l'objet courant par un autre objet ou par un acteur.

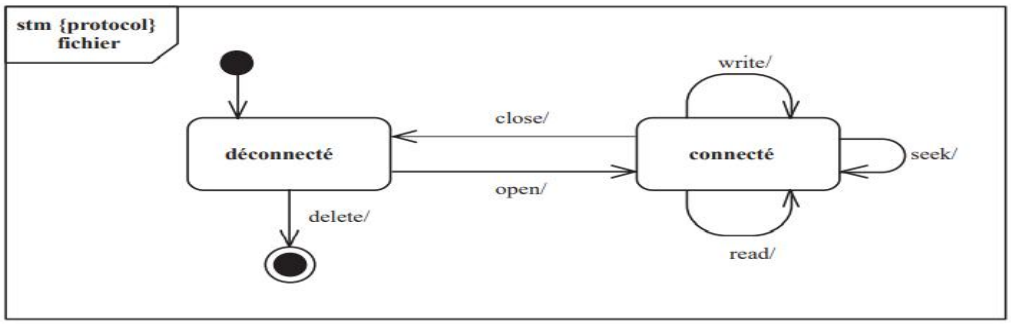
La méthode est déclarée dans le diagramme de classe, mais à l'intérieur de la classe</td></tr>
</table>

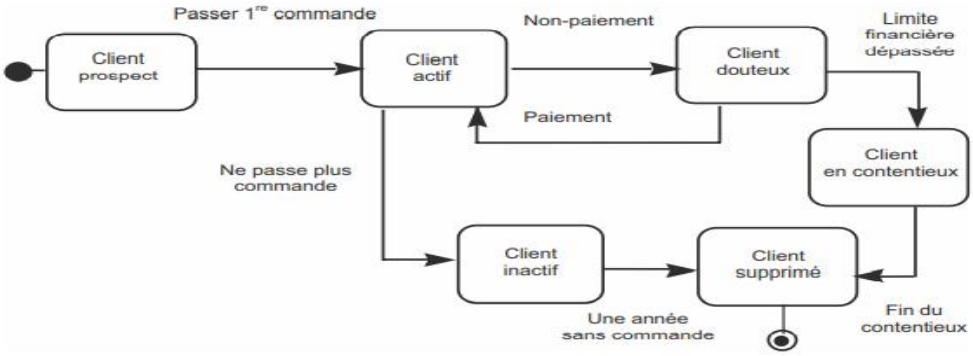
<table>
<tr><td></td><td colspan="2"></td></tr>
<tr><td>Type signal</td><td>Exemple : clic de souris, interruption d'entrées-sorties…
Les signaux sont déclarés dans le diagramme de classes de la même manière qu'une classe, en ajoutant le stéréotype <<signal>> au dessus du nom.</td><td> 
</td></tr>
<tr><td>Type changement de valeur (vrai/faux)</td><td colspan="2">C'est le cas de l'évaluation d'une expression booléenne. C-á-d un changement dans la satisfaction (vrai ou faux) d'une condition booléenne sur des valeurs d'attributs.

On utilise alors le mot-clé when, suivi d'une expression booléenne.

Syntaxe: when(expression booléenne)

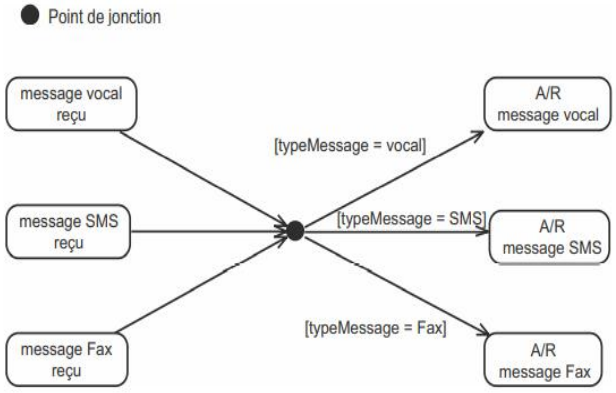
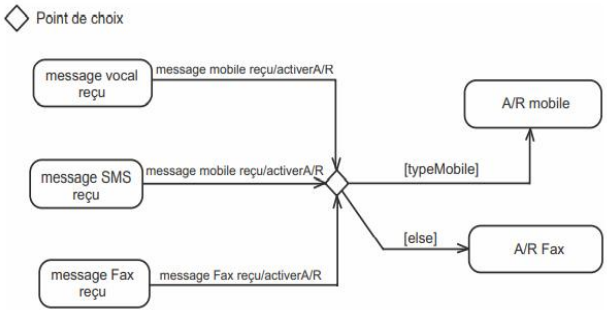
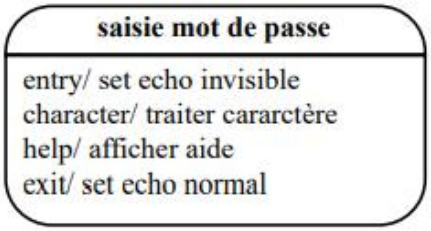
L'événement de changement se produit lorsque la condition passe à vrai.</td></tr>
<tr><td>Type écoulement du temps</td><td>C'est un événement lié à une condition de type after (durée) ou when (date)
Exemple : when(date=11/12/2017)</td><td> 

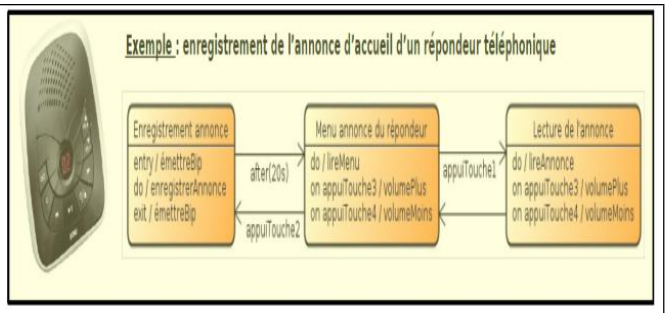
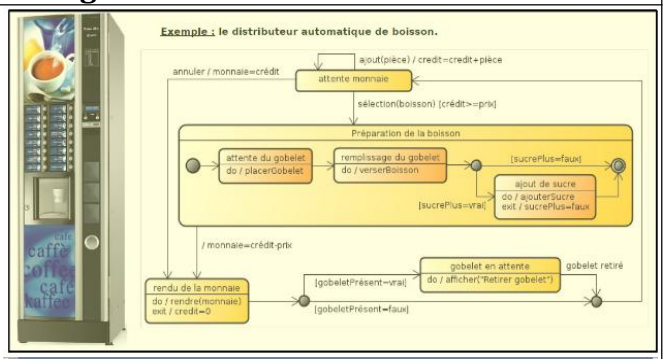
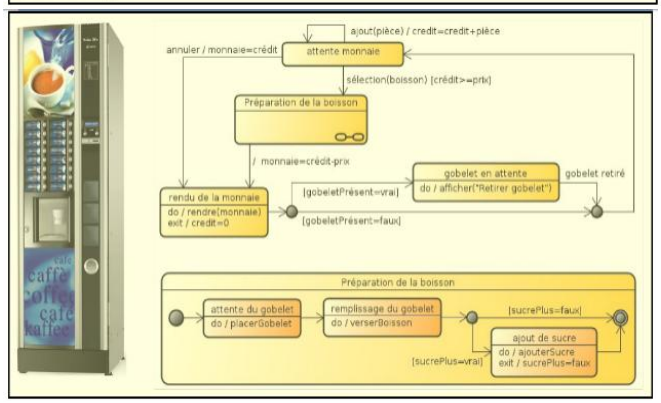
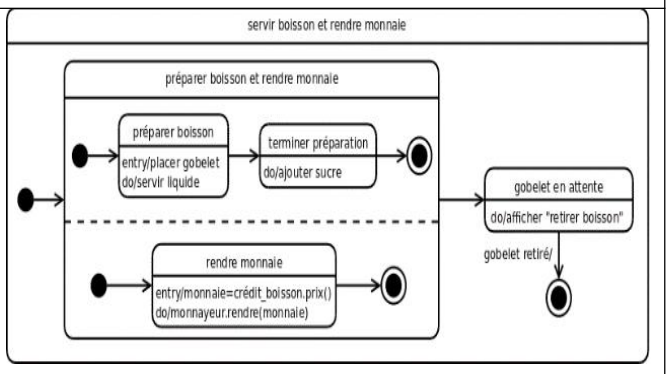
Source: coursUML7.pdf (remy-manu.no-ip.biz)</td></tr>
<tr><td colspan="3">Exemples</td></tr>
<tr><td>Exemple 01: diagramme état/transition d'un objet fichier</td><td colspan="2"> 
</td></tr>
</table>

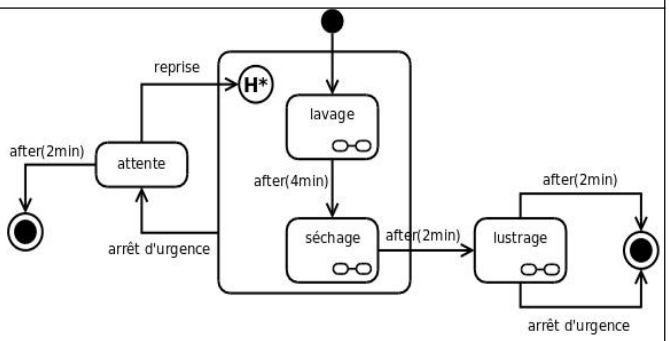
Exemple 02 : Diagramme état/transition d'un objet client	

Points de décision

Il est possible de représenter des alternatives pour le franchissement d'une transition. On utilise pour cela des pseudo-états particuliers :

Les points de jonction	Lorsque l'on veut relier plusieurs états vers d'autres états, un point de jonction permet de décomposer une transition en deux parties en indiquant si nécessaire les gardes propres à chaque segment de la transition. À l'exécution, un seul parcours sera emprunté, c'est celui pour lequel toutes les conditions de garde seront satisfaites.	
Les points de choix	Le point de choix se comporte comme un test de type : si condition faire action1 sinon faire action2.	
Etat et transition interne	La syntaxe d'une transition interne est la même que celle d'une transition externe : *Déclencheur [garde] / effet* UML définit des mots clé correspondant à des événements particuliers : *entry* : définit l'action à exécuter lors de l'entrée dans l'état. *exit* : définit l'action à	

	exécuter lors de la sortie de l'état. *do* : définit l'activité à exécuter dès que celle définie par *entry* est terminée. *Event (on)* : (optionnel) définit l'activité à exécuter à chaque fois que nous avons un événement particulier. *Les transitions internes s'écrivent à l'intérieur de l'état, séparé du nom de l'évènement par un trait.*	 Source: coursUML7.pdf (remy-manu.no-ip.biz)
Emboitement de diagrammes		
Etat composite	Un *état composite* est un état qui contient d'autres états, appelés sous-états ou états imbriqués. Ces derniers peuvent être : *séquentiels* (ou encore disjoints), ou *concurrents* (aussi appelés parallèles). Les sous-états sont à leur tour susceptibles d'être décomposés. Afin d'éviter de surcharger le diagramme d'états-transition, il est possible de placer le symbole :  À l'intérieur de l'état composite et de spécifier ensuite son contenu ailleurs dans un autre diagramme d'état.	  Source : coursUML7.pdf (remy-manu.no-ip.biz)
Etats concurrents	Les diagrammes d'états-transitions permettent de décrire efficacement les mécanismes concurrents grâce à l'utilisation d'*états orthogonaux*. Un état orthogonal est un état composite comportant plus d'une région, chaque région représentant un flot	

	d'exécution.	
Etat historique	Un état historique, également qualifié d'*état historique plat*, est un pseudo-état qui mémorise le dernier sous-état actif d'un état composite. Graphiquement, il est représenté par un cercle contenant un *H*. Il est également possible de définir un *état historique profond* représenté graphiquement par un cercle contenant un *H**. Cet état historique profond permet d'atteindre le dernier état visité dans la région, quel que soit son niveau d'imbrication, alors que le l'état historique plat limite l'accès aux états de son niveau d'imbrication.	

Complément de cours (Diagramme état/transition)

Q1) Un diagramme d'état transition est utilisé pour décrire le cycle de vie d'un objet.?
Oui, un diagramme d'états-transitions montre l'ensemble des états que peut prendre un objet au cours de son cycle de vie.

Q2) Il peut exister plusieurs états initiaux?

Non, il existe un et un seul état initial.

Q3) Une transition peut lier plus de deux états ?

Non, une transition lie deux états, à savoir son état d'origine et son état de destination. Une transition réflexive possède le même état d'origine et de destination.

Q4) Qu'est-ce qui peut être associé à une transition ?

Il est possible d'associer plusieurs éléments à une transition :

a. ***Un événement : il faut que cet événement soit reçu par l'objet pour que la transition soit franchie ;***
b. ***Une condition de garde : il faut que cette condition soit vraie pour que la transition soit franchie ;***
c. ***Une ou plusieurs activités : ces activités sont exécutées lorsque la transition est franchie.***

Q5) Dans la liste suivante, quelles sont les activités qui peuvent être utilisées dans un diagramme d'états-transitions ?

d. Envoyer un signal à soi-même,
e. Appeler une méthode d'un autre objet,
f. Changer d'état,
g. Affecter une valeur à un attribut de l'objet,
h. Appeler une des méthodes de l'objet,
i. Envoyer un signal à un autre objet,
j. Annuler le franchissement de la transition, dans le cas d'une activité intervenant lors du franchissement d'une transition.

Les activités a, b, d, e, f sont tout à fait possibles.

Q6) Si une activité est associée à un état, celle-ci peut être interrompue par un événement?

Non, une activité associée à un état ne peut donc pas être interrompue par un événement.

Q7) A quoi sert un état composé ?

Il sert à simplifier la représentation. Il permet également de gérer des sous-états parallèles.

Q8) Dans un état composé, à quoi sert le sous-état de mémoire, noté par un H entouré d'un cercle ?

Le sous-état de mémoire H permet de revenir au sous-état de l'état composé qui était actif lorsque l'objet a quitté cet état composé.

Q9) Un objet peut se trouver simultanément dans plusieurs sous-états ?

Oui, il est possible qu'un objet se trouve simultanément dans des sous-états parallèles.

Q10) Que signifie le mot clé entry dans un état ?

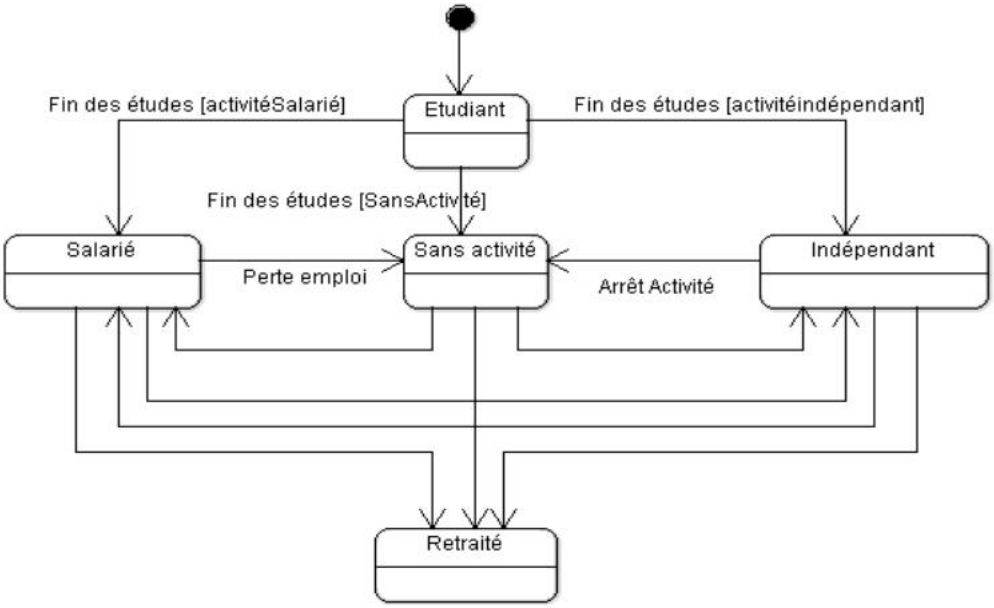
Le mot clé entry sert à décrire les activités qui sont déclenchées lorsque l'objet entre dans l'état.

Exercices corrigés

Exercice N°1 :

L'objectif de cet exercice est de décrire les différentes états que peut traverser la situation professionnelle d'une personne, ainsi que les transitions qui y sont associées. Les états professionnels possibles pour cette personne sont étudiante, salariée, sans activité, indépendante ou retraitée. Il est à noter que les activités simultanées, telles que le fait d'être salarié et indépendant en même temps, ne sont pas prises en compte. Le point de départ de cette situation professionnelle est l'état d'étudiante.

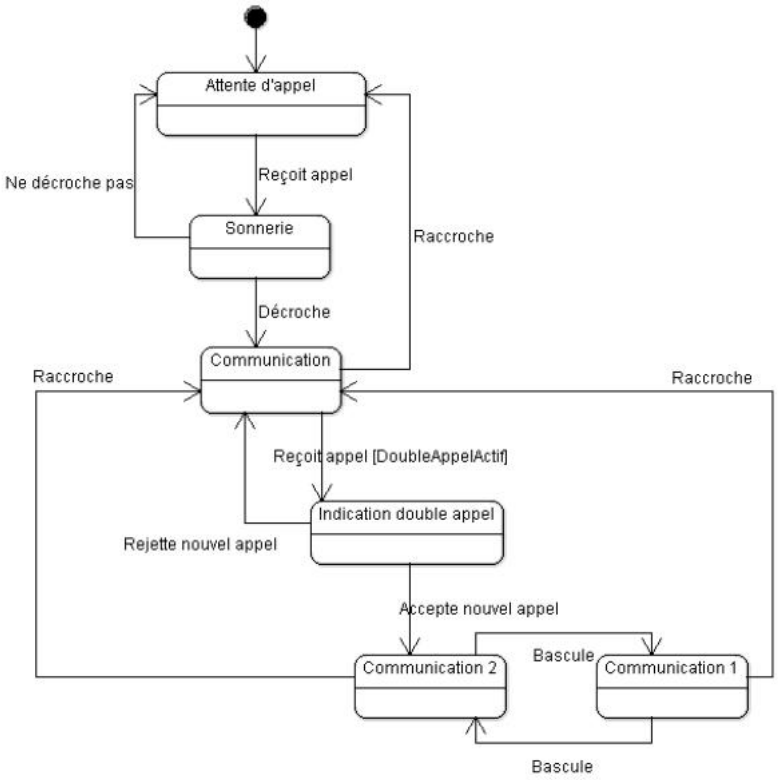
- Élaborez le diagramme d'états-transitions correspondant (utilisez les conditions de garde pour différencier les possibilités multiples).



Exercice N°2 :

L'objectif de cet exercice est d'explorer les différents états que peut traverser un téléphone portable lors de la réception d'appels, y compris la possibilité de gérer deux conversations simultanément en utilisant le principe du double appel.

- Élaborez un diagramme d'états-transitions représentant les différents états que peut prendre le téléphone portable (le téléphone est bien sûr allumé).


Attente d'appel
Reçoit appel
Ne décroche pas
Sonnerie
Raccroche
Décroche
Communication
Raccroche
Raccroche
Reçoit appel [DoubleAppelActif]
Indication double appel
Rejette nouvel appel
Accepte nouvel appel
Bascule
Communication 2
Communication 1
Bascule

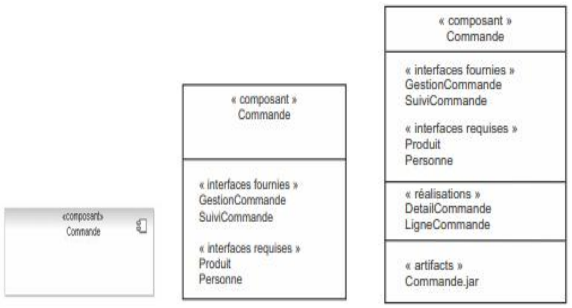
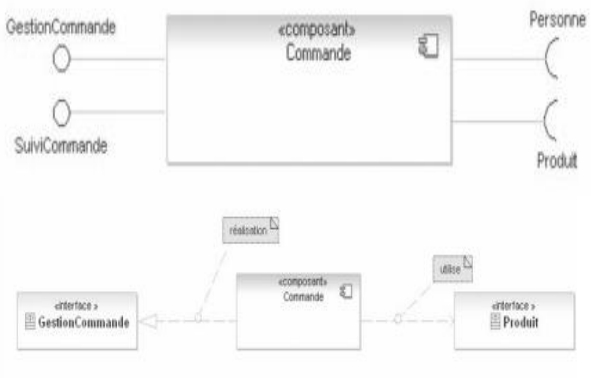
6.Chapitre 6/ Autres notions et diagrammes UML

6.1. Composants, déploiement, structures composites.

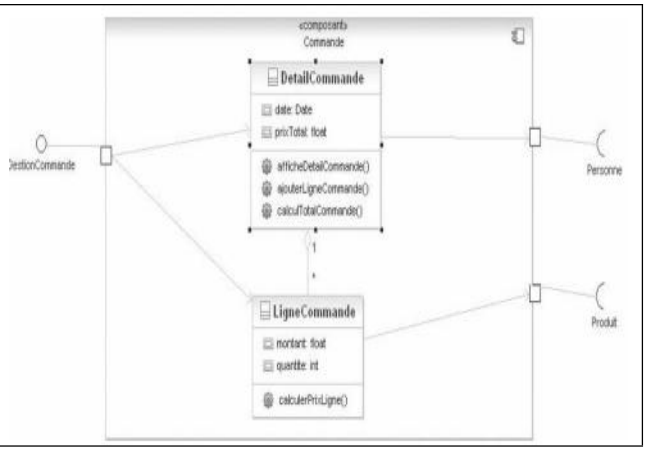
6.1.1. Diagramme de composants

Objectif : il permet de représenter les différents constituants ainsi que les relations entre ces constituants du logiciel au niveau de l'implémentation d'un système[15] .

Concepts de base :

Concept	Description	Formalisme/Exemple
Composant	Il est assimilé à un élément exécutable du système. Un composant est caractérisé par son nom, sa spécification externe et son port de connexion	
Spécification externe	Elle peut être soit sous forme : - d'une ou plusieurs interfaces requises - d'une ou plusieurs interfaces fournies	

[15] UML 2 analyse et conception, Mise en œuvre guidée avec étude de cas, Joseph Gabay, David Gabay, DUNOD, 2008

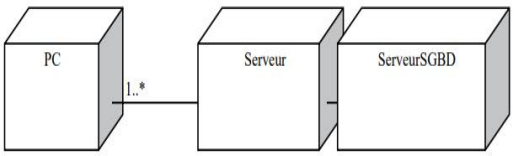
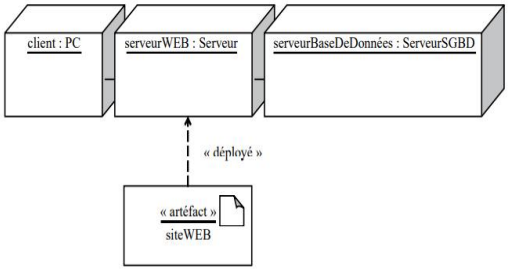
Port de connexion	Il représente le point de connexion entre le composant et une interface.	

6.1.2. Diagramme de déploiement

Le langage UML n'est pas limité à l'établissement de modèles (en phases d'analyse et de conception). Le déploiement est une étape importante du développement d'un système.

Objectif : le diagramme de déploiement permet de représenter un environnement d'exécution ainsi que des ressources physiques.

Concepts de base :

Concept	Description	Formalisme/Exemple
Nœud	C'est l'environnement d'exécution	
Artefact	C'est une partie d'un système qui s'exécute sur un nœud Exemples : un fichier binaire exécutable, un fichier source, une table d'une base de données, un document, un courrier électronique, etc.	

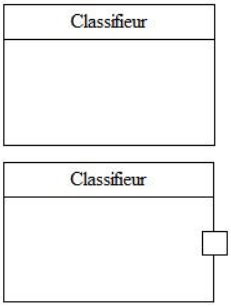
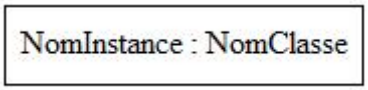
6.1.3. Diagramme de structure composite

Le diagramme de structures composites décrit la structure interne d'une classe, ainsi que les collaborations que cette dernière rend possible. Les éléments de ce diagramme sont les parties (parts), les ports par le biais desquels les parties interagissent entre elles, avec différentes instances de la classe ou encore avec le monde extérieur, et enfin les connecteurs reliant les parties et les ports.

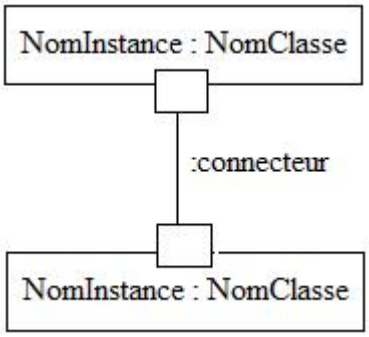
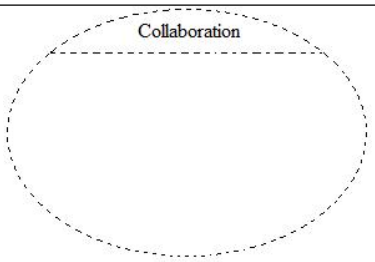
Une structure composite est un ensemble d'éléments interconnectés et collaborant dans un but commun lors de l'exécution d'un tâche.

Objectif : il permet de représenter des collaborations d'instances constituant des fonctions particulières du système à développer.

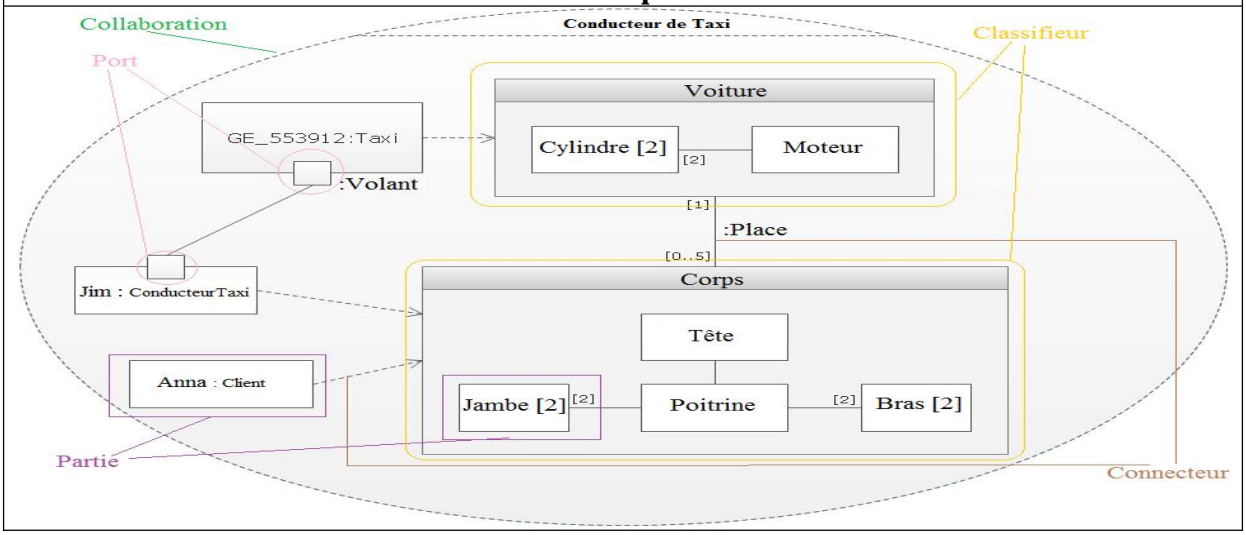
Concepts de base [16]:

Concept	Description	Formalisme/Exemple
Les classifieurs structurés	Il incarne généralement une classe abstraite dont le comportement peut être décrit en tout ou en partie par des interactions entre ses composants. Un classifieur encapsulé est un type de classifieur structuré qui est équipé de ports.	
Les parties	Une partie est définie comme un rôle joué par une ou plusieurs instances d'une classe lors de l'exécution. Elle peut porter le nom d'un rôle, d'une super-classe abstraite ou encore d'une classe concrète spécifique. En outre, une partie peut comporter une cardinalité spécifique.	
Les ports	Un port représente un point d'interaction qui permet de connecter un classifieur structuré à ses parties ou à son environnement. En outre, les ports peuvent éventuellement spécifier les services qu'ils fournissent ou requièrent d'autres parties du système. Les ports sont généralement représentés sur les diagrammes par un carré.	

[16]Les définitions, formalismes et exmples de cette partie sont extraits de: le cours pratique: 6- le diagramme de structures composites

Les connecteurs	Les connecteurs sont des éléments qui permettent de relier plusieurs entités entre elles afin de faciliter leur interaction pendant l'exécution. Un connecteur peut être symbolisé par une ligne reliant différentes combinaisons de parties, de ports ou de classifieurs structurés.	
Les collaborations	En règle générale, une collaboration est d'un niveau d'abstraction plus élevé qu'un classifieur. Elle est représentée par un ovale en pointillé qui contient les rôles attribués à chaque instance lors de son exécution.	

Exemple



6.2. Mécanismes d'extension : langage OCL et les profils

6.2.1. Introduction

OCL (Object Constraint Language) est un langage à objets déclaratif. Il permet de spécifier des contraintes dans les diagrammes d'UML[17] :

- Typer les différentes variables utilisées et notamment les attributs et opérations.

- Définir les assertions (invariants de classes, pré- et post- conditions d'opérations)

[17] GÉNIE LOGICIEL - Développement de logiciels avec UML 2 et OCL - Cours, études de cas et exercices corrigés (Niveau B) (Technosup) Paperback – December 10, 2013

- Préciser les contraintes dans différents diagrammes : contraintes temporelles, contraintes d'objets, attributs et associations dérivées, contraintes d'héritage, …

- Ocl définit un langage de prédicat muni d'une opération de navigation.

6.2.2. Types et opérations de base du langage

Types et opération de base

Type	Valeurs	Opérations
Boolean	True, false	And, or, xor, not, implies, if-then-else
Integer	3, -15	*, +, -, /, div, mod, min, max, …
Real	2.212, -1.777	*, +, -, /, abs, floor, round, ..
String	'abc'	Size, concat, toUpper, toLower, substring

Types énumérés

Les types énumérés en OCL se fait par une déclaration d´une classe UML stéréotypée par «enumeration»

Types de collections

- Set : ensemble au sens mathématique, pas de doublons, pas d'ordre. Exemple : { 1, 4, 3, 5 } : Set(Integer)

- OrderedSet : idem mais avec ordre (les éléments ont une position dans l'ensemble)

- Bag : comme un Set mais avec possibilité de doublons. Exemple: { 1, 4, 1, 3, 5, 4 } : Bag(Integer)

- Sequence : un Bag dont les éléments sont ordonnés

OCL propose un ensemble d'opérations de base des collections

- size() : int : le nombre d'éléments de la collection

- isEmpty() : Bool : retourne vrai si la collection est vide

- notEmpty() : Bool: retourne vrai si la collection n'est pas vide

- includes(T): Bool : test d´appartenance d´une valeur

- excludes(T): Bool : test de non appartenance d´une valeur

- count(T) : int: integer : le nombre d'occurrences d´une valeur dans la collection

- includesAll(col(T)): Bool : la collection contient tous les éléments de la collection paramètre

- excludesAll(col(T)) : Bool: la collection ne contient aucun des éléments de la collection paramètre

- **sum():T** : addition des élément de la collection (T+T)

- **product(Coll(T1)) Set(Tuple:(first:T, second:T1)):** produit cartésien de deux collections

- **Exists(Expr):Bool**: au moins un élément vérifie l´expression

- **forAll(Expr):Bool**: tous les éléments vérifient l´expression

- **isUnique(Expr):Bool** : retourne vrai si l´expression est évaluée differement pour chaque élément

- **any(Expr):T** : retourne un élément pour lequel l´expression est vraie

- **one(Expr):Bool** : retourne vrai si l´expression est vraie pour un seul élément

- **iterate(Expr)** : évalue l´expression sur chaque élément, le résultat dépend de l´élément et de l´évaluation du précédent

- collect(Expr): Coll(T): retourne la collection des évaluations de l´expression sur chaque élément

OCL propose un ensemble d´opérations d´itération des collections

- select (Expr): CollC(T): retourne la collection des élément vérifiant léxpression.

- reject (Expr): CollC(T) : retourne la collection sans les éléments vérifiant léxpression**- collectNested(Expr): CollC(T)** : retourne la collection des évaluations de léxpression sur chaque élément

- sortedBy(Expr): CollC(T) : retourne la collection ordonnée des élément trée par évaluation de l´expression.

OCL propose un ensemble d´opérations des ensembles ordonnés :

-append(T): OSet(T) : ajout à la fin

- prepend(T): OSet(T) : ajout au début

- insertAt(Int, T):OSet(T) : lajout à la position index

- subOrderedSet(low, up) : l'ordered set contenant les éléments de la position lower à upper à partir d'un ordered set

- at(i):T : séquence á la position i

- indexOf(T):Int : la position de l'élément dans la collection

- first() : le premier élément de l´ensemble

- last() : le dernier élément de l´ensemble

OCL propose un ensemble d´opérations des ensembles

- =(Set(T)):Bool: vrai si les deux ensembles ont les mêmes éléments

- union(Set(T)):Set(T) : l'union des deux ensembles

- intersection(Set(T)):Set(T) : l'intersection des deux ensembles

- -**(Set(T)):** la différence de deux ensembles

- including(T): Set(T): ajout de l´élément á l´ensemble

- **excluding(T): Set(T)**: retrait de l´élément á l´ensemble

- symmetricDifference : différence entre l´union et l´intersection de deux ensembles

6.2.3. Utilisation pratique d´OCL

- **Chaque expression OCL est évaluée dans un contexte donné. OCL propose quatre contextes d´évaluation principaux**
  - La classe (invariants inv[nom]:, valeur initiale init: ou carctéristique dérivé derive:)

Exemple :

Le solde d'un compte doit toujours être positif.

context Compte
inv : solde > 0

  - L´opération (pré-condition pre[nom]: et post-condition post [nom]:)
  - La déclaration globale (contraintes générales)
  - La déclaration locale (expression let <OclDeclaration> in <OclExpression>)
- **La variable self désigne l´objet receveur.**
- **L´expression Let sert á définir des variables locales.**
- **La naviation en OCL permet de passer d´un objet á l´autre via une association:**

- Pour une association qualifiée, on utilise comme une collection indexée (tableau, dictionnaire): ensemble de valeurs ou valeur á l´index donné.

- Pour le classes-associations unaires, on ajoute le nom du rôle comme clé du nom de l´association (obj.nomAssoc[rôle])

6.2.4. Utilisation d´OCL pour exprimer les contraintes[18]

- **Un invariant**

Un invariant exprime une contrainte sur un objet ou un groupe d'objets qui doit être respectée en permanence : inv

Exemples:

context Compte

inv: solde > 0 -- solde d'un compte doit toujours être positif

context Personne

inv: (age <= 140) and (age >=0) -- l'age est compris entre 0 et 140 ans

context Personne

inv : (self.age <= 140) and (self.age >=0)

context p:Personne inv borneAge: (p.age <= 140) and (p.age >=0)

«.» permet d'accéder á une caractéristique d'un objet (méthode, attribut, terminaison d'association) self objet désigné par le contexte.

- **Pré-conditions et Post-conditions**

- Spécifie une contrainte qui doit être vérifiée avant/après l'appel d'une opération.

Exemple:

context Personne::setAge(a :entier)

pre : (a <= 140) and (a >=0) and (a >= age)

post : age = a -- on peut écrire également a=age

Exemple :

context Compte::d´ebiter(somme : Real)

pre: somme > 0

post: solde = solde@pre - somme --@pre : accès á la valeur d'une propriété avant l'exécution de l'opération

context Compte::getSolde() : Real

post: result = somme--result : accès au résultat de l'opération

[18] https://perso.liris.cnrs.fr/laetitia.matignon/index/ISI32012/coursOCL.pdf

- **Valeur initiale et dérivée d'un attribut**

Spécifier la valeur initiale ou dérivé d'un attribut

Exemple:

context Personne::age :entier

init : 0

context Personne::age : entier

derive : dateDeNaissance.getYear() - Date::current().getYear()

- **Définition de variables et d'opérations**

- Définir des variables pour simplifier l'écriture de certaines expressions

syntaxe : let variable : type = expression1 in expression2

Exemple:

```
context Personne
inv : let montantImposable : Real = salaire*0.8 in
if (montantImposable >= 100000)
then impot = montantImposable*45/100
else if (montantImposable >= 50000)
then impot = montantImposable*30/100
else impot = montantImposable*10/100
Endif
Endif
```

- Définir des variables/opérations utilisables dans plusieurs contraintes de la classe

syntaxe : def variable : type = expression1

Exemple:

```
context Personne
def : montantImposable : Real = salaire*0.8
context Personne def : ageCorrect(a :Real) :Boolean = (a>=0) and (a<=140)
```

on peut réécrire l'invariant sur l'age :

context Personne

inv: ageCorrect(age) -- l'age ne peut depasser 140 ans

6.2.5. Outils supportant OCL

Il existe de nombreux outils qui permettent de vérifier la syntaxe, la sémantique et d'évaluer une expression OCL :

Use 5.0 (cf. http://www.db.informatik.uni-bremen.de/projects/USE/)

Dresden OCL Toolkit 2.0 (cf. http://dresden-ocl.sourceforge.net/)

LCI OCL Evaluator OCLE 2.0.4 (cf. http://lci.cs.ubbcluj.ro/ocle/)

The Kent OCL library v1 (cf. http://www.cs.kent.ac.uk/projects/ocl/)

Octopus OCL (cf. http://octopus.sourceforge.net/)

RoclET (cf. http://www.roclet.org)

Topcased (cf. http://www.topcased.org)

...

Plus d'outils : http://www.klasse.nl/ocl/ocl-tools.html ...

Exercices corrigés [19]

Exercice N°01

Le directeur d'une chaîne d'hôtels vous demande de concevoir une application de gestion de ses hôtels. Un hôtel est constitué d'un certain nombre de chambres. Un responsable de l'hôtel gère la location des chambres. Chaque chambre se loue à un prix donné.

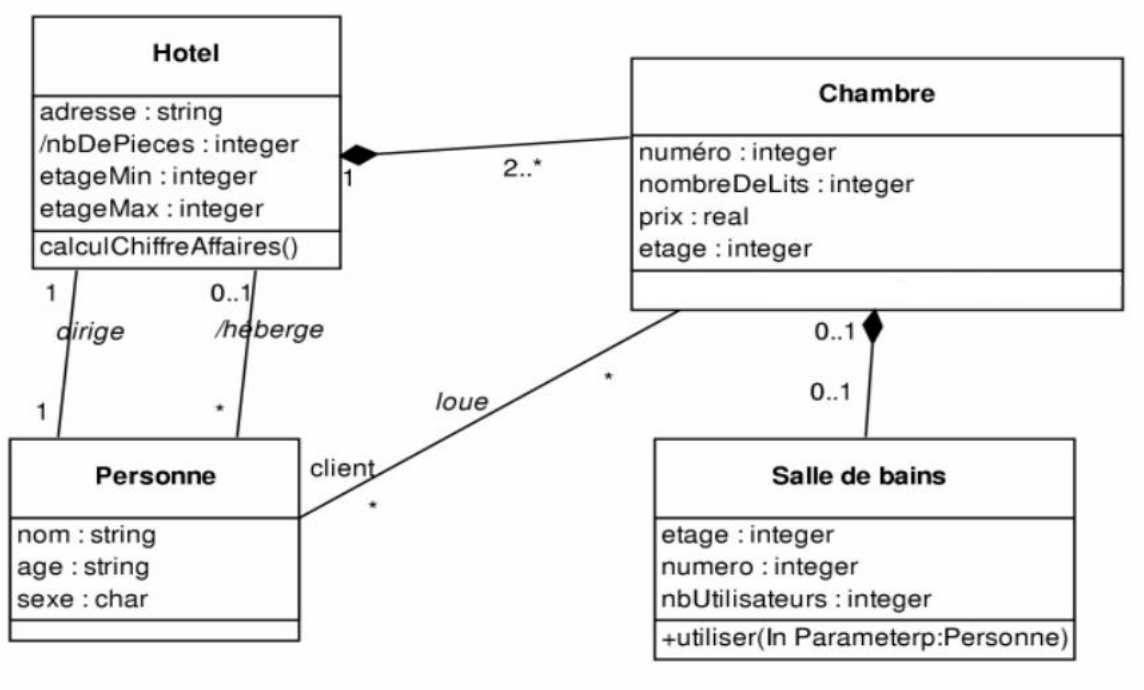
L'accès aux salles de bains est compris dans le prix de la location d'une chambre. Certaines chambres comportent une salle de bains, mais pas toutes. Les hôtes de chambres sans salle de bain peuvent utiliser une salle de bains sur le palier. Ces dernières peuvent être utilisées par plusieurs hôtes.

Les pièces de l'hôtel qui ne sont ni des chambres ni des salles de bain (hall d'accueil, cuisine...) ne font pas partie de l'étude (hors sujet).

[19] https://hlipn.univ-paris13.fr/~gerard/docs/exos/uml-exos05.pdf

Des personnes peuvent louer une ou plusieurs chambres d'hôtel afin d'y résider. En d'autres termes : l'hôtel héberge un certain nombre de personnes, ses hôtes (il s'agit des personnes qui louent au moins une chambre de l'hôtel).

- Donnez le diagramme de classes qui modélise ce problème?



- Donnez une formulation en langage naturel pour chacune des contraintes OCL suivantes :

Q1 :

context Chambre

inv : self.etage <>13

context SalleDeBains

inv : self.etage <>13

R1 : Un hôtel ne contient jamais d'étage numéro 13

Q2 :

context Chambre

inv : client->size <= nombreDeLits or (client->size = nombreDeLits +1 and client->exists(p:Personne | p.age < 4))

R2 : Le nombre de personnes par chambre doit être inférieur ou égal au nombre de lits dans la chambre louée. Les enfants (accompagnés) de moins de 4 ans ne comptent pas dans cette règle de calcul (à hauteur d'un enfant de moins de 4 ans maximum par chambre)

Q3 :

context Hotel

inv : self.chambre->forAll (c : Chambre |c.etage <= self.etageMax and c.etage >= self.etageMin)

R3 : L'étage de chaque chambre est compris entre le premier et le dernier étage de l'hôtel

Q4 :

context Hotel inv :

Sequence{etageMin..etageMax}->forAll(i : Integer |

if i<>13 then

self.chambre->select(c : Chambre | c.etage = i)->notEmpty

endif)

R4 :

Chaque étage possède au moins une chambre (sauf le 13 qui n'existe pas, bien entendu...)

Q5 :

context SalleDeBains::utiliser(p:Personne)

pre : if chambre->notEmpty then chambre.client->includes(p) else p.chambre.etage = self.etage endif

post : nbUtilisateurs = nbUtilisateurs@pre + 1

R6 : Une salle de bain privative ne peut être utilisée que par des personnes qui louent la chambre contenant la salle de bains et une salle de bains sur le palier ne peut être utilisée que par les clients qui logent sur le même palier

Q6:

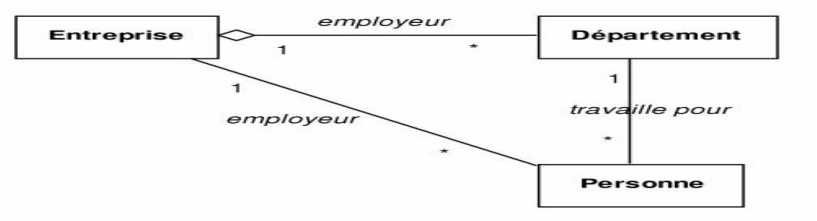
context Hotel::calculerLoyer() : integer

post : result = self.chambre->select(client->notEmpty).prix->sum

R6 : Le loyer de l'hôtel est égal à la somme du prix de toutes les chambres louées

Exercice N°02 :

Considérez le diagramme de classes suivant :



Question 1 : Donnez une expression OCL qui permette d'indiquer que la personne qui travaille dans le département est la même que celle qui est employée par l'entreprise.

context Personne inv :

self.employeur = self.departement.employeur

Question 2 : Donner une expression OCL qui permette d'indiquer qu'une personne travaillant pour une entreprise doit être âgée de 18 ans et plus. On suppose que la classe Personne a un attribut âge.

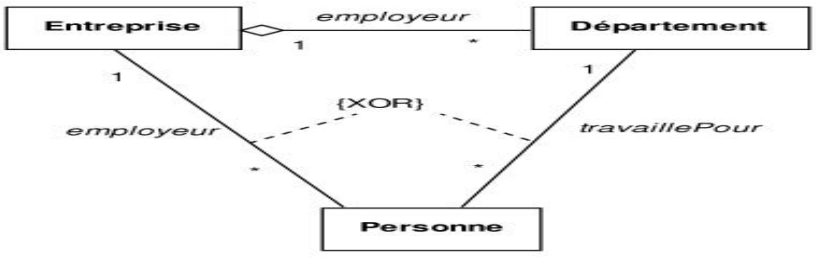
context Personne inv :

self.age > 18

Question 3 : Modifier, graphiquement, le diagramme de classes précédent pour prendre en compte la contrainte suivante :

context personne inv :

(self.departement -> isEmpty) xor (self.Entreprise -> isEmpty)



Question 4 : Ajouter la contrainte indiquant que deux personnes ne doivent pas avoir le même nom.

context Personne inv :

Personne.allinstances -> forAll(p1,p2 | p1<>p2 implies p1.nom <> p2.nom)

7. Chapitre 7/ Introduction aux méthodes de développement[20]

Introduction

Lorsqu'on développe un système, la modélisation est une étape importante qui permet de mieux comprendre les différentes composantes du système et leurs interactions.

Une question fréquente lorsqu'on utilise UML est : "***Dans quel ordre doit-on utiliser les différents types de diagrammes d'UML ?***". En effet, il existe différents types de diagrammes dans UML, chacun étant conçu pour représenter un aspect spécifique du système. Il est donc important de savoir dans quel ordre utiliser ces diagrammes pour une modélisation efficace et complète.

Une autre question qui se pose souvent est : "***A quel moment de la conception d'un système doivent-ils intervenir ?***". En effet, il est important de déterminer le bon moment pour utiliser les différents types de diagrammes d'UML afin de garantir une conception efficace et sans erreur.

Pour répondre á ces question, il faut intégrer l'élaboration des modèles UML dans un ***processus de développement*** pour :

- Guider le développement,
- Maitriser les coûts du développement
- Maitriser les délais associés à chaque phase
- Gérer les risques liés aux projets
- Assurer l'évolutivité du système en tenant compte de nouveaux besoins au fur et mesure ou des changements qui se produisent dans l'environnement de l'entreprise.

Un processus de développement logiciel

Un processus définit une séquence d'étapes ordonnées, qui concourent à l'obtention d'un *système logiciel* ou à l'évolution d'un *système existant*.

[20] Ce chapitre est fortement inspiré de :
- UML 2 et les design patterns, analyse et conception orientées objet et développement itératif, Craig Larman, 3ème édition Pearson Education, 2005 (disponible au biblithèque),
- Le Processus Unifié de développement logiciel, Ivar Jacobson, Grady Booch, James Rumbaugh, édition Eyrolles, 2000 (disponible à la bibliothèque),
- UML en action, de l'analyse des besoins à la conception en Java, Pascal Roques et Franck Vallée, Deuxième édition 2003, EYROLLES,

L'objet d'un processus de développement est de produire des *logiciels de qualité* qui *répondent aux besoins de leurs utilisateurs* dans *des temps* et *des coûts prévisibles*.

Le processus de développement régit les activités de production du logiciel selon deux aspects.

L'aspect statique qui représente le processus en termes de tâches à réaliser. Il définit:

- Le « qui ». Les intervenants
- Le « comment ». Les activités à réaliser
- Le « quoi ». Les résultats d'une activité

L'aspect dynamique qui représente la dimension temporelle du processus. Il représente :

- Le nom et le nombre de phases du cycle de vie du processus
- L'organisation et l'ordre de réalisation des activités de développement

Les activités d'un processus de développement

Un processus de développement gère généralement les activités suivantes: L'expression des besoins, La spécification des besoins, L'analyse des besoins, La conception, L'implémentation, Les tests et La maintenance.

UML et les activités de développement

Les auteurs d'UML insistent sur les caractéristiques suivantes qui sont essentielles pour un processus de développement basé sur UML :

- Piloté par les cas d'utilisation
- Centré sur l'architecture
- Itératif et incrémental

Pour bien répondre à ces exigences, le ***Processus Unifié UP (Unified Process)*** est le résultat des efforts déployés pour fournir un processus accompagnant UML.

7.1. Processus unifié

Le processus unifié est un processus générique qui peut être adapté à différents contextes :

- A une large classe de systèmes logiciels,

- A différents domaines d'application, à différents types d'entreprises, à différents niveaux de compétences et à différentes tailles de l'entreprise.

Le processus unifié est un processus de développement logiciel construit sur UML. Il est itératif, centré sur l'architecture, piloté par des cas d'utilisation et orienté vers la diminution des risques.

Il regroupe les activités à mener pour transformer les besoins d'un utilisateur en système logiciel (Figure suivante).

Figure 2: les entrées /sorties d´un processus unifié

7.2. Caractéristiques d'un processus unifié

7.2.1. UP est piloté par les cas d'utilisation

L'objectif principal d'un système logiciel est de rendre service à ses utilisateurs, il faut par conséquent bien comprendre les désirs et les besoins des futurs utilisateurs. Le processus de développement répond aux besoins métiers à travers des cas d'utilisation qu'ils permettent de détecter puis de décrire les besoins du point de vue de l'utilisateur. Les cas d'utilisation vont guider le processus de développement à travers l'utilisation de modèles basés sur l'utilisation du langage UML

- A partir du modèle des cas d'utilisation, les développeurs créent une série de modèles de conception et d'implémentation réalisant les cas d'utilisation.
- Chacun des modèles successifs est ensuite révisé pour en contrôler la conformité par rapport au modèle des cas d'utilisation.
- Enfin, les testeurs testent l'implémentation pour s'assurer que les composants du modèle d'implémentation mettent correctement en œuvre les cas d'utilisation.

7.2.2. UP est centré sur l'architecture

C'est-à-dire il prend en compte l'architecture de l'entreprise. Dès le démarrage du processus, on aura une vue sur l'architecture à mettre en place. L'architecture d'un système logiciel peut être décrite comme les différentes vues du système qui doit être construit.

L'architecture subit également l'influence d'autres facteurs :

- la plate-forme sur laquelle devra s'exécuter le système ;

- les briques de bases réutilisables disponibles pour le développement ;
- les considérations de déploiement, les systèmes existants et les besoins non fonctionnels (performance, fiabilité…).

7.2.3. UP est itératif et incrémental

Le développement d'un grand produit logiciel peut s'étendre sur plusieurs mois. On ne va pas tout développer d'un coup. On peut découper le travail en plusieurs parties qui sont autant de mini projets. Chacun d'entre eux représentant une itération qui donne lieu à un incrément.

Une itération désigne la succession des étapes de l'enchaînement d'activités, tandis qu'un incrément correspond à une avancée dans les différents stades de développement.

Chaque itération parcourt les cinq enchaînements d'activités principaux (Figure suivante).

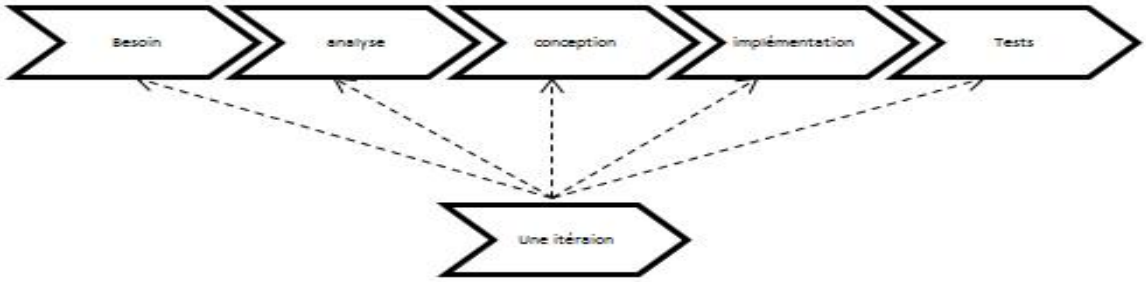


Figure 3. Représentation d´une itération

Les itérations parcourent les enchaînements d'activités, de la « capture » des besoins aux tests. Le choix de ce qui doit être implémenté au cours d'une itération repose sur deux facteurs :

- Une itération prend en compte un certain nombre de cas d'utilisation qui ensemble, améliorent l'utilisabilité du produit à un certain stade de développement.
- L'itération traite en priorité les risques majeurs.

A chaque itération, les développeurs identifient et spécifient les cas d'utilisations pertinents, créent une conception en se laissant guider par l'architecture choisie, implémentent cette conception sous forme de composants et vérifient que ceux-ci sont conformes aux cas d'utilisation.

7.2.4. UP est piloté par les risques

Le processus unifié permet de prendre en compte les besoins des utilisateurs et les spécificités et contraintes de mise en œuvre.

7.2.5. UP est orienté modèles

A travers les étapes du processus unifié, les diagrammes UML sont utilisés (classes, cas d'utilisation, ...) afin de capturer les besoins ou de concevoir le système.

7.2.6. UP est à base de composants

Ce qui signifie que le système logiciel en cours de fabrication est fait de composants logiciels reliés les uns aux autres par des interfaces clairement définies. Le processus unifié permet la réutilisation puisqu'il est centré sur le développement de composants réutilisables.

7.3. Activités et Phases du processus unifié

UP gère le développement selon deux axes (figure suivante[21]) :

- ***L'axe vertical*** représente les principaux enchaînements d'activités, qui regroupent les activités selon leur nature. Cette dimension rend compte *l'aspect statique* du processus qui s'exprime en termes de composants, de processus, d'activités, d'enchaînements, d'artefacts et de travailleurs.
- ***L'axe horizontal*** représente le temps et montre le déroulement du cycle de vie du processus, cette dimension rend compte de l'aspect dynamique du processus qui s'exprime en terme de cycles, de phases, d'itérations et de jalons.

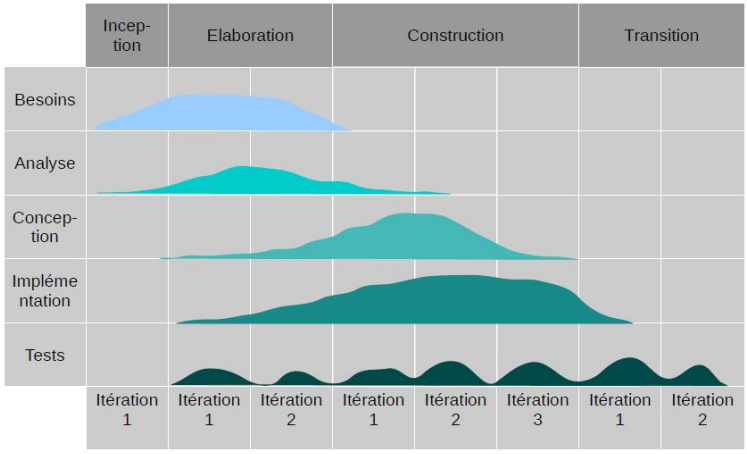


Figure 4. Activités et phases du processus unifié

7.3.1. Les Activités

Expression des besoins : L'expression des besoins permet de:

- inventorier les besoins principaux et fournir une liste de leurs fonctions
- recenser les besoins fonctionnels (du point de vue de l'utilisateur) qui conduisent à l'élaboration des modèles de cas d'utilisation

[21] https://www.u-picardie.fr/~furst/docs/1-UML_Besoins.pdf

- appréhender les besoins non fonctionnels (technique) et livrer une liste des exigences.

Analyse : L'objectif de l'analyse est d'accéder à une compréhension des besoins et des exigences du client. Il s'agit de réaliser des spécifications permettant de concevoir la solution.

Conception : La conception permet d'acquérir une compréhension approfondie des contraintes liées au langage de programmation, à l'utilisation des composants et au système d'exploitation. Elle détermine les principales interfaces et les transcrit à l'aide d'une notation commune.

- Elle constitue un point de départ à l'implémentation :
- Elle décompose le travail d'implémentation en sous-système
- Elle créée une abstraction transparente de l'implémentation

Implémentation : L'implémentation est le résultat de la conception pour implémenter le système sous formes de composants, c'est-à-dire, de code source, de scripts, de binaires, d'exécutables et d'autres éléments du même type.

Les objectifs principaux de l'implémentation sont de planifier les intégrations des composants pour chaque itération, et de produire les classes et les sous-systèmes sous formes de codes sources.

Test : Les tests permettent de vérifier des résultats de l'implémentation en testant la construction. Pour mener à bien ces tests, il faut les planifier pour chaque itération, les implémenter en créant des cas de tests, effectuer ces tests et prendre en compte le résultat de chacun.

7.3.2. Les Phases du développement du PU

Les phases du développement sont les grandes étapes du développement du logiciel. Le projet commence en phase d'initialisation et se termine en phase de transition.

La phase de création ou d'inception : La phase de création donne une vue du projet sous forme de produit fini. Cette phase porte essentiellement sur les besoins principaux du point de vue de l'utilisateur, l'architecture générale du système, les risques majeurs, les délais et les coûts. Elle répond aux questions suivantes :

- que va faire le système ? pour chacun de ses principaux utilisateurs ?
- A quoi peut rassembler l'architecture d'un tel système ?
- quels vont être : les délais, les coûts, les ressources, les moyens à déployer ?

Elaboration : L'élaboration reprend les éléments de la phase d'analyse des besoins et les précise pour arriver à une spécification détaillée de la solution à mettre en œuvre.

Elle permet de préciser la plupart des cas d'utilisation, de concevoir l'architecture du système et surtout de déterminer l'architecture de référence.

Au terme de cette phase, les chefs de projet doivent être en mesure de prévoir les activités et d'estimer les ressources nécessaires à l'achèvement du projet. Les tâches à effectuer dans la phase élaboration sont les suivantes :

- créer une architecture de référence
- identifier les risques, ceux qui sont de nature à bouleverser le plan, le coût et le calendrier
- définir les niveaux de qualité à atteindre
- formuler les cas d'utilisation pour couvrir les besoins fonctionnels et planifier la phase de construction
- élaborer une offre abordant les questions de calendrier, de personnel et de budget

Construction : La construction est le moment où l'on construit le produit (architecture= produit complet). Le produit contient tous les cas d'utilisation que les chefs de projet, en accord avec les utilisateurs ont décidé de mettre au point pour cette version.

Transition : Un groupe d'utilisateurs essaye le produit et détecte les anomalies et défauts. Cette phase suppose des activités comme la formation des utilisateurs clients, la mise en œuvre d'un service d'assistance et la correction des anomalies constatées.

7.4. Modèles mis en place

Pour mener efficacement le cycle, les développeurs ont besoin de construire toutes les représentations du produit logiciel par un ensemble de modèles (Figure suivante):

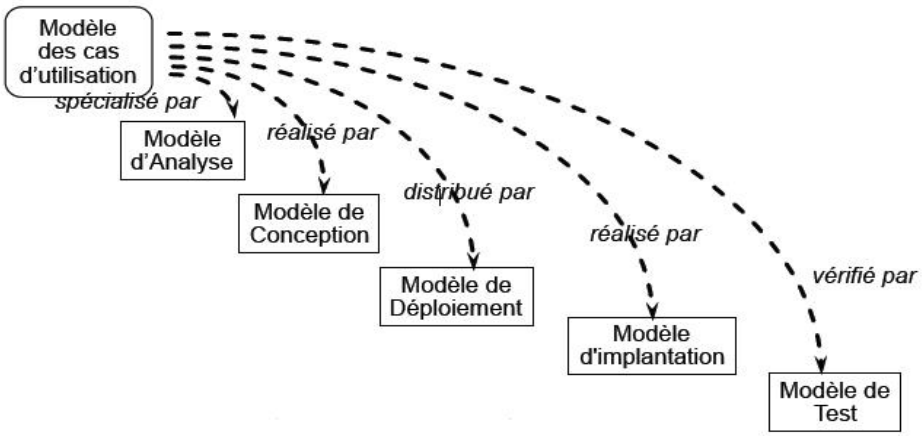


Figure 5. Les modèles du processus unifié mis en place

7.4.1. Modèle des cas d'utilisation (Use Case Model)

Dans UP, ce modèle décrit le système d'un point de vue fonctionnel, il expose les cas d'utilisation et leurs relations avec les utilisateurs et détaille leurs déroulements à l'aide principalement des :

- Diagrammes de cas d'utilisation
- Diagrammes de séquences
- Diagrammes de communication

Ce modèle se construit en relation entre les utilisateurs/client et les développeurs. Il permet d'identifier des éléments internes au système et d'entamer la définition de l'architecture du système dans le *modèle d'analyse*.

7.4.2. Modèle d'analyse (Analysis Model)

Dans UP, le modèle d'analyse décrit le système d'un point de vue structurel, il détaille les cas d'utilisation et procède à une première répartition du comportement du système entre divers objets.

Ce modèle permet de définir l'architecture générale du système à l'aide principalement de :

- Diagrammes de classes
- Diagrammes de paquetages

Et complétés par :

- Diagrammes d'objets
- Diagrammes d'interaction.

A ce stade, les classes n'ont pas être forcément détaillées (types des attributs, visibilité des membres, etc.).

Le modèle de conception va compléter le modèle d'analyse en spécifiant les détails structurels et comportementaux.

7.4.3. Modèle de conception (Design Model)

Il affine la description du système :

- D'un point de vue structurel : on complète des diagrammes de classes et de paquetages
- D'un point de vue fonctionnel : on détaille le fonctionnement du système avec des diagrammes d'interaction (séquences, communication) et des diagrammes d'états et d'activités.

Le modèle de conception est construit à partir des modèles de cas d'utilisation et d'analyse. Il va compléter le modèle d'analyse en spécifiant les détails structurels et comportementaux.

7.4.4. Modèle d'implémentation (Implementation Model)

Il décrit l'architecture du système sous forme de composants. Un composant regroupe un ensemble de briques logicielles (classes en POO[22]) présentant des fonctionnalités liées ou complémentaires et formant un module réutilisable.

Un composant est souvent une librairie (dll, jar, …) partagée entre différentes applications et décrite par des interfaces.

7.4.5. Modèle de déploiement (Deployement Model)

Le modèle de déploiement décrit la façon dont le logiciel sera déployé sur une infrastructure informatique. C'est-à-dire, qu'il définit les nœuds physiques des ordinateurs et l'affectation de ces composants sur ces nœuds.

Il est spécifié essentiellement dans un diagramme de déploiement qui décrit :

- Les éléments matériels et logiciels sur lesquels sont déployés les artefacts (éléments logiciels produits par le développement du système)
- Les protocoles de communications.

7.4.6. Modèle de test (Test Model)

Il décrit les cas de test vérifiant les cas d'utilisation.

7.5. Méthodologies de développement

Il existe plusieurs travaux qui adaptent le processus unifie comme démarche de développement.

Les méthodes agiles sont des méthodes de développement et de gestion de projets informatiques. Elles ont pour priorité la satisfaction réelle du besoin du client.

Les méthodes agiles apportent des solutions itératives, incrémentales et basées sur l'acceptation du changement. Les principales méthodes sont :

- Dynamic Software Development Method (DSDM),
- Rapid Application Development (RAD),
- eXtreme Programming (XP),
- Rational Unified Process (RUP),
- Two Tracks Unified Process (2TUP).

Ces méthodes sont globalement similaires et la plupart des techniques qu'elles préconisent sont communes.

[22] Programmation Orientée Objet

7.5.1. RUP - Rational Unified Process

RUP (Rational Unified Process) est un processus de développement itératif, incrémental et tient compte des besoins des utilisateurs, promu par la société Rational Software. Il se caractérise par une approche globale nommée "Vue 4+1". Les cinq composants de cette vue sont :

- la vue des cas d'utilisation,
- la vue logique,
- la vue d'implémentation,
- la vue de processus et
- la vue de déploiement.

RUP peut être vu comme une implémentation de la démarche générique UP, complétée par une série d'outils informatiques ainsi qu'une documentation des processus décrivant ces outils. Mais ce processus reste très procédural et lié à des outils bien particuliers qui ont tendance à l'alourdir.

7.5.2. 2TUP (2 Tracks UP)

2TUP (2 Tracks UP) est un processus qui répond aux caractéristiques du processus unifié. Il s'appuie sur UML tout au long du cycle de développement. Il est itératif, centré sur l'architecture et conduit par les cas d'utilisation.

Le 2TUP a deux branches qui correspondent aux deux axes de changement (fonctionnel et technique)(Figure suivante[23]) :

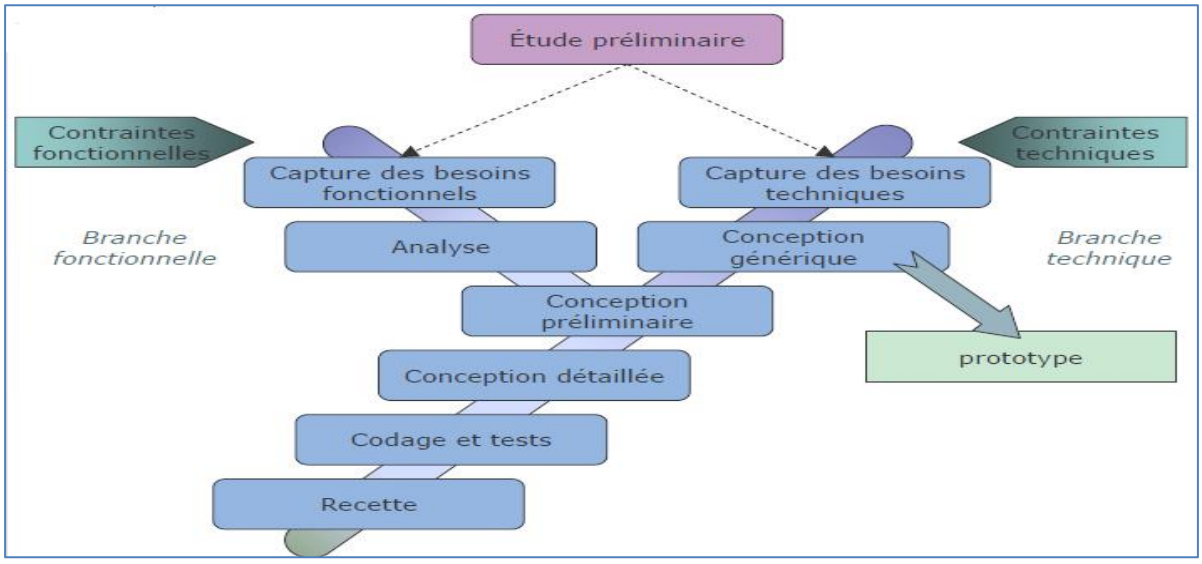


Figure 6. Les étapes de 2TUP

- *Branche fonctionnelle* : capitalise de la connaissance du métier de l'entreprise. Elle comporte les fonctions suivantes :

[23] www.memoireonline.com/05/13/7195/m-mise-en-place-dune-application-webmapping-de-geolocalisation-des-points-direct-de-la-vill6.html

  - La capture des besoins fonctionnels, qui produit un modèle des besoins focalisé sur le métier des utilisateurs
  - L'analyse.
- *Branche technique* : capitalise un savoir-faire technique. Elle comporte les étapes suivantes :
  - La capture des besoins techniques
  - La conception générique
- *La branche du milieu* : consiste à fusionner les résultats des deux branches. Cette fusion conduit à l'obtention d'un processus en forme Y. Elle comporte les étapes suivantes :
  - La conception préliminaire
  - La conception détaillée
  - Le codage
  - L'intégration

Les deux branches (fonctionnelle et technique) sont modifiables indépendamment l'une de l'autre.

7.5.3. XP (eXtreme Programming)

XP (eXtreme Programming) est un ensemble de bonnes pratiques de développement qui vise à remettre le développeur au centre du processus de développement. Ses prises de positions originales (et extrêmes) lui ont valu un succès d'estime depuis plusieurs années, surtout aux Etats-Unis. Les principaux éléments du fonctionnement de XP :

- Gestion des livraisons
- Gestion des itérations
- Suivi du projet
- Qualité du design et du code
- Travail en équipe

7.5.4. Projection de XP et de 2TUP sur la matrice du RUP

Au-delà de l'itératif, on notera que les méthodologies présentées en figure suivante mettent l'accent sur des phases projets différentes.

- Le RUP couvre l'ensemble du processus
- XP se concentre sur la phase de développement, tandis que 2TUP fait une large place à l'analyse et à l'architecture.

Aussi, ces processus peuvent se compléter plutôt qu'entrer en concurrence, comme l'illustre la figure suivante(*Projection de XP et de 2TUP sur la matrice du RUP*[24]).

[24] Processus de développement objet et nouvelles technologies- www.application-servers.com, votre portail sur les serveurs d'applications, 2001.

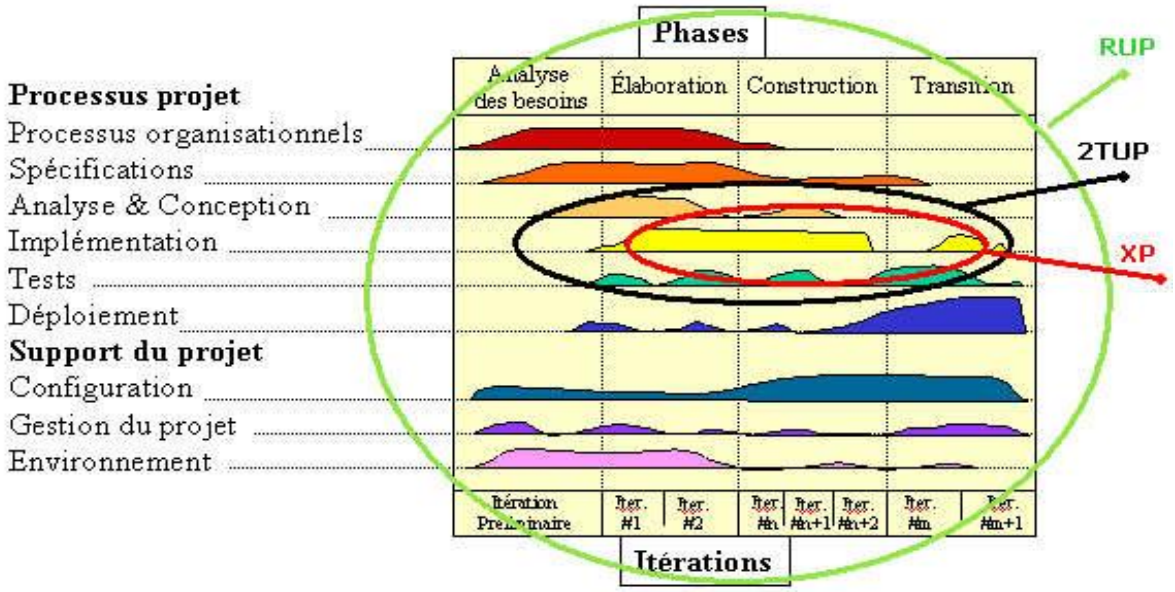


Figure 7. Projection de XP et de 2TUP sur la matrice du RUP

7.5.5. Tableau comparatif

Le tableau suivant[25] présente une synthèse des processus les plus en vogue dans la communauté Objet et Nouvelles Technologies.

[25] Processus de développement objet et nouvelles technologies- www.application-servers.com, votre portail sur les serveurs d'applications, 2001.

	Description	Points forts	Points faibles
Cascade	• Propose de dérouler les phases projet de manière séquentielle • **Cité pour des raisons historiques**	• Distingue clairement les phases projet	• **Non itératif** • Ne propose pas de modèles de documents
RUP Rational Unified Process	• Promu par Rational. • **Le RUP est à la fois une méthodologie et un outil prêt à l'emploi** (documents types partagés dans un référentiel Web) • Cible des projets de plus de 10 personnes	• Itératif • **Spécifie le dialogue entre les différents intervenants du projet : les livrables, les plannings, les prototypes...** • Propose des modèles de documents, et des canevas pour des projets types	• **Coûteux** à personnaliser : batterie de consultants • Très axé processus, au détriment du développement : peu de place pour le code et la technologie
XP eXtreme Programming	• **Ensemble de « Bests Practices » de développement** (travail en équipes, transfert de compétences...) • Cible des projets de moins de 10 personnes	• Itératif • **Simple à mettre en œuvre** • Fait une large place aux aspects techniques : prototypes, règles de développement, tests... • **Innovant**: programmation en duo, kick-off matinal debout ...	• Ne couvre pas les phases en amont et en aval au développement : capture des besoins, support, maintenance, tests d'intégration... • **Elude la phase d'analyse**, si bien qu'on peut dépenser son énergie à faire et défaire • **Assez flou dans sa mise en œuvre**: quels intervenants, quels livrables ?
2TUP Two Track Unified Process	• **S'articule autour de l'architecture** • **Propose un cycle de développement en Y** • Détaillé dans « UML en action » (voir références) • Cible des projets de toutes tailles	• Itératif • **Fait une large place à la technologie et à la gestion du risque** • Définit les profils des intervenants, les livrables, les plannings, les prototypes	• **Plutôt superficiel sur les phases situées en amont et en aval du développement** : capture des besoins, support, maintenance, gestion du changement... • Ne propose pas de documents types

Conclusion

Nous avons présenté, dans ce chapitre, une introduction sur les méthodes de développement. Dans le chapitre suivant, nous allons présenter les patrons de conception et leur place dans le processus de développement.

8. Chapitre 8. Patrons de conception et leur place au sein du processus de développement[26]

Introduction

Les Design Patterns ont plusieurs noms francisés qui permettent de les désigner comme : Motif de conception, Modèle de conception ainsi que Patron de conception.

8.2. Historique

Au début des années 70, les travaux de Christopher Alexander à la phase de conception en architecture apparaître des problèmes récurrents. Il cherche à résoudre l'ensemble de ces problèmes liés à des contraintes interdépendantes : solidité de la structure, étanchéité....

Il établit un langage de 253 patterns. Ces patrons couvrent tous les aspects de la construction par exemple : la façon de concevoir une charpente.

Dans les années 90, l'idée de Christopher Alexander va être reprise et étendue au domaine de la conception des logiciels et le concept de Design Pattern est développé dans un ouvrage publié en 1995 par le 'Gang of Four'. Cette équipe qui regroupe : Erich Gamma, Richard Helm, Ralph Johnson et John Vlissides

Le livre : 'Design Patterns : Elements of Reusable Object-Oriented Software' Présente 23 Design Patterns, et est consédéré comme une référence dans le monde de l'informatique.

8.3. Définition

Un patron de conception est défini comme une solution à un problème récurrent dans la conception d'applications orientées objet. Il décrit la solution éprouvée pour résoudre ce problème d'architecture de logiciel.

Exemple : Comme problème récurrent on trouve par exemple la conception d'une application où il sera facile d'ajouter des fonctionnalités à une classe sans la modifier (voir la solution du Design Pattern Visiteur).

[26] Laurent DEBRAUWER, Design Patterns en java, les 23 modèles de conception : descriptions et solutions illustrées en UML 2 et Java, edition eni, 2013.

- A noter qu'en se plaçant au niveau de la conception les Design Patterns sont indépendants des langages de programmation utilisés.

8.3. Représentation d'un patron de conception

Un patron de conception est représenté par :

- ***Nom*** : qui permet de l'identifier clairement
- ***Problématique*** : c'est une description du problème auquel il répond
- ***Solution*** : c'est une description de la solution souvent accompagnée d'un schéma UML

8.4. Les avantages

L'utilisation des Design Patterns offre de nombreux avantages.

- Tout d'abord cela permet de répondre à un problème de conception grâce à une solution éprouvée et validée par des experts.
- Ainsi on gagne en rapidité et en qualité de conception ce qui diminue également les coûts.
- De plus, les Design Patterns sont réutilisables et permettent de mettre en avant les bonnes pratiques de conception.
- Les Design Patterns étant largement documentés et connus d'un grand nombre de développeurs ils permettent également de faciliter la communication.
- Si un développeur annonce que sur ce point du projet il va utiliser le Design Pattern Observateur il est compris des informaticiens sans pour autant rentrer dans les détails de la conception (diagramme UML, objectif visé...).

8.5. Organisation des patrons de conception

Les patrons de conception sont classés en trois catégories :

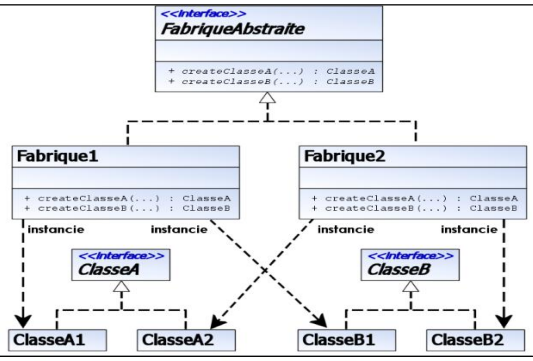
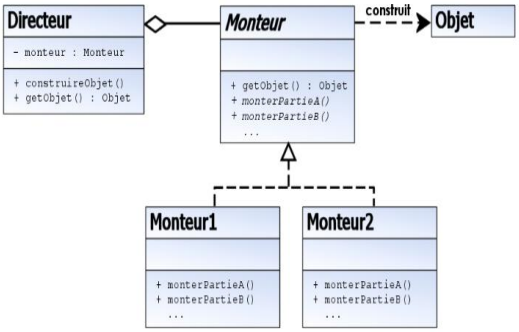
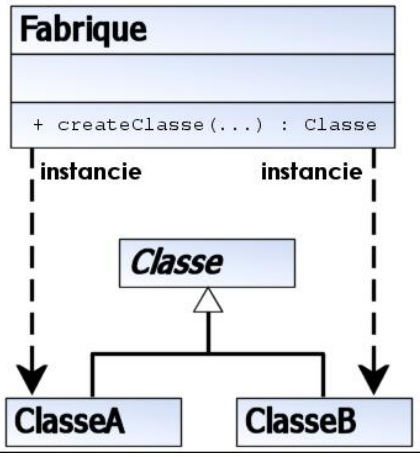
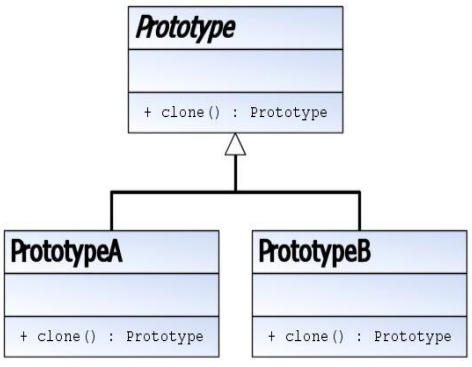
Les patrons de création

Ils permettent d'instancier et de configurer des classes et des objets. Déterminent comment faire l'instanciation et la configuration des classes et des objets.

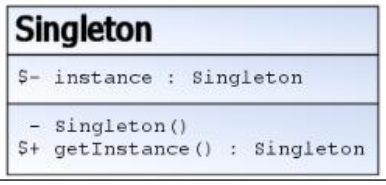
Ces patterns permettent de rendre indépendant la façon dont les objets sont créés :

- Ils encapsulent l'utilisation des classes concrètes
- Ils favorisent l'utilisation des interfaces (abstraites)
- Ils augmentent les capacités d'abstraction dans la conception globale du système

Les différents patrons de création sont les suivants[27] :

Le patron de conception	Description	Schéma UML
Fabrique abstraite (abstract factory)	Il offre la possibilité de gérer diverses fabriques concrètes en utilisant l'interface d'une fabrique abstraite.	
Moniteur (builder)	Sa fonctionnalité consiste à construire des objets complexes en construisant chacune de ses parties sans être dépendant de leur représentation concrète.	
Fabrique (factory method)	Il offre la possibilité de créer un objet dont la classe dépend des paramètres de construction, tels qu'un nom de classe.	
Prototype (prototype)	Sa fonctionnalité consiste à créer un nouvel objet en copiant un objet existant. Il permet également d'appeler un constructeur pour créer un objet, puis de configurer la valeur de ses attributs.	

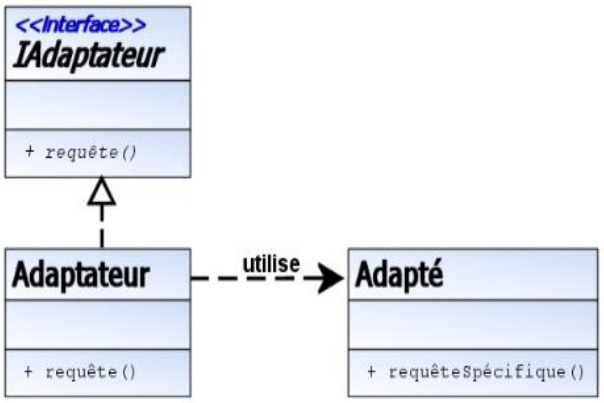
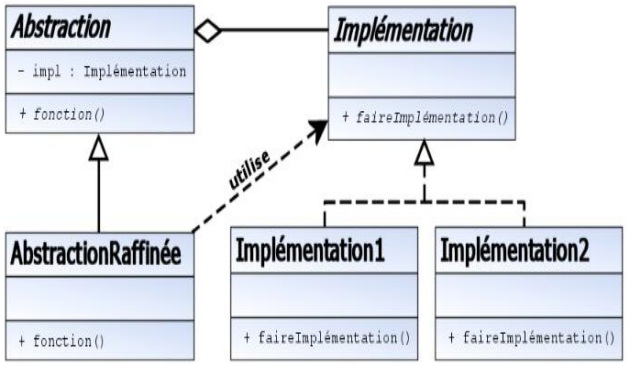
27 https://fr.wikibooks.org/w/index.php?title=Patrons_de_conception/Version_imprimable&printable=yes#Patrons_du_%C2%AB_Gang_of_Four_%C2%BB

Singleton (singleton)	Il est utilisé lorsqu'une classe ne peut être instanciée qu'une seule fois.	

Les patrons de structure[28]

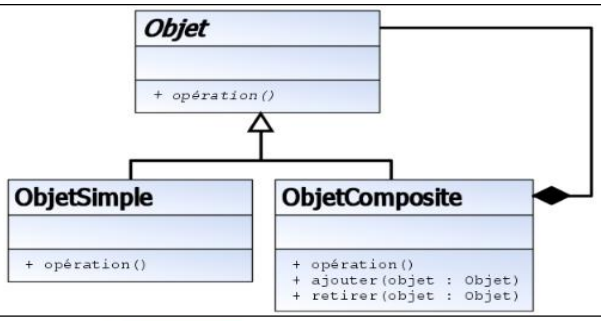
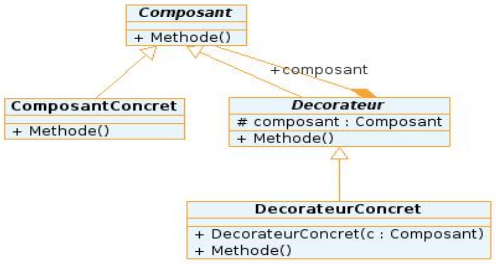
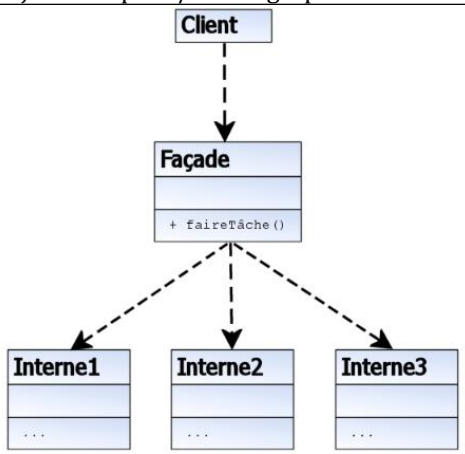
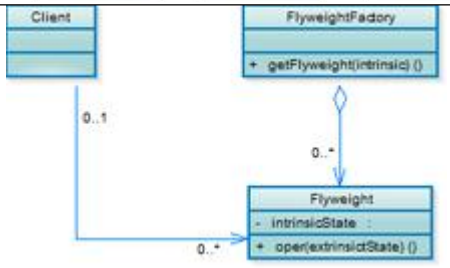
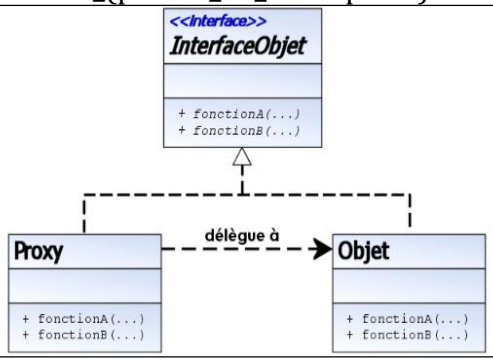
Ils permettent d'organiser les classes d'une application. Ils permettent de résoudre les problèmes liés à la structuration des classes et leur interface.

Les différents patrons de structure sont les suivants[29] :

Le nom du patron de conception	Description	Schéma UML
Adaptateur (adapter)	Sa fonctionnalité consiste à adapter une interface existante à une autre interface.	 <<Interface>> IAdaptateur + requête() Adaptateur + requête() utilise Adapté + requêteSpécifique()
Pont (bridge)	Il offre la possibilité d'utiliser une interface à la place d'une implémentation spécifique, ce qui permet de garantir l'indépendance entre l'utilisation et l'implémentation.	 Abstraction - impl : Implémentation + fonction() Implémentation + faireImplémentation() utilise AbstractionRaffinée + fonction() Implémentation1 + faireImplémentation() Implémentation2 + faireImplémentation()

28 https://fr.wikibooks.org/w/index.php?title=Patrons_de_conception/Version_imprimable&printable=yes#Patrons_de_structure

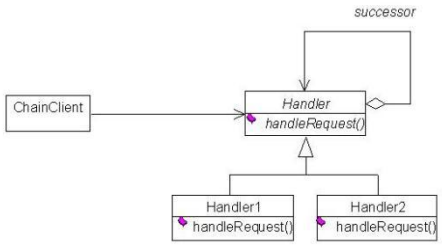
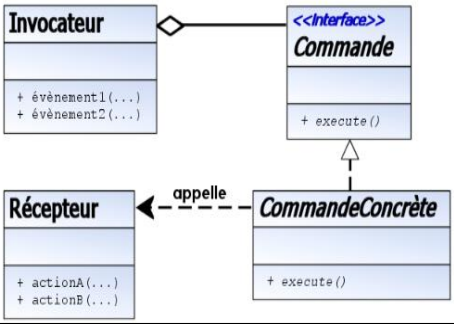
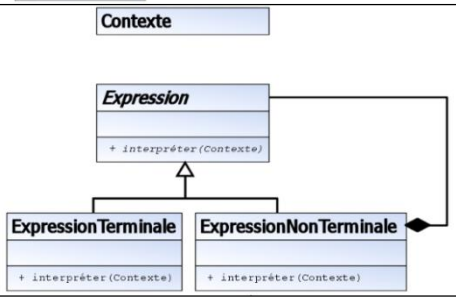
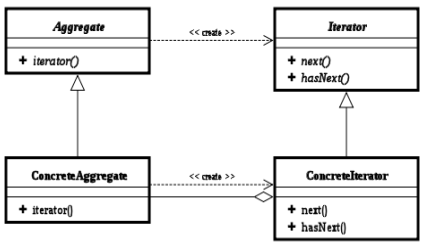
29 https://fr.wikibooks.org/w/index.php?title=Patrons_de_conception/Version_imprimable&printable=yes#Structure

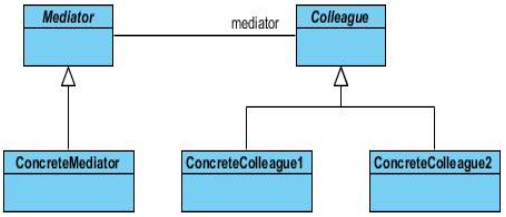
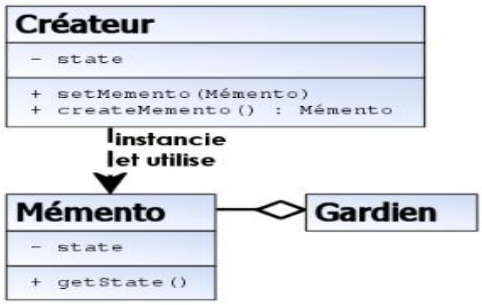
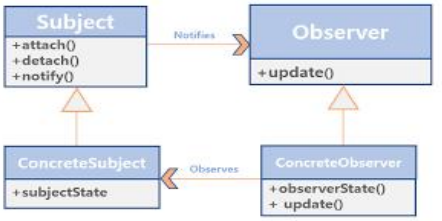
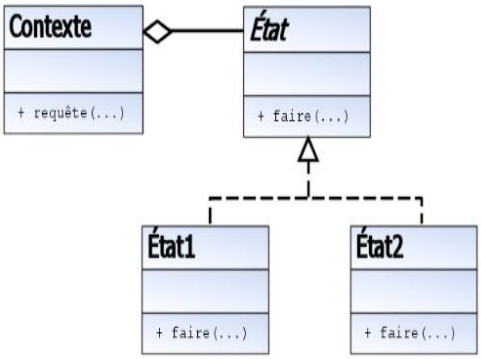
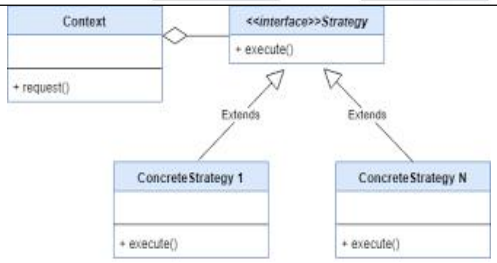
Objet composite (composite)	Sa fonctionnalité consiste à manipuler des objets composites à travers la même interface que les éléments qui les composent.	
Décorateur (decorator)	Il offre la possibilité d'ajouter de nouvelles responsabilités à un objet de manière dynamique.	 Source : https://www.codingame.com/playgrounds/8396/simple-java-template/le-design-pattern-decorator
Façade (facade)	Sa fonctionnalité consiste à simplifier l'utilisation d'une interface complexe.	
Poids-mouche ou poids-plume (flyweight)	Sa fonctionnalité consiste à réduire le nombre de classes en fusionnant des classes similaires en une seule et en transmettant des paramètres supplémentaires aux méthodes appelées.	 Source : https://fr.wikipedia.org/wiki/Poids-mouche_(patron_de_conception)
Proxy (proxy)	Il offre la possibilité de remplacer une classe par une autre en utilisant la même interface, ce qui permet de contrôler l'accès à la classe	

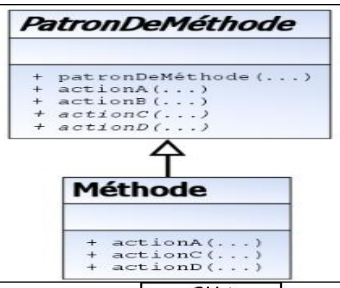
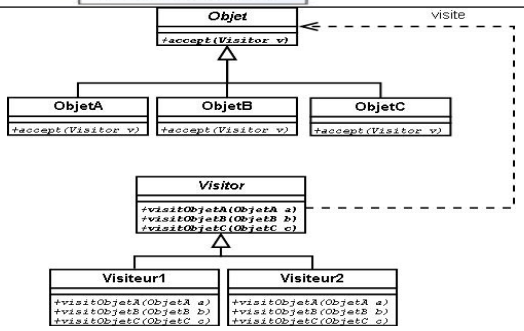
Les patrons de comportement

Ils permettent d'organiser les objets pour qu'ils collaborent entre eux. Ils permettent de résoudre les problèmes liés aux comportements, à l'interaction entre les classes.

Les patrons de conception de comportement sont les suivants :

Le nom du patron de conception	**Description**	**Schéma en UML**
Chaîne de responsabilité (chain of responsibility)	Sa fonctionnalité consiste à construire une chaîne de traitement pour une même requête	 Source : CBR*Tools - Chaîne de Responsabilité (Chain of Responsibility) (inria.fr)
Commande (command)	Il permet d'encapsuler l'appel d'une commande.	
Interpréteur (Interpreter)	Il consiste á interpréter un langage spécialisé	
Itérateur (iterator)	Il offre la possibilité de parcourir un ensemble d'objets en utilisant un objet de contexte.	 Source : https://stackoverflow.com/questions/68990162/why-are-there-different-ways-to-describe-same-design-

		pattern-using-uml-what-to
Médiateur (mediator)	Il consiste à limiter les dépendances entre un ensemble de classes en ayant recours à une classe médiateur qui sert d'intermédiaire de communication.	 Source : https://circle.visual-paradigm.com/mediator/
Memento (memento	Il offre la possibilité de conserver l'état d'un objet pour pouvoir le restaurer ultérieurement.	
Observateur (observer)	Il consiste à intercepter un événement afin de le traiter.	 Source : https://hyosup0513.github.io/design%20pattern/2020/08/09/Observer-Pattern.html
Etat (state)	Il permet de gérer divers états en utilisant des classes distinctes	
Stratégie (strategy)	Il permet de modifier de manière dynamique la stratégie (l'algorithme) utilisée en fonction du contexte.	 Source: Strategy Design Pattern In Java - Java Code Geeks - 2023

Patron de méthode (template method)	Il offre la possibilité de définir un modèle de méthode en recourant à l'utilisation de méthodes abstraites.	
Visiteur (visitor	Il consiste à séparer les classes et les traitements, de manière à pouvoir intégrer de nouveaux traitements sans avoir à ajouter de nouvelles méthodes aux classes existantes.	 Source : Patron de conception Visiteur - Deseign Patterns (wikidot.com)

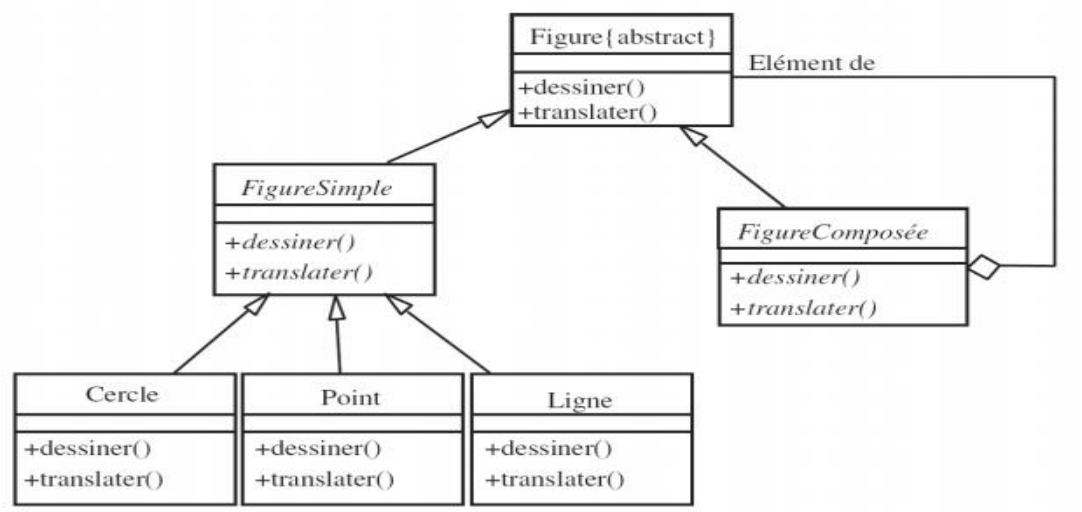
Exercices corrigés [30]

Exercice N°01 :

Une figure simple peut être un point, une ligne ou un cercle. Une figure peut être composée d'autres figures, simples ou elles-mêmes composées d'autres figures. Toutes les figures peuvent être dessinées ou translatées.

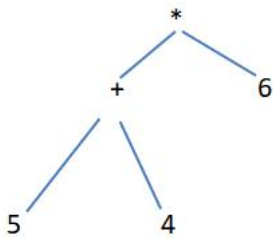
Utilisez le pattern composite pour construire un diagramme de classe rendant compte de cette hiérarchie d'objets.

[30] Extraits de : Charroux, Benoît, Aomar Osmani, and Yann Thierry-Mieg. UML 2: pratique de la modélisation. Paris: Pearson Education, 2010.



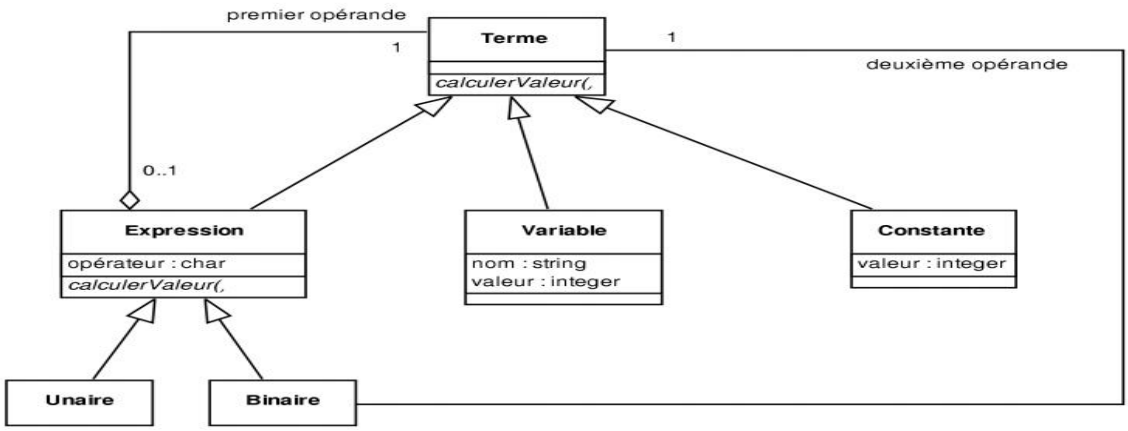
Exercice N°2:

On peut représenter une expression arithmétique sous forme d'arbre. Par exemple, l'expression (5+4)*6 peut être représentée par l'opération * appliquée à 6 et au résultat de l'opération + entre 5 et 4. Dans cette représentation, 6 et +(5,4) sont considérés comme des opérandes de l'expression, tandis que * est l'opérateur. Une autre notation pour cette expression arithmétique est la notation polonaise, également utilisée sur les calculatrices HP, qui s'écrit *(+(5,4),6). Cette notation peut également être représentée sous forme d'arbre. Avec cette notation, les parenthèses ne sont pas nécessaires, car il n'y a pas d'ambiguïté lorsqu'on utilise la notation *+546.



Il existe deux catégories d'expressions : les expressions binaires, telles que +(5,4), et les expressions unaires qui utilisent des opérations à un seul argument, comme l'opérateur de changement de signe dans l'expression 1. Les deux types d'expression sont caractérisés par un opérateur et disposent d'une opération "calculerValeur()". Le terme est un concept plus général qu'une expression, pouvant être une valeur constante (1, 2, 3 ou 4), une expression comme +(5,4), ou une variable ayant un nom et une valeur associée. Dans tous les cas, un terme doit avoir une opération "calculerValeur()".

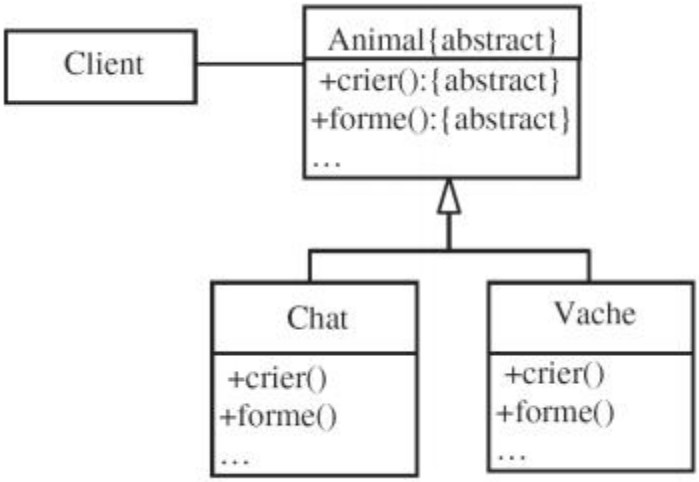
Question : Utilisez le patron composite pour produire un diagramme de classes adéquat pour la représentation des expressions arithmétiques.



Exercice N°03:

Un jeu éducatif destiné aux enfants a été développé par un éditeur de jeux pour leur apprendre les différentes espèces animales. Les enfants peuvent apprendre les caractéristiques distinctes de chaque animal, y compris leur forme et leur cri, tels que le chat et la vache. Pour ce faire, la classe LeChat a été créée avec les méthodes formeChat() et criChat(), tandis que LaVache a été modélisée avec les méthodes criVache() et formeVache(). Toutefois, l'éditeur souhaite améliorer ce jeu en créant une interface universelle pour tous les animaux, qui lui permettra d'en ajouter de nouveaux sans avoir à modifier l'interface avec le code client. Cette interface commune facilitera également l'utilisation du polymorphisme dans la gestion des animaux, permettant ainsi de manipuler des troupeaux hétéroclites.

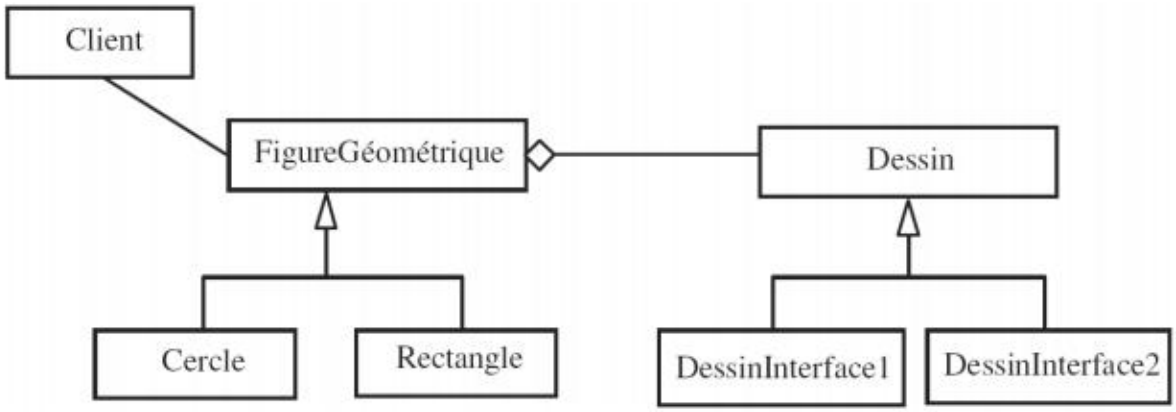
Question : Proposez une modélisation des classes pour cette nouvelle version du jeu en faisant apparaître le client. Les classes seront Chat, Vache... et non plus LeChat, LaVache...



Exercice N°04:

Pour le dessin des figures géométriques, deux formes sont possibles : le cercle et le rectangle. Afin de proposer un outil de dessin de figures géométriques aux clients informatiques, il est nécessaire de l'adapter à leur environnement spécifique. Les interfaces ainsi que la qualité des dessins dépendent donc de cet environnement particulier. Le client est responsable de dessiner les figures géométriques en question. Chaque figure est définie par un ensemble de propriétés, qui incluent des caractéristiques liées au dessin telles que la couleur ou l'épaisseur du trait. Ces propriétés sont spécialisées et varient en fonction de l'environnement dans lequel les figures géométriques sont dessinées.

Question : En utilisant le patron bridge, proposez une modélisation qui isole le dessin d'une figure géométrique et qui la spécialise en fonction de l'environnement.



Travaux Pratiques :

Objectifs :

- Effectuer une recherche sur les fonctionnalités d'une application orientée objet.
- Modéliser ces fonctionnalités sous forme de cas d'utilisation et de scénarios.
- Identifier les classes et acteurs nécessaires à la conception de l'application.
- Appliquer les patrons de conception (design patterns).
- Structurer l'architecture logicielle en couches.

TP01: Analyse, Conception et Implémentation d'une Application de Gestion des Unités d'Enseignement

Ce TP permettra aux étudiants de mettre en pratique les concepts clés du génie logiciel, de la phase d'analyse à la phase d'implémentation, tout en collaborant en binôme pour résoudre un problème réel de gestion des unités d'enseignement.

1. Contexte :

Le TP porte sur la gestion des unités d'enseignement (UE) dans le cadre du système Licence - Master - Doctorat (LMD) au sein de la Faculté de Mathématiques, Informatique et Sciences de la Matière de l'Université 08 Mai de Guelma. Le LMD est une organisation des études comprenant trois niveaux : Licence (Bac + 3), Master (Bac + 5), et Doctorat (Bac + 8). L'objectif principal du TP est de fournir une solution informatique répondant aux besoins concrets de la faculté, de la définition des besoins à la création d'un logiciel évolutif et maintenable. Ce logiciel devra gérer les UE pour différents parcours et filières.

2. Recueil des besoins et Cahier des Charges :

L´étudiant devra effectuer des recherches sur le terrain, en interagissant avec le personnel administratif de la faculté pour identifier précisément les besoins de l'application. Il est essentiel de répondre aux attentes des utilisateurs. Des ressources en ligne peuvent également être consultées pour collecter des informations pertinentes.

https://elearning.univ-guelma.dz/course/index.php?categoryid=14

https://www.univ-guelma.dz/fr/syst%C3%A8me-lmd

3. Organisation des Séances de TP :

Séance 1 : Présentation du Sujet

- Identifier les besoins fonctionnels.

- Identifier les acteurs impliqués.

Séance 2 : Expression des Besoins sous forme de Cas d'Utilisation

- Création d'un diagramme des cas d'utilisation.

- Description préliminaire et détaillée des cas d'utilisation.

Séance 3 : Analyse

- Présentation d'un diagramme de classes d'analyse.

- Description détaillée des scénarios.

- Présentation des diagrammes de séquences.

- Présentation des diagrammes d'état.

Séance 4 : Conception

- Définition de l'architecture de l'application.

- Application des Design Patterns (patrons de conception).

Séance 5 : Implémentation

Séance 6 : Rapport Final & Validation des Applications

TP02: Étude de Cas : Système d'Évaluation des Mémoires de Fin d'Études (MFE) au Département d'Informatique

Ce TP permettra aux étudiants de mettre en pratique les principes du génie logiciel pour répondre à un besoin réel du département d'informatique de l'université 08 Mai 1945 Guelma, en améliorant l'efficacité du processus d'évaluation des mémoires de fin d'études.

1. Contexte :

Le département d'informatique de l´université 08 Mai 1945 Guelma, souhaite mettre en place un système informatisé pour évaluer les mémoires de fin d'études (MFE) de ses étudiants. Actuellement, ce processus est géré manuellement, ce qui prend beaucoup de temps et peut être sujet à des erreurs. L'objectif est de développer une application web qui permettra aux responsable de parcours d´affecter les mémoires déposés aux enseignants évaluateurs (Examinateurs, Présidents de jery) et permettra ainsi de noter et de suivre les MFE de manière efficace.

2. Objectifs :

L'objectif principal de ce TP est de concevoir et de développer un système d'évaluation des MFE qui simplifiera et automatisera le processus d'évaluation, tout en offrant une vue claire de l'état d'avancement de chaque MFE. Voici les objectifs spécifiques :

- Créer une interface conviviale pour les enseignants et les évaluateurs.

- Permettre aux enseignants de soumettre des MFE à évaluer.

- Permettre aux évaluateurs de noter les MFE de manière électronique.

- Suivre l'état d'avancement de chaque MFE, de la soumission à l'évaluation finale.

- Générer des rapports de soutenance.

3. **Étapes du TP** :

Analyse des Besoins :

- Rencontre avec les enseignants et les évaluateurs pour comprendre les exigences.

- Identification des fonctionnalités clés, y compris la soumission de MFE, l´affectation, la notation, la gestion des commentaires, etc.

- Présentation d'un diagramme de classes d'analyse.

- Description détaillée des scénarios.

- Présentation des diagrammes de séquences.

- Présentation des diagrammes d'état.

Conception

- Définition de l'architecture de l'application.

- Application des Design Patterns (patrons de conception).

Implémentation

- Implémentation de l'interface utilisateur.

- Développement des fonctionnalités de soumission de MFE, d´affectation, de notation et de suivi.

Rapport Final & Validation des Applications

Conclusion Générale

En conclusion, le génie logiciel est une discipline essentielle dans notre monde numérique actuel, car elle permet de concevoir, développer, tester et maintenir des logiciels de qualité en suivant des principes d'ingénierie rigoureux.

Ce support de cours de génie logiciel destiné aux étudiants en licence informatique couvre plusieurs chapitres clés pour comprendre les principes fondamentaux de cette discipline. En suivant ces chapitres, les étudiants pourront apprendre à modéliser en UML, spécifier les exigences fonctionnelles et concevoir la structure et le comportement d'un logiciel à travers les différentes vues proposées par UML.

Le processus unifié de développement de logiciels et les patrons de conception sont également abordés dans ce support de cours pour fournir des méthodes et des solutions éprouvées pour le développement de logiciels de qualité.

Références

Charroux, Benoît, Aomar Osmani, and Yann Thierry-Mieg. UML 2: pratique de la modélisation. Paris: Pearson Education, 2010. Disponible à la bibliothèque de la faculté «L/005.121»

Englender, Olivier, and Sophie Fernandes. Manager un projet informatique: comprendre pour faire les bons choix tout au long du projet. Editions Eyrolles, 2017. http://livre21.com/LIVREF/F6/F006080.pdf . Disponible à la bibliothèque de la faculté «L/004.1277»

Gabay, Joseph, and David Gabay. UML 2 Analyse et conception: Mise en œuvre guidée avec études de cas. Dunod, 2008. (PDF) UML 2 ANALYSE ET CONCEPTION | Ines Ghorbel - Academia.edu. Disponible à la bibliothèque de la faculté «L/004.1090»

Craig Larman, UML 2 et les design patterns, analyse et conception orientées objet et développement itératif, , 3ème édition Pearson Education, 2005. Disponible à la bibliothèque de la faculté «L/004.1010»

Jacobson, Ivar, Grady Booch, and James Rumbaugh. Le processus unifié de développement logiciel. Eyrolles, 2000. Disponible à la bibliothèque de la faculté «L/004.939»

Laurent DEBRAUWER, Design Patterns en java, les 23 modèles de conception: descriptions et solutions illustrées en UML 2 et Java, edition eni, 2013. Disponible à la bibliothèque de la faculté «L/005.093»

Laurent DEBRAUWER. Fien VAN DER HEYDE. UML 2. Initiation, exemples et exercices corrigés. 5e édition. Disponible à la bibliothèque de la faculté «L/005.018»

Naur, B. randall (eds) "Software Engineering: A Report on a Conference Sponsored by the NATO Science Committee", NATO, Bruxelles, Belgique, 1969.

Olivier Englender, Sophie Fernandes. Manager un projet informatique, Troisième édition, EYROLLES, 2012. Disponible à la bibliothèque de la faculté «L/004.1277»

Pascal André and Alan Yally, GÉNIE LOGICIEL - Développement de logiciels avec UML 2 et OCL - Cours, études de cas et exercices corrigés (Niveau B) (Technosup) Paperback – December 10, 2013. Disponible à la bibliothèque de la faculté «L/005.113»

Roques, Pascal. UML 2 par la pratique: Etudes de cas et exercices corrigés. Editions Eyrolles, 2008. http://jpaulgibson.synology.me/Teaching/TSP/Teaching-ReadingMaterial/Roques06.pdf . Disponible à la bibliothèque de la faculté «L/005.007»

Roques, Pascal, and Franck Vallée. UML 2 en action: de l'analyse des besoins à la conception. Editions Eyrolles, 2011. https://doc.lagout.org/programmation/Databases/SQL/UML.pdf. Disponible à la bibliothèque de la faculté «L/004.1122»

Soutou, Christian, and Frédéric Brouard. UML 2 pour les bases de données. Editions Eyrolles, 2012. http://livre21.com/LIVREF/F6/F006110.pdf . Disponible à la bibliothèque de la faculté «L/004.1092»

Valéry Guilhem Frémaux. Le projet informatique de A à Z, Approche pragmatique de la gestion de projet, ellipses, 2006. Disponible à la bibliothèque de la faculté «L/004.1019»

Supports de cours

AUDIBERT, L. "UML 2 [en ligne]." Disponible sur http://laurent-audibert. developpez. com/Cours-UML/

Raphael Yende, support de cours de génie logiciel. Licence. RDC (Béni), Congo Kinshasa. 2019. cel-01988734. https://hal.science/cel-01988734/document

Remy Manu, cours Langage UML Développement d'applications et informatique répartie (remy-manu.no-ip.biz)

OCL - Object Constraint Language La¨etitia Matignon Département Informatique - Polytech Lyon Universit´e Claude Bernard Lyon 1 2012 - 2013. https://perso.liris.cnrs.fr/laetitia.matignon/index/ISI32012/coursOCL.pdf