Licence en Systèmes Informatiques

Les objectifs de la licence en Systèmes Informatiques sont l'acquisition des compétences fondamentales, des méthodes théoriques et pratiques et du savoir-faire techniques représentatifs des différentes tâches de la discipline informatique. Cette formation couvre l'ensemble de la filière : fondements, architectures et matériels, conception d'interface homme-machine, technologies Web, objets, réseaux, systèmes, méthodes et technologies logicielles, applications informatiques, systèmes d'information, langages de programmation, systèmes d'exploitation, algorithmique, logique... A l'issue de la formation, construite de façon suffisamment généraliste, les étudiants peuvent intégrer des masters variés, comme ils auront la possibilité à l'insertion dans le monde du travail de tous les domaines de l'informatique, les débouchés étant nombreux et intéressants.

Unité d'Enseignement	Matière	Crédits	Coefficient	Volume horaire hebdomadaire			VHS 14 Semaines	Mode d'évaluation	
				Cours	TD	TP		Continu	Examen
UE Fondamentale 1 Crédits : 10 Coefficients : 6	Système d'exploitation 2	5	3	1h30	1h30	1h30	63h	40%	60%
	Compilation	5	3	1h30	1h30	1h30	63h	40%	60%
UE Fondamentale 2 Crédits : 10 Coefficients : 6	Génie logiciel	5	3	1h30	1h30	1h30	63h	40%	60%
	Interface homme machine	5	3	1h30	1h30	1h30	63h	40%	60%
UE Méthodologie Crédits : 8 Coefficients : 4	Programmation linéaire	4	2	1h30	1h30		42h	40%	60%
	Probabilités et statistique	4	2	1h30	1h30		42h	40%	60%
UE Transversale Crédits : 2 Coefficients : 1	Economie numérique et veille stratégique	2	1		1h30		21h	100%	
Total Semestre 5		30	17	9h	10h30	6h	357h		

Matière : Génie logiciel

1. Chapitre 1: Introduction
   1.1. Définitions et objectifs
   1.2. Principes du Génie Logiciel
   1.3. Qualités attendues d'un logiciel
   1.4. Cycle de vie d'un logiciel
   1.5. Modèles de cycle de vie d'un logiciel
2. Chapitre 2: Modélisation avec UML
   2.1. Introduction : Modélisation, Modèle, Modélisation Orientée Objet, UML en application.
   2.2. Eléments et mécanismes généraux
   2.3. Les diagrammes UML
   2.4. Paquetages
3. Chapitre 3: Diagramme UML de cas d'utilisation : vue fonctionnelle
   3.1. Intérêt et définition, Notation
4. Chapitre 4: Diagrammes UML de classes et d'objets : vue statique
   4.1. Diagramme de classes
   4.2. Diagramme d'objets
5. Chapitre 5: Diagrammes UML : vue dynamique
   5.1. Diagramme d'interaction (Séquence et collaboration)
   5.2. Diagramme d'activités
   5.3. Diagramme d'état/transitions
6. Chapitre 6: Autres notions et diagrammes UML
   6.1. Composants, déploiement, structures composite.
   6.2. Mécanismes d'extension : langage OCL + les profils.
7. Chapitre 7: Introduction aux méthodes de développement : (RUP, XP)
8. Chapitre 8: Patrons de conception et leur place au sein du processus de développement

Matière : Génie Logiciel

Table des matières

La partie : Cours

1. Chapitre 1: Introduction

Les logiciels et les solutions informatiques spécialisées sont de plus en plus utilisés dans la plupart des aspects de la vie quotidienne (systèmes d'exploitation, outils, logiciels d'entreprises, logiciel scientifiques, systèmes embarqués, etc.). La plupart des systèmes de communication, de transport, de production et de contrôle utilisent des solutions logicielles intégrées afin de renforcer leur rentabilité, leur flexibilité et leur qualité.

Le développement de ces logiciels, souvent complexes et de grande taille, présente de nombreux défis tels que la maîtrise des coûts et des délais de réalisation, l'évolution des besoins des utilisateurs, le développement collectif et la difficulté de communication entre les différents intervenants, l'évolution du matériel, la diversification des architectures et des environnements, etc.

Les défis ont mis en évidence, depuis les années 60, la crise du logiciel qui se caractérise par le fait que les projets de développement de solutions logicielles n'ont pas toujours été réalisés avec succès. En fait, la plupart de ces projets ont été abandonnés, annulés ou refusés, à cause de l'absence de maîtrise de ces projets, au niveau des coûts, des délais, et de la mauvaise qualité des produits logiciels développés.

Ce constat d'échec a conduit au développement et à l'adoption de nouveaux procédés, outils et formalismes permettant de développer efficacement des solutions logicielles fonctionnelles et de qualité en dépit de leur complexité et de tout autre défi. Le domaine d'étude de ces procédés, outils et formalismes est appelé génie logiciel.

1.1. Définitions et objectifs

Le génie logiciel est un domaine des sciences de l'ingénieur dont la finalité est la conception, la fabrication et la maintenance de systèmes logiciels complexes, sûrs et de qualité. C'est un ensemble de méthodes, techniques et outils pour la production et la maintenance de composants logiciels corrects et de qualité.

Contrairement à la programmation individuelle (production individuelle d'un système simple), le génie logiciel soutient une production collective d'un système complexe caractérisée par un ensemble de documents de conception, de programmes et de jeux de tests avec souvent de multiples versions2. Cet appui est traduit par la définition et le développement de plusieurs concepts du génie logiciel répondant à différents types de besoins tels que les processus de développement ou modèles de cycles de vie de logiciels (besoins de gestion de ressources, coûts et délais), les méthodes d'analyse et de conception (besoins de gestion du développement), les langages de modélisation (besoins de communication), les design patterns et les frameworks (besoins de réutilisation), etc.

1.2. Principes du Génie Logiciel

Le génie logiciel se préoccupe des procédés de fabrication de logiciels de façon à garantir que les quatre critères suivants soient satisfaits :

- Le système développé doit fournir les fonctionnalités attendues
- Les coûts de développement doivent rester dans les limites prévues au départ
- Les délais doivent rester dans les limites prévues au départ
- Le système développé doit garantir les qualités requises par le contrat de service

1.3. Qualités attendues d'un logiciel

La qualité désigne l'appréciation générale d'un logiciel selon des critères comme :

- **fiabilité** : capacité d'un logiciel à assurer de manière continue le service attendu
- **correction (validité)** : aptitude d'un logiciel à réaliser exactement les tâches telles qu'elles ont été définies par sa spécification
- **robustesse** : aptitude d'un logiciel à fonctionner même dans des conditions anormales
- **extensibilité**: facilité d'adaptation d'un logiciel aux changements de spécification
- **réutilisabilité**: aptitude d'un logiciel à être réutilisé en tout ou partie
- **compatibilité** : aptitude des logiciels à être combinés les uns aux autres
- **efficacité**: capacité d'un logiciel à optimiser l'utilisation de ressources (mémoire, processeur, bande passante, etc.)
- **portabilité**: facilité à être porté sur différents environnements matériels et/ou logiciels
- **traçabilité**: capacité à identifier et/ou suivre un élément du cahier des charges lié à un composant logiciel
- **autres critères**: simplicité, intégrité, réparabilité, vérifiabilité, etc.

Ces qualités sont parfois contradictoires et doivent être pondérées selon le type du logiciel (critique/grand public, systèmes sur mesure/produits logiciels de grande diffusion, etc.).

1.4. Cycle de vie d'un logiciel

1.4.1. Définition

Le cycle de vie du logiciel est un ensemble cohérent d'**activités** pour spécifier, concevoir, implémenter et tester des systèmes logiciels. A chaque activité sont associés différents **livrables** qui se présentent sous forme de **documents** tels que le plan du projet, les plans de tests, les modèles d'analyse, les modèles de conception, le code source, les rapports de tests et les manuels d'utilisation.

1.4.2. Activités

Le cycle de vie de logiciels définit de nombreuses activités qui peuvent être de différentes natures : activités de définition (planification, Spécification), activités de production (conception, implémentation et vérification), activités de livraison (installation et déploiement, conversion de données et formation) et activités de maintenance.

1. Phase de définition

a. Planification du projet : il s'agit d'une étude préliminaire pour déterminer les possibilités de réalisation du projet. Les sous-activités principales de cette activité sont les suivantes :

- définition globale du problème
- étude de la faisabilité et analyse du marché
- évaluation des stratégies possibles
- évaluation des ressources, coûts et délais
- assurance qualité
- élaboration du calendrier du projet

Le document résultat de l'activité de planification est le rapport de planification.

b. Analyse des besoins : il s'agit de déterminer ce qu'il faut faire. Les sous-activités de base de l'analyse des besoins sont les suivantes :

- recueil d'informations
- déterminer les exigences fonctionnelles
- déterminer les exigences non-fonctionnelles (contraintes)
- spécification du système (modèles d'analyse)
- construction de prototypes (pour élaborer la spécification)

Les documents associés à cette activité sont le cahier des charges, les modèles d'analyse, le plan de tests et le prototype.

2. Phase de développement

a. Conception : l'objectif de l'activité est de déterminer comment procéder pour réaliser ces besoins. Les sous-activités essentielles étant :

- conception architecturale du système : décomposition et organisation du système en modules et définition des interfaces entre modules
- conception détaillée de modules : description de la manière dont les services et les fonctions sont réalisés

Les documents produits de l'activité de conception sont les modèles de conception, le prototype, le plan de tests global et le plan de tests par module.

b. Implémentation : Il s'agit de construire les composantes logicielles par mise en œuvre de la conception dans un langage de programmation ou en utilisant des outils de développement.

Les documents produits de cette activité sont les dossiers de programmation, le code source commenté et le prototype.

c. Vérification : déterminer si le produit réalise correctement le travail attendu par évaluation de la solution en fonction de la spécification (test) :

- test **unitaire** : vérifier séparément le module développé
- test **d'intégration** : tester le produit durant l'intégration d'un module
- test du **système** : évaluer la conformité du produit logiciel par rapport aux exigences spécifiées
- test **d'acceptation** : évaluer la conformité du produit logiciel par rapport aux spécifications en présence effective des différents acteurs du projet

Le document associé à cette activité est le rapport de tests.

3. Phase de Livraison

a. Installation et déploiement : mettre le produit logiciel en fonctionnement opérationnel dans son environnement (chez les utilisateurs)

b. Conversion des données: transférer les données de l'ancien système et les données manquantes dans le nouveau système c. Formation : former les utilisateurs à utiliser le logiciel

4. Phase de maintenance

a. Maintenance : mettre-à-jour le logiciel pour garantir une utilisation efficace continue et faciliter les opérations de maintenance à venir :

- maintenance **corrective** : corriger les erreurs
- maintenance **adaptative** : s'adapter à des changements d'environnement
- maintenance **perfective** : améliorations

1.4.3. Documents

Les livrables des différentes activités sont représentés par plusieurs types de documents, dont les plus importants sont :

- **Plan du projet** : décrit l'ordre des tâches et estime les besoins en matière de délais et d'efforts
- **Plan de tests**: décrit comment le produit serait testé afin de garantir un comportement correct
- **Spécification des besoins**: décrit ce que doit faire le logiciel
- **Modèles d'analyse**: décrivent la solution logicielle indépendamment des choix techniques ou organisationnels
- **Modèles de conception**: décrivent la solution logicielle finale retenue
- **Exécutables** : désignent le ou les différents composants exécutables du produit final
- **Code source** : comprend la totalité du code source du produit final
- **Rapport de tests**: décrit quels sont les tests effectués et quel était le comportement du système enregistré

1.5. Modèles de cycle de vie d'un logiciel

Un modèle de cycles de vie ou processus de développement est une description abstraite et personnalisée de l'organisation des activités de développement d'un logiciel. Il présente alors la description détaillée du cycle de vie d'une perspective particulière tout en négligeant les parties inutiles.

1.5.1. Modèle en cascade

C'est un modèle linéaire axé sur la documentation dans lequel le déroulement des activités est réalisé de manière séquentielle d'une activité à l'autre (figure 1). Il s'agit d'un modèle simple et efficace pour les systèmes complexes si tous les besoins sont déterminés à priori et ne changent pas au cours du développement.

D'autre part, ce modèle impose des écarts de temps considérables entre l'expression des besoins et l'implémentation et manipule des délivrables volumineux à chaque activité.

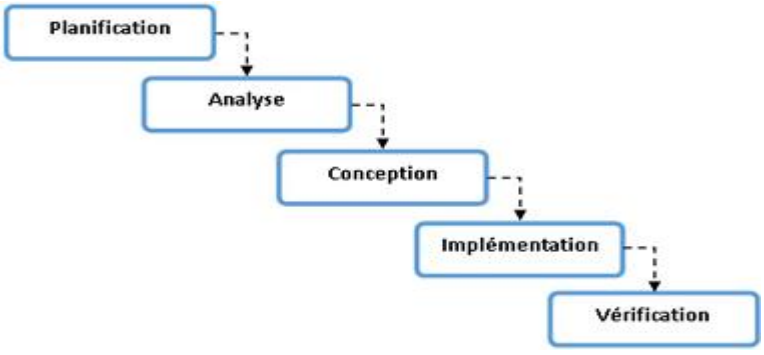


Figure 1 : Modèle en cascade

Il existe plusieurs versions et variantes du modèle linéaire qui mettent l'accent sur certaines activités plutôt que d'autres selon les besoins et le contexte. Les deux variantes les plus importantes du modèle séquentiel sont le modèle en V et le modèle incrémental.

1.5.2. Modèle en V

C'est une variante du modèle séquentiel axée sur la vérification où chaque étape de développement (analyse ou conception) a un niveau de tests qui lui est associé (figure 2). Cette liaison activité-test permet de définir des tests pertinents et de haute qualité.

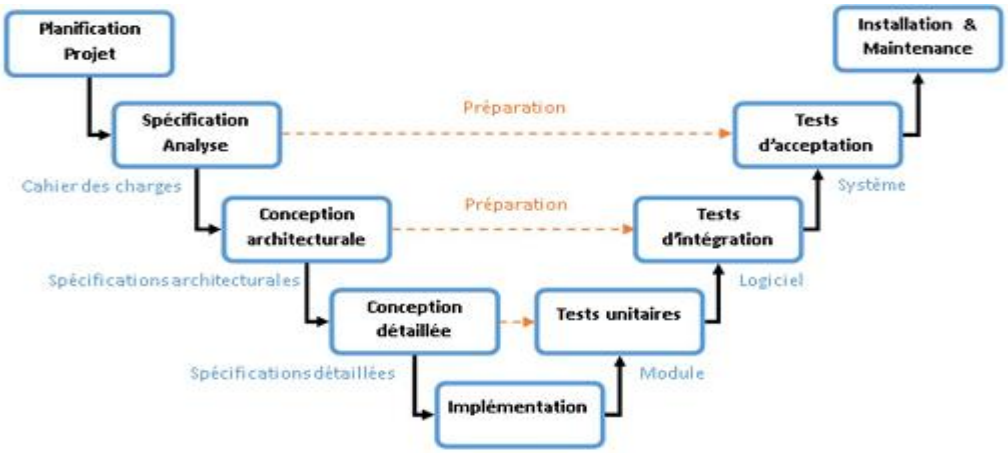


Figure 2 : Modèle en V

1.5.3. Modèle incrémental

Le modèle incrémental est une autre variante du modèle en cascade axée sur la décomposition du système global en sous-projets indépendants (conception générale) et l'itération, comme le montre la figure 3, du développement (conception détaillée et implémentation pour chaque sous-projet) et de l'intégration des incréments au système final. Le modèle offre la possibilité de développement parallèle d'incréments, réduisant ainsi le temps nécessaire pour la livraison du produit final.

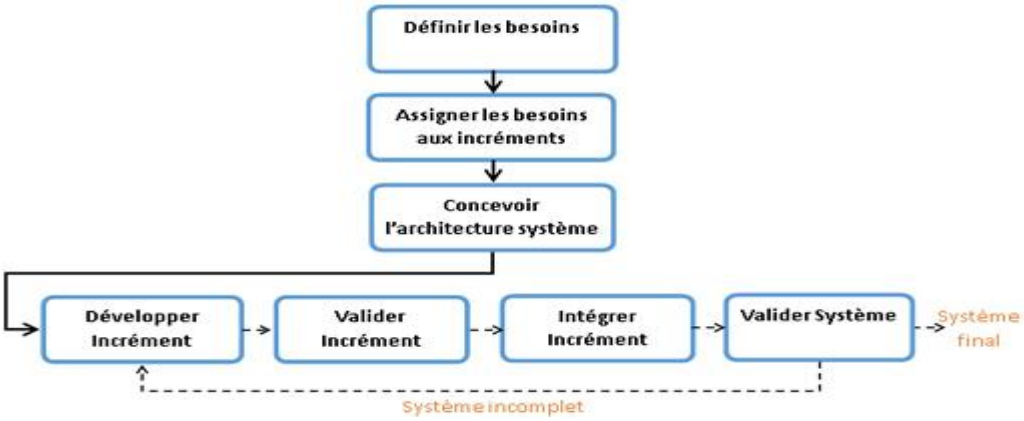


Figure 3 : Modèle incrémental

L'approche incrémentale souffre en général, en plus des problèmes du modèle en cascade, du sérieux problème d'intégration dans le cas où les sous-projets ne sont pas complètement indépendants.

1.5.4. Modèle de prototypage

Le modèle de prototypage est un modèle de développement itératif3 dans lequel les activités d'analyse, de conception et d'implémentation sont réalisées de façon concurrente. L'objectif principal est de livrer rapidement un minimum de fonctions stables (prototype jetable ou évolutif) qui sera fourni aux clients pour éventuel feedback. Sur la base de ce retour d'information, les besoins sont raffinés et le développement de la prochaine version du prototype se poursuit en procédant de la même manière. Le modèle de prototypage est, alors, adapté pour les projets où les besoins ne sont pas clairement définis ou qui sont susceptibles de changer avec le temps.

1.5.5. Modèle en spirale

C'est un modèle axé sur de la gestion des risques et implémente les éléments des modèles en cascade, incrémental et prototypage. L'image du modèle est une spirale qui commence au milieu et qui réitère continuellement les tâches de base (figure 4).

NB : le développement itératif décompose les besoins du système global en versions qui sont développées séquentiellement en commençant par les besoins les plus importants et en passant par les mêmes activités de développement.

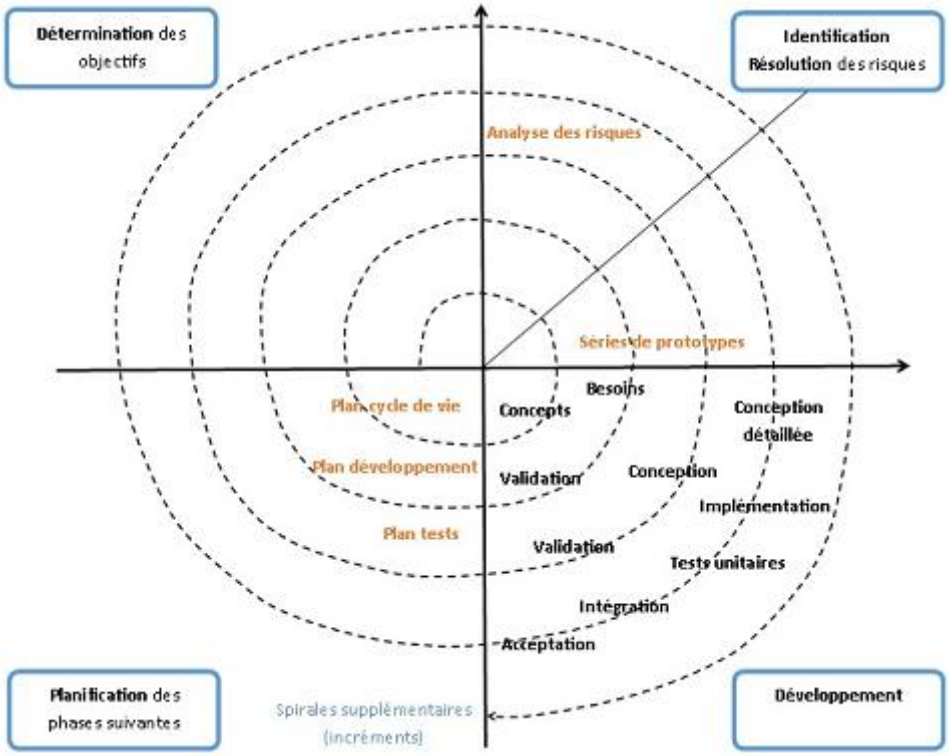


Figure 4 : Modèle en spirale

Les principaux risques (et leurs solutions) sont les suivants :

- **défaillance de personnel** : embauches de haut niveau, formation mutuelle, adéquation profil/fonction, etc.
- **calendrier et budgets irréalistes** : estimation détaillée, développement incrémental, réutilisation, adaptation des besoins, etc.
- **développement de fonctions inappropriées** : revues d'utilisateurs, manuel d'utilisation précoce, etc.

- **développement d'interfaces utilisateurs inappropriées** : maquettage, analyse des tâches, etc.
- **produit "plaqué or"** : analyse des coûts/bénéfices, conception tenant compte des coûts, etc. volatilité des besoins : développement incrémental de la partie la plus stable d'abord, masquage d'information, etc.
- problèmes de performances : simulations, modélisations, essais et mesures, maquettage, etc.
- **exigences démesurées par rapport à la technologie** : analyses techniques de faisabilité, maquettage, etc.
- **tâches ou composants externes défaillants** : audit des sous-traitants, contrats, revues, essais et mesures, etc.

1.5.6. Processus unifié et modèles dérivés

Le processus unifié est un processus d'ingénierie logicielle générique qui regroupe les caractéristiques communes et essentielles des différents processus de développement objet :

- itératif et incrémental
- modélisation visuelle avec **UML (Unified Modeling Language)**
- piloté par les cas d'utilisation
- centré sur l'architecture
- guidé par les patrons de conception (**Design Patterns**)

Ces caractéristiques sont, toutefois, génériques ; le processus unifié ne peut pas être utilisé directement et nécessite une spécialisation qui tient compte des facteurs techniques et organisationnels du domaine. Ses principaux modèles dérivés sont : **RUP** (Rational Unified Process), **ESA** (Extreme System Analysis), **EUP** (Enterprise Unified Process), **2TUP** (2 Tracks Unified Process) et **Catalysis**.

1.5.7. Développement rapide d'applications et Modèles agiles

L'objectif de ces catégories de modèles est d'éviter les écarts importants entre les résultats obtenus et l'expression des besoins initiaux et avoir ainsi une livraison dès que possible du produit final. Les modèles de ces catégories choisissent les solutions les plus simples, impliquent au maximum les clients et favorisent les interactions et les applications fonctionnelles plutôt que les processus et les documents exhaustifs.

Le développement rapide d'applications (RAD) priorise les itérations rapides et les versions de prototypes en utilisant des techniques spéciales et des outils d'aide tels que les outils CASE (Computer-Aided Software Engineering), la génération automatique du code, la programmation visuelle, etc. L'utilisation de ces techniques et outils permet d'accélérer les phases d'analyse, de conception et d'implémentation. Ce type de développement rapide peut être réalisé de différentes manières : développement itératif, prototype évolutif ou prototype jetable4.

Le développement agile est un ensemble de modèles de développement plus récents, centrés sur la programmation, qui se caractérisent par des cycles de développement courts et itératifs destinés aux projets d'applications simples. Chaque itération comprend les activités essentielles d'un projet (planification, analyse des besoins, conception, codage, tests et documentation) mais avec élimination d'une grande partie de la modélisation et de la documentation. Des exemples de modèles agiles peuvent comprendre les modèles de déveoppement XP (eXtreme Programming), ASD (Adaptative Software Development), FDD (Future Driven Development), etc.

NB : prototypes considérés en tant que nouveau système (prototypage de système) ou destinés uniquement à explorer des alternatives de conception (prototypes jetables).

2. Chapitre 2: Modélisation avec UML

2.1. Introduction

L'un des principaux défis auxquels sont confrontés les projets de développement d'applications logicielles est de déterminer avec précision les exigences exactes du futur système dans chacune des étapes de développement. La manière de communication avec le client, dans l'étape d'expression des besoins, et entre développeurs (analyste, concepteur, programmeur, etc.), dans le reste des étapes, est cruciale pour la réussite du projet. L'utilisation du langage naturel dans ce contexte est fortement déconseillée car il est imprécis et ambiguë ; des malentendus peuvent facilement survenir si des personnes, de milieux différents ou de spécialités différentes, utiliseront le langage naturel comme moyen de définition de leurs besoins (spécifications incomplètes, spécifications surchargées, mauvaise compréhension, etc.).

Il est, donc, nécessaire de pouvoir créer un modèle du système à développer. Un modèle qui met en évidence les aspects importants de ce système sous une forme de notation claire aussi simple que possible, et qui ne tient pas compte des détails non pertinents qui rendent la description plus compliquée que nécessaire. Des modèles séparés peuvent être construits, à différentes étapes et pour différents aspects, afin d'éviter de présenter trop d'informations à la fois et faciliter ainsi la lecture, l'interprétation et la mise en œuvre de ces modèles.

Pour répondre à ce besoin, différents langages de modélisation ont été développés afin de permettre une description structurée de systèmes réels à base de règles clairement définies. Ces langages peuvent être textuels (logiques, langages de programmation, etc.) ou graphiques (modèles entité-association, graphes de flux, etc.). Ils peuvent être complets, et couvrent ainsi, tous les aspects du système développé, ou destinés à la modélisation d'un aspect particulier de l'application (par exemple, l'utilisation des modèles d'automates pour des besoins de vérification). Ils peuvent être, également, spécifiques à un domaine particulier (langage de modélisation des applications web) ou à objectifs d'utilisation générale. UML est un langage de modélisation graphique, complet et à objectifs d'utilisation générale. Il est très adapté pour le développement des applications logicielles à base de concepts objet.

2.2. Modèles

Les modèles présentent un moyen pour décrire les systèmes1 de manière efficace et détaillée. Ils permettent de limiter la représentation du système à l'essentiel afin de réduire la complexité du système à des aspects maîtrisables. Les caractéristiques principales qui permettent de déterminer la qualité des modèles sont les suivantes :

- **Abstraction** : les détails qui ne sont pas pertinents dans un contexte spécifique sont cachés ou supprimés ce qui permet une meilleure concentration sur l'essentiel
- **Compréhension** : les détails pertinents sont présentés le plus intuitivement possible grâce à l'expressivité du langage de modélisation

NB : un système est un ensemble intégré constitué d'éléments qui sont liés les uns aux autres et qui s'influencent mutuellement de telle sorte qu'ils peuvent être perçus comme une unité unique, à base de tâches ou d'objectifs (voiture, université, logiciel, etc.).

- **Précision** : un modèle doit mettre en évidence les propriétés pertinentes qui reflètent le plus fidèlement possible la réalité ;
- **Prédictivité** : un modèle doit permettre de prévoir (par simulation ou analyse) des propriétés évidentes du système modélisé ;

- **Rentabilité** : il est moins coûteux de créer le modèle que de créer le système modélisé Généralement, un système est décrit par plusieurs vues qui, ensemble, fournissent une représentation globale unifiée. Par exemple, une vue peut décrire les objets du système et leurs relations, une autre vue peut décrire les interactions entre ces objets et une troisième vue pour présenter la dynamique intra ou inter-objets.

2.3. Concepts de l'orientation objet

Afin de pouvoir traiter la modélisation objet, il est nécessaire de clarifier certains concepts essentiels de l'orientation objet. Cette approche est basée sur la notion d'objets qui sont des éléments du système dont les données et les opérations sont décrites. Les objets peuvent interagir et communiquer entre eux pour réaliser les fonctions du système. Les concepts génériques de l'approche orientée objet peuvent être résumés dans les points clés suivants :

NB : la capacité à présenter un contenu complexe avec le moins de concepts possible ce qui réduit l'effort intellectuel nécessaire pour comprendre le contenu représenté.

2.3.1. Objet

Un objet définit une représentation d'un concept réel ou virtuel qui encapsule une partie des connaissances du monde dans lequel il évolue. Il est caractérisé par une identité et possède un état et un comportement en ne laissant visible que son interface (les opérations que l'on peut appliquer sur cet objet) :

- **Identité** : donnée implicite qui permet de distinguer l'objet de manière non ambiguë indépendamment de son état
- **Etat** : l'ensemble des valeurs des attributs de l'objet à un moment donné (et qui évolue au cours du temps)
- **Comportement** : compétences ou services d'un objet décrivant les actions et les réactions de cet objet. Chaque comportement élémentaire d'un objet est appelé opération et est déclenché suite à une stimulation externe (message envoyé par un autre objet)

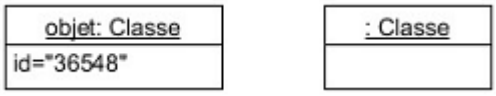


Figure 1 : Objets

2.3.2. Classe

Une classe décrit les attributs et le comportement d'un ensemble d'objets de façon abstraite et donc de regrouper les caractéristiques communes de ces objets.

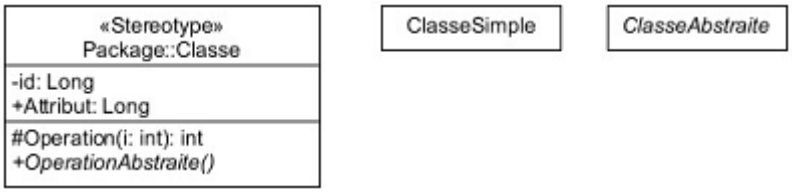


Figure 2 : Classes

La démarche d'abstraction se définit par rapport à un point de vue (critères pertinents du domaine considéré) et procède à l'identification des caractéristiques communes à un ensemble d'objets, puis à la définition de ces caractéristiques dans une classe. La classe représente, ainsi, le domaine de définition de cet ensemble d'objets.

Les attributs correspondent aux propriétés statiques de la classe. Par exemple, les étudiants ont un nom, une adresse et un numéro d'inscription, les cours ont un identifiant, un titre et une description, etc.

En phase d'analyse, il est recommandé de ne pas confondre entre objet et attribut ; si l'on ne peut demander à un élément que sa valeur, il s'agit d'un simple attribut. Si plusieurs questions s'y appliquent, il s'agit plutôt d'un objet qui possède lui-même des attributs, des opérations ou des liens avec d'autres objets.

Une classe définit également un ensemble d'opérations qui peuvent être appliquées à ses instances (constructeurs, sélecteurs, itérateurs, etc.). Par exemple, il est possible d'enregistrer un nouvel étudiant, de consulter la description d'un cours, de réserver une salle à une date, d'inscrire un étudiant à un cours, etc.

En phase d'analyse et/ou de conception, les opérations sont identifiées après étude des scénarios - interactions entre objets - qui décrivent les différentes fonctionnalités du système.

2.3.3. Encapsulation

L'encapsulation permet de protéger, via une interface définie de manière unique, l'état interne d'un objet contre les accès non autorisés. Différents niveaux de visibilité des interfaces permettent de définir différentes autorisations d'accès. Java, par exemple, adopte les indicateurs de visibilité explicites public, privé et protégé, qui permettent respectivement l'accès pour tous, uniquement dans l'objet, et uniquement pour les membres de la même classe, de ses sous-classes, et du même package.

2.3.4. Message

Les objets collaborent pour réaliser les fonctions de l'application. Le comportement global de l'application repose sur la communication entre les objets. Cette communication est réalisée par envoi et réception de messages qui représentent des demandes d'exécution d'opérations. L'opération n'est exécutée que si l'objet expéditeur est autorisé à faire appel à cette opération - condition de visibilité - et qu'une implémentation appropriée de l'opération est disponible. Le concept de surcharge de définition est supporté dans la plupart des langages de programmation et de modélisation objet. Ce concept permet de définir des comportements différents d'une opération selon les types de ses paramètres.

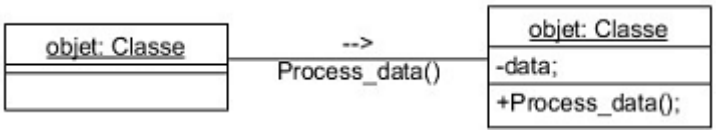


Figure 3 : Communication entre objets

2.3.5. Héritage

L'héritage définit une classification des objets au sein d'une arborescence de classes permettant de gérer leur complexité par réutilisation des caractéristiques héritées. C'est une relation asymétrique non réflexive. Suivant le besoin, la définition de la relation d'héritage peut prendre l'une des deux formes : généralisation ou spécialisation.

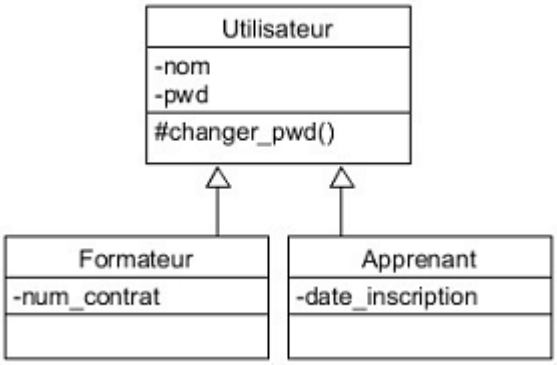


Figure 4 : Héritage

La généralisation est employée une fois que les classes du domaine sont identifiées. Elle consiste alors à factoriser les informations communes entre classes dans une nouvelle classe, super-classe de classes déjà existantes.

La spécialisation permet, toutefois, de capturer des caractéristiques (d'un sous-ensemble d'objets d'une classe) non couvertes par les classes déjà identifiées. Les nouvelles caractéristiques sont représentées par une nouvelle classe, sous-classe de classes déjà existantes. La sous-classe dérivée hérite de tous les attributs et opérations visibles (spécification et implémentation) et des associations de la super-classe. La sous-classe peut, en plus, définir de nouveaux attributs et/ou opérations, remplacer l'implémentation des opérations héritées, ajouter son propre code aux opérations héritées ou avoir de nouvelles associations.

La définition des relations d'héritage doit répondre à un critère de classification pertinent (différence de structure) et non pas sur la base de changement de valeurs particulières d'attributs d'une même classe (voiture blanche, voiture noire, etc.) ou de changement de rôles de classes par rapport à une association (étudiant inscrit et étudiant délégué de classe, etc.).

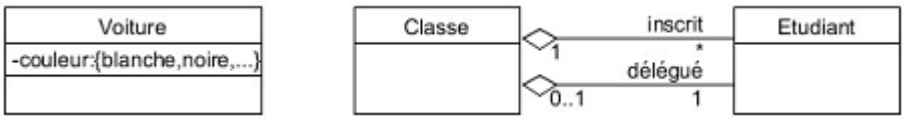


Figure 5 : Critères de classification non pertinents (valeurs, rôles, etc.) – Résolu

De plus, l'héritage n'est pas adapté pour représenter les métamorphoses. En fait, l'instanciation d'objets introduit un couplage statique très fort et non mutable entre classe et instance; une instance ne peut jamais changer sa dépendance à sa classe de définition.

Dans l'exemple de la figure, l'employé stagiaire est titularisé après une certaine période de stage ; c'est sa situation qui change et non pas l'employé lui-même.

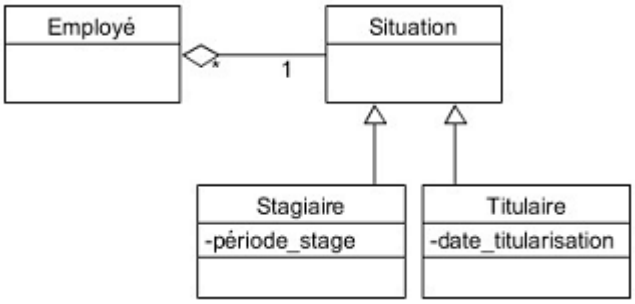


Figure 6 : Critère de classification non pertinent (métamorphose) – Résolu

2.3.6. Polymorphisme

Le polymorphisme d'attributs implique qu'un attribut peut avoir des références à des objets de différentes classes sous-classes du type de cet attribut polymorphe.

Le polymorphisme d'opérations offre la possibilité d'exécuter des opérations de différentes implémentations sur des objets de différentes classes (appartenant à une même hiérarchie de classes) en réponse à un même message (spécification donnée au niveau de la super-classe de cette hiérarchie).

2.3.7. Associations entre classes

Une association entre classes représente une abstraction des liens - d'une même sémantique - qui existent entre objets de ces classes. L'association peut prendre différentes formes : association simple uni ou bidirectionnelle, agrégation ou composition.

- **Association simple** : connexion sémantique uni ou bidirectionnelle entre objets. Par exemple, un étudiant suit un ou plusieurs cours.

Figure 7 : Association simple

- **Agrégation** : une association qui exprime un rapport maître-esclave entre objets (ensemble-élément, tout-partie, composé-composant, etc.). Par exemple, un comité est constitué de plusieurs enseignants et un enseignant peut être membre de plusieurs comités.

NB : en général, le polymorphisme est la capacité d'un élément à adopter différentes formes.

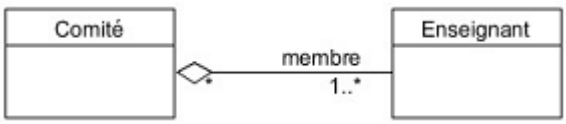


Figure 8 : Agrégation

- **Composition** : une forme d'agrégation avec couplage plus important ; les éléments agrégés ne sont pas partageables et la destruction de l'élément agrégat engage celle des

éléments agrégés. Par exemple, un étudiant possède un compte Moodle et le compte est associé à un seul étudiant.

Figure 9 : Composition

En phase d'analyse, la classe peut exister indépendamment des autres éléments du système, alors que l'existence de l'association est conditionnée par celle des éléments qui participent à cette association. La même remarque s'applique aux concepts liés aux associations tels que les rôles d'objets et les attributs des classes-associations.

Par exemple, dans la règle "une classe possède un étudiant délégué", les concepts possède (association entre classe et étudiant) et délégué (rôle de l'étudiant dans cette association) ne peuvent pas exister sans qu'il y ait étudiant et classe. De même, selon la règle "un étudiant obtient une note dans une matière", la note ne peut pas exister s'il n'y a pas étudiant et cours. Les concepts étudiant, classe et cours peuvent exister seuls et constituent, donc, des classes dans le modèle.

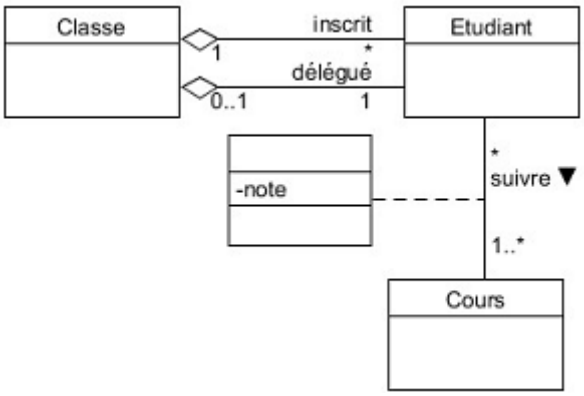


Figure 10 : Association, Classe-association et Rôle

2.4. UML

Le langage UML constitue une référence dans le domaine de la modélisation objet adopté par l'OMG (Object Management Group), la structure la plus importante de normalisation des concepts de développement orienté objet. Le langage fournit un support considérable et extensible pour les différentes activités de développement (analyse, conception d'architecture, conception de modules, implémentation, etc.) et les différentes classes de systèmes à développer (systèmes d'information, applications web, systèmes temps réel, etc.).

2.4.1. Définition

Le langage UML est un standard de modélisation graphique semi-formelle qui regroupe les meilleures pratiques de modélisation objet. Grâce à une notation très riche et suffisamment expressive, UML permet la modélisation de la structure et du comportement de systèmes logiciels indépendamment des méthodes de développement et des langages de programmation. En général, UML permet de spécifier, construire, visualiser et décrire les artefacts des systèmes logiciels :

- **Spécifier et Documenter** : les éléments de modélisation UML possèdent une syntaxe et une sémantique bien définies ce qui permet de produire une modélisation précise, non ambiguë et complète
- **Construire** : le passage entre modèles UML et implémentation peut être réalisé manuellement ou de manière semi-automatique grâce aux correspondances déjà mises en place entre constituants UML et langages de programmation objet
- **Visualiser** : UML propose un ensemble exhaustif de diagrammes couvrant les différentes vues d'un système logiciel (fonctionnalités, structure, dynamique, etc.) et les différents niveaux d'abstraction (modèles d'analyse, modèles d'architecture, modèles de conception, etc.)

2.4.2. Historique UML

L'introduction de concepts objet au début des années 1960 [SIMULA] a marqué le début d'une révolution dans le développement de systèmes logiciels. Les décennies suivantes ont connu l'apparition de plusieurs langages de programmation basés sur le paradigme objet tels que C++, Eiffel, Smalltalk, Java, C#, etc. En parallèle, beaucoup de méthodes d'analyse et de conception objet, comme OMT (Object Modeling Technique), BOOCH (auteur Grady Booch) et OOSE (Object Oriented Software Engineering), se sont imposées grâce à leurs démarches et notations incontournables dans le domaine de développement de systèmes logiciels.

La plupart de ces méthodes objet étaient liées uniquement par un accord sur les concepts de base de l'objet (objet, classe, héritage, etc.). Toutefois, chacune de ces méthodes proposait sa propre notation et aucune méthode ne pouvait prétendre couvrir tous les besoins, ni modéliser correctement les différentes vues d'une application logicielle.

En 1995, des efforts d'unification des méthodes objet, pratiques industrielles et notations (menés principalement par Ivar Jacobson (OOSE), Grady Booch (BOOCH) et James Rumbaugh (OMT)) ont conduit à la proposition de la méthode unifiée (Unified Method 1995). Les résultats d'unification n'ont pas pu aboutir à cause de deux problèmes majeurs : (i) la dissemblance des styles de conception des développeurs et (ii) la diversité des classes de systèmes à développer. En fait, les méthodes objet partagent les concepts objets et non pas les démarches et il serait insensé d'imposer une approche unifiée pour des styles de conception très variés et des classes de systèmes fortement différentes. Par la suite, les efforts ont été redirigés vers l'unification des notations manipulées par les méthodes. En 1996, l'OMG (Object Management Group), a lancé un premier appel pour la spécification d'une norme de modélisation uniforme.

La première version d'UML adopté par l'OMG (Object Management Group) était la version UML 1.1 en novembre 97. De nombreuses versions, ensuite, ont été adoptées par l'OMG dont la plus importante est UML 2.0 (juillet 2005) qui a connu des révisions majeures du langage avec la définition de nouveaux types de diagrammes. La version actuelle est UML 2.5.1 (depuis décembre 2017) et les travaux d'amélioration du langage continuent toujours.

A partir de sa version 1.4, UML a été publié en tant que norme ISO approuvée. Cette norme est révisée périodiquement pour couvrir la dernière révision UML [ISO/IEC 19505- 2:2012].

2.4.3. Diagrammes UML

En UML, le modèle du système est représenté graphiquement sous forme de diagrammes. Chaque diagramme fournit une vue d'une partie du système décrit par ce modèle. Certains diagrammes présentent quelle fonctionnalité du système est utilisée par quel utilisateur et d'autres décrivent les composants du système et leur déploiement. Il existe également des diagrammes qui

représentent l'aspect statique du système et d'autres qui décrivent sa dynamique. Dans la version actuelle, UML propose 14 diagrammes qui peuvent être répartis en deux grandes catégories ; des diagrammes de structure et des diagrammes de comportement (figure 11).

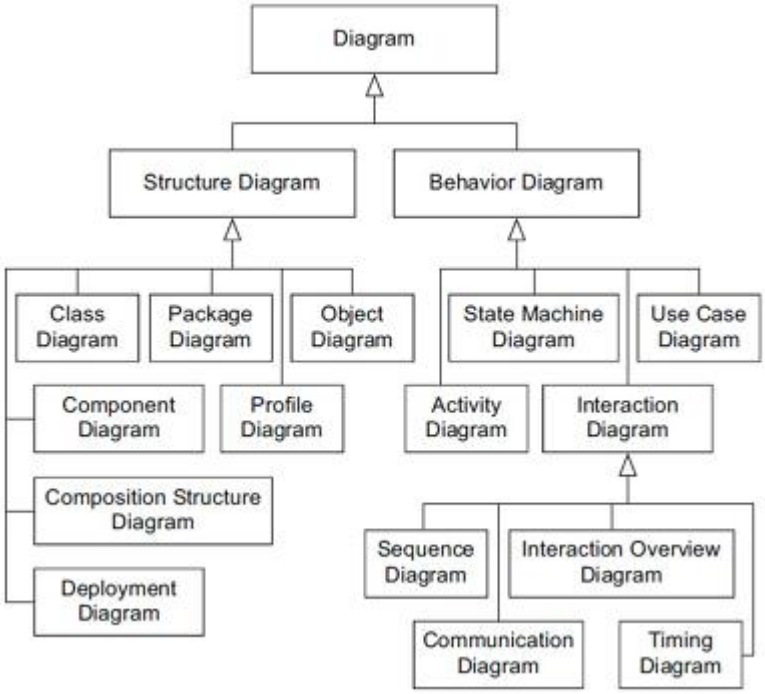


Figure 11 : Diagrammes UML (taxonomie structure-comportement)

2.4.4. Utilisation du langage

UML est une notation et ne définit pas de démarche et n'impose pas de processus de développement. Cependant, le langage est facilement intégrable à des méthodes d'analyse et de conception objet ou dans le cadre d'un processus de développement mettant en œuvre les caractéristiques essentielles du processus unifié. La figure 12 présente l'une des organisations communes des différents diagrammes UML en vues (4+1 vues - UML 2.0).

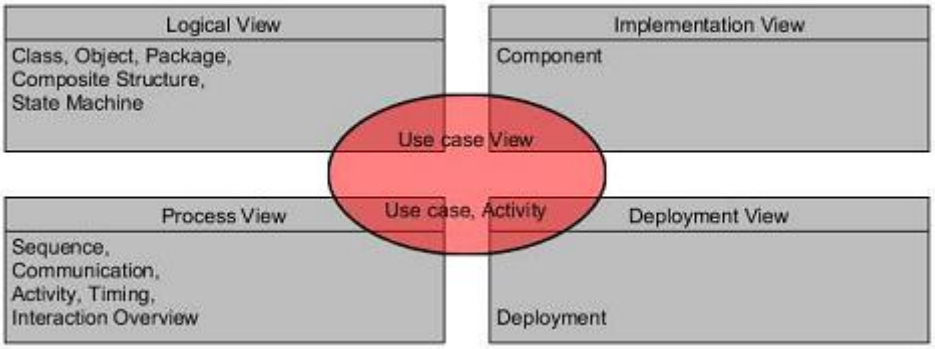


Figure 12 : Organisation des diagrammes UML 2.0 en 4+1 vues

3. Chapitre 3: Modélisation des fonctionnalités-Diagramme UML de cas d'utilisation

3.1. Introduction

Les fonctionnalités du système à développer désignent les exigences du client et ses attentes de ce système. Lors de l'analyse des besoins, il est essentiel de pouvoir repérer et représenter soigneusement ces fonctionnalités ; si des fonctionnalités sont oubliées ou spécifiées de manière imprécise ou incorrecte, les conséquences peuvent être sérieuses, à savoir les coûts de développement et de maintenance augmentent, les utilisateurs sont insatisfaits, etc.

La modélisation de ces fonctionnalités est un concept clé du développement objet qui est exploité tout au long des activités d'analyse et de conception. Le langage UML comprend les diagrammes de cas d'utilisation qui constituent un moyen pratique et très efficace pour documenter les fonctionnalités des systèmes à développer.

3.2. Diagramme de cas d'utilisation

Le diagramme de cas d'utilisation permet de décrire les scénarios d'utilisation possibles pour lesquels un système est développé. Il exprime ce que le système doit faire mais ne traite pas les détails de réalisation (structures de données, algorithmes, etc.) qui sont couverts par d'autres diagrammes (diagramme de classe, diagrammes d'interaction, etc.).

Le diagramme des cas d'utilisation modélise également quel utilisateur du système utilise quelle fonctionnalité, c'est-à-dire qu'il exprime qui travaillera réellement avec le système à construire. Pour résumer, ce diagramme peut être utilisé pour modéliser (i) ce qui est décrit (système), (ii) qui est en interaction avec le système (acteurs), (iii) ce que peuvent faire les acteurs (cas d'utilisation).

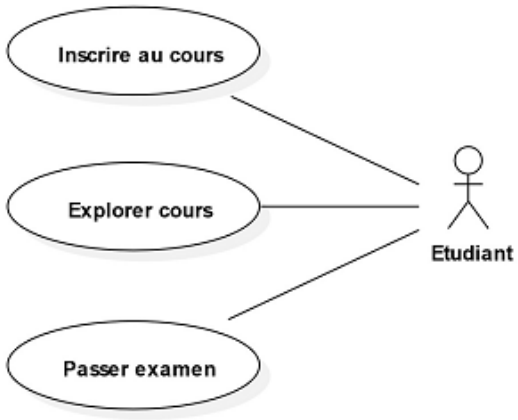


Figure 1 : Diagramme de cas d'utilisation

3.2.1. Cas d'utilisation

Un cas d'utilisation décrit une fonctionnalité attendue du système à développer. Cette fonctionnalité constitue un avantage significatif aux acteurs qui communiquent avec le cas d'utilisation qui la représente. Le cas d'utilisation inclut un certain nombre de tâches qui sont exécutées lors de l'utilisation de ce système. En général, un cas d'utilisation est déclenché soit par un acteur, soit par un événement déclencheur. Un exemple d'événement déclencheur est que c'est la fin de la formation et que, par conséquent, le cas d'utilisation "Délivrer diplôme" doit être exécuté. Un cas d'utilisation est généralement représenté par une ellipse. Le nom du cas d'utilisation est

spécifié directement dans l'ellipse. D'autres alternatives de notation existent et sont toutes valables (par exemple, un rectangle qui contient le nom du cas d'utilisation dans le centre et une petite ellipse dans le coin supérieur droit), mais la première forme de représentation est communément utilisée.

Les cas d'utilisation sont généralement regroupés dans un rectangle qui indique les limites du système à décrire. L'exemple de la figure 1 montre le système de suivi de formations, qui propose trois cas d'utilisation : "Inscrire au cours", "Explorer cours" et "Passer examen". Ces cas d'utilisation peuvent être déclenchés par l'acteur Etudiant.

3.2.2. Acteurs

En plus des fonctionnalités du système, il est essentiel de préciser qui travaille et interagit réellement avec le système. Le diagramme de cas d'utilisation permet de représenter les acteurs qui interagissent avec le système dans le cadre des cas d'utilisation avec lesquels ils sont associés. Les acteurs sont représentés par l'icône standard en forme de bâton, des rectangles (contenant l'information complémentaire «acteur»), ou par un symbole librement définissable. L'exemple de la figure 1 ne contient que l'acteur Etudiant, qui peut s'inscrire à un cours, explorer un cours et passer examen.

Les acteurs ne représentent pas des utilisateurs spécifiques dans le système ; ils représentent les rôles que les utilisateurs assument. Si un utilisateur a adopté un rôle donné, cet utilisateur est autorisé pour exécuter les cas d'utilisation associés à ce rôle.

Les diagrammes de cas d'utilisation peuvent contenir des acteurs humains (par exemple, étudiant ou professeur) ou non humains (par exemple, serveur de courrier électronique), des acteurs actifs ou passifs et des acteurs primaires ou secondaires.

Un acteur en interaction avec le système peut être actif, ce qui signifie que l'acteur initie l'exécution du cas d'utilisation (par exemple, le professeur). Si l'interaction implique plutôt que l'acteur soit utilisé par le système pour fournir un service pour l'exécution du cas d'utilisation, l'acteur est qualifié de passif (serveur de courrier par exemple).

Les acteurs secondaires prennent toujours un avantage réel de l'exécution du cas d'utilisation, tandis que les acteurs secondaires ne reçoivent aucun avantage direct de l'exécution du cas d'utilisation.

Un acteur est toujours manifestement en dehors du système, c'est-à-dire qu'un utilisateur ne fait jamais partie du système et n'est donc jamais implémenté. Les données concernant cet utilisateur peuvent être, cependant, nécessaires pour le système et doivent être donc implémentées. Il est donc crucial de distinguer entre éléments faisant partie du système à implémenter et ceux qui servent d'acteurs.

3.2.3. Liaisons

Les diagrammes de cas d'utilisation définissent différentes formes de liaisons entre éléments de modélisation ; des associations entre acteurs et cas d'utilisation, des relations entre acteurs et des relations entre cas d'utilisation.

3.2.3.1. Associations

Un acteur est connecté avec les cas d'utilisation via des associations (représentées par des lignes pleines) qui expriment le fait que l'acteur communique avec le système et utilise une certaine

fonctionnalité. Une association est toujours binaire, ce qui signifie qu'elle est toujours spécifiée entre un cas d'utilisation et un acteur, mais des multiplicités peuvent être spécifiées pour indiquer le nombre d'acteurs impliqués dans l'exécution du cas d'utilisation.

Dans un diagramme de cas d'utilisation, chaque acteur doit communiquer avec au moins un cas d'utilisation et chaque cas d'utilisation doit avoir une association directe ou indirecte avec au moins un acteur.

3.2.3.2. Relations entre acteurs

Certains cas d'utilisation peuvent être utilisés par différents acteurs ce qui signifie que ces acteurs possèdent des propriétés communes qui peuvent être regroupées et décrites dans un super-acteur commun. Par exemple, il est possible que le professeur principal et l'assistant soient autorisés à créer des travaux pratiques.

Pour une meilleure structuration du diagramme de cas d'utilisation, ces acteurs sont représentés dans une relation d'héritage les uns avec les autres. Lorsqu'un acteur hérite d'un autre acteur, le sous-acteur sera impliqué dans tous les cas d'utilisations associées au super-acteur.

L'héritage entre acteurs est représenté graphiquement de la même façon que dans le cas d'héritage entre classes. S'il n'y a pas d'instance d'un acteur, celui-ci peut être étiqueté avec le mot-clé {abstract}. Alternativement, les noms des acteurs abstraits peuvent être représentés en caractères italiques.

Il y a une différence majeure entre acteurs participants eux-mêmes à un cas d'utilisation et acteurs ayant un super-acteur commun qui participe à ce cas. Dans la première situation, les acteurs doivent participer au cas d'utilisation ; dans la deuxième situation, chacun d'entre eux hérite l'association avec le cas d'utilisation et participe ensuite individuellement à ce cas.

Dans l'exemple de la figure 2, les acteurs Principal et Assistant héritent de l'acteur abstrait Professeur et peuvent, par conséquent, exécuter le cas d'utilisation "Créer TPs". Seuls les professeurs principaux peuvent créer un nouveau cours ; en revanche, les TPs ne sont publiés que par les assistants. Pour le cas "Evaluer étudiant", le professeur principal est un acteur requis ; en outre, l'assistant peut être impliqué de manière facultative, ce qui s'exprime par la multiplicité 0..1.

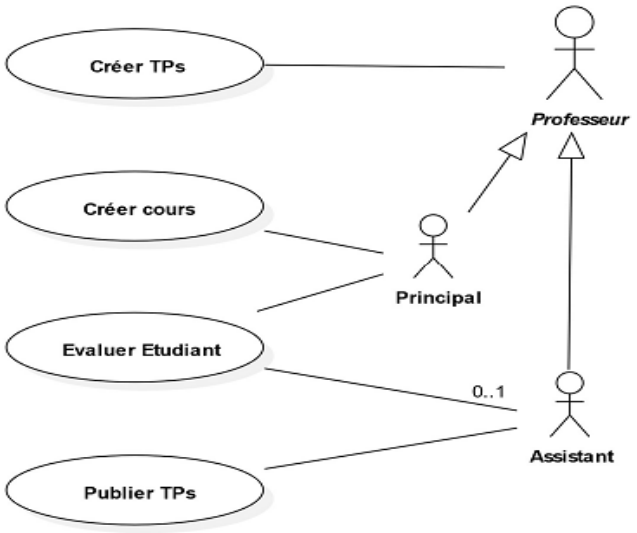


Figure 2 : Diagramme de cas d'utilisation avec héritage entre acteurs

3.2.3.3. Relations entre cas d'utilisation

Les cas d'utilisation peuvent avoir trois différentes formes de relations entre cas ; des relations "d'inclusion", des relations "d'extension" et des relations d'héritage entre cas d'utilisation.

Si un cas d'utilisation de base inclut un autre cas d'utilisation, le comportement du cas d'utilisation inclus est intégré dans le comportement du cas d'utilisation de base ; le cas d'utilisation de base requiert toujours le comportement du cas d'utilisation inclus pour pouvoir offrir sa fonctionnalité, mais le cas d'utilisation inclus peut être exécuté seul. La relation d'inclusion est représentée par une flèche en pointillé (allant du cas de base au cas inclus) étiquetée avec le stéréotype «include».

Dans le diagramme de cas d'utilisation de la figure 3, l'utilisation Les cas "Annoncer événement" et "Affecter conférencier" sont dans une relation "d'inclusion", où l'annonce d'événement représente le cas d'utilisation de base. Par conséquent, chaque fois qu'un nouvel événement est annoncé, le cas d'utilisation inclus "Affecter conférencier" doit être également exécuté. Dans cet exemple, l'acteur Professeur est impliqué dans l'exécution des deux cas d'utilisation et le cas inclus peut être exécuté indépendamment du cas de base ; par exemple, il est possible d'affecter des conférenciers à un événement déjà existant.

Un cas d'utilisation peut inclure plusieurs autres cas d'utilisation et peut être inclus par de multiples cas d'utilisation. Dans de telles situations, il faut veiller à ce qu'aucun cycle ne se produise. Si un cas d'utilisation est dans une relation d'extension avec un cas d'utilisation de base, alors le cas d'utilisation de base peut utiliser le comportement du cas d'extension mais n'est pas obligé de le faire.

Le cas d'extension peut donc être activé par le cas de base afin d'insérer son comportement dans le cas de base. Les deux cas d'utilisation peuvent aussi être exécutés indépendamment les uns des autres.

Une relation d'extension est représentée par une flèche en pointillé (allant du cas d'utilisation d'extension au cas d'utilisation de base) étiquetée avec le stéréotype «extend».

La relation définit deux éléments additionnels ; la condition et le point d'extension.

La condition qui doit être remplie pour que le cas d'utilisation de base puisse insérer le comportement du cas d'utilisation d'extension peut être spécifiée pour chaque relation d'extension. Elle est indiquée par le mot-clé Condition et spécifiée entre parenthèses dans une note attachée à la relation d'extension correspondante.

Le point d'extension définit l'endroit auquel le comportement du cas d'extension doit être inséré dans le cas d'utilisation de base. Les points d'extension sont écrits directement dans le cas d'utilisation ou dans une note attachée à la relation d'extension correspondante et sont indiqués par le mot clé Extension Point.

Dans l'exemple de la figure 3, les deux cas d'utilisation "Annoncer événement" et "Réserver salle" sont reliés par une relation d'extension. Lorsqu'un nouvel événement est annoncé, il est possible (et non pas obligatoire) de réserver une salle de conférence.

Un cas d'utilisation étend plusieurs cas et peut lui-même être étendu par plusieurs cas d'utilisation. Là encore, aucune forme de cycle ne peut être tolérée.

Tout comme dans le cas des acteurs, l'héritage entre cas d'utilisation est également possible. Ainsi, les propriétés communes et le comportement commun de différents cas d'utilisation peuvent être regroupés dans un cas d'utilisation père. Le cas d'utilisation fils hérite du comportement du cas père et adopte par conséquent, sa fonctionnalité de base, mais peut aussi soit étendre, soit modifier ce comportement. Le cas fils hérite également toutes les relations du cas père. Si un cas d'utilisation est étiqueté {abstract}, il ne peut pas être exécuté directement ; seuls les cas d'utilisation qui spécialisent ce cas sont exécutables.

Le diagramme de la figure 3 montre un exemple de relation d'héritage entre le cas d'utilisation abstrait "Annoncer événement" et les cas "Annoncer exposé" et "Annoncer conférence". Les deux cas d'utilisation héritent l'association entre le cas père et l'acteur Professeur et sont ainsi liés à au moins un acteur Professeur. Les deux cas d'utilisation doivent également exécuter le comportement du cas d'utilisation "Affecter conférencier" en raison de la relation d'inclusion définie entre ce cas et le cas père "Annoncer événement".

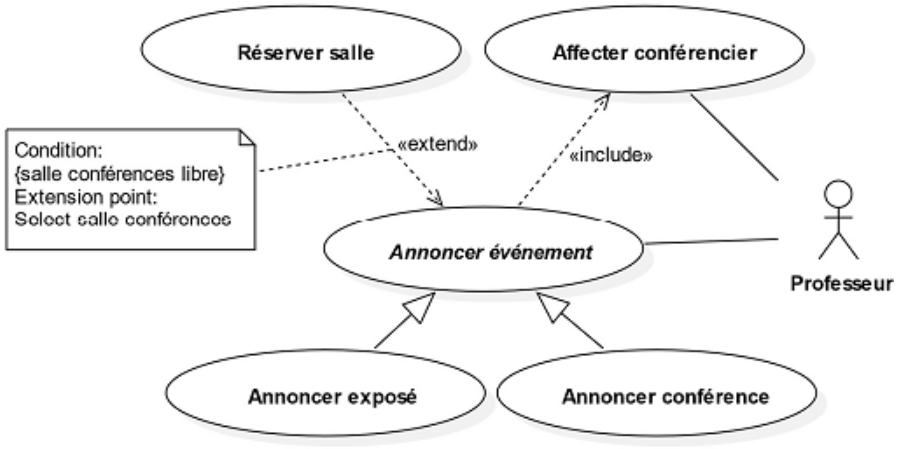


Figure 3 : diagramme de cas d'utilisation avec relations entre cas.

3.3. Développement de diagrammes de cas d'utilisation

Le développement de diagrammes de cas d'utilisation consiste à identifier d'abord les acteurs et les cas d'utilisation, puis à les mettre en relation les uns avec les autres et enfin, à décrire de manière détaillée les cas d'utilisation.

Les cas d'utilisation sont généralement identifiés par analyse des documents d'exigences ou des attentes des futurs utilisateurs. Les documents relatifs aux exigences sont généralement des spécifications en langage naturel qui expliquent ce que le client attend d'un système. Ils doivent documenter de manière relativement précise qui utilisera le système (acteurs) et comment il l'utilisera (cas d'utilisation).

L'approche basée sur l'analyse des attentes des futurs utilisateurs procède d'abord par l'identification des futurs utilisateurs, c'est-à-dire les acteurs et détermine ensuite les cas d'utilisation associés à ces acteurs.

3.3.1. Identification des acteurs

Afin d'identifier les acteurs du diagramme de cas d'utilisation, l'analyse des attentes des futurs utilisateurs doit répondre aux questions suivantes :

- qui utilise les principales fonctionnalités du système ?

- qui a besoin de soutien pour son travail quotidien ?
- qui est responsable de l'administration du système ?
- quels sont les systèmes logiciels ou dispositifs externes avec lesquels le système doit communiquer ?
- qui est intéressé par les résultats du système ?

3.3.2. Identification des cas d'utilisation

Après identification des acteurs, il est possible d'en déduire les cas d'utilisation en répondant aux questions suivantes :

- quelles sont les principales tâches qu'un acteur doit accomplir ?
- un acteur veut-il consulter ou modifier des informations internes dans système ?
- un acteur veut-il informer le système sur des changements dans d'autres systèmes ?
- un acteur doit-il être informé des événements inattendus au sein du système ?

Dans de nombreuses situations, les cas d'utilisation sont développés de manière itérative et incrémentale. Généralement, il faut commencer par l'identification des exigences de haut niveau qui reflètent les objectifs métiers du système. Ces exigences sont ensuite raffinées jusqu'à ce que, sur le plan technique, tout ce que le système devrait être en mesure de faire soit identifié.

Par exemple, une exigence de haut niveau pour un système d'administration des études pourrait être que le système est utilisé pour le suivi des formations. Cette exigence peut être raffinée, par exemple, en exigences plus détaillées telles que les professeurs devraient pouvoir créer des cours et des examens et que les étudiants devraient pouvoir s'inscrire aux cours, explorer des cours et passer des examens, etc.

3.3.3. Description des cas d'utilisation

Afin de maintenir la clarté et l'utilité des diagrammes de cas d'utilisation même s'ils sont larges, il est extrêmement important de choisir des noms courts et concis pour les cas d'utilisation et de procéder à la description détaillée de ces cas d'utilisation. L'approche structurée généralement adoptée pour la description des cas d'utilisation maintient les informations suivantes :

- Nom
- Description succincte
- Précondition : condition préalable à la bonne exécution
- Postcondition : état du système après une exécution réussie
- Situations d'erreur : erreurs relevant du domaine du problème
- État du système après occurrence d'erreur
- Les acteurs qui communiquent avec le cas d'utilisation
- Evénements déclencheur : événements qui initient/démarrent le cas d'utilisation
- Exécution standard : étapes individuelles à suivre
- Exécutions alternatives : déviation par rapport à l'exécution standard

La table 1 montre une description du cas d'utilisation "Réserver salle" dans le système d'administration des études. La description est extrêmement simplifiée mais tout à fait suffisante. Le chemin d'exécutions standard et le chemin alternatif pourraient être raffinés davantage ou d'autres situations d'erreur et des exécutions alternatives pourraient être envisagés par la suite.

Table 1 Table de description de cas d'utilisation

Nom	Réserver salle
Description succincte	un professeur réserve une salle de conférences à l'université pour un événement donné
Pré-condition	(i) le professeur est un utilisateur authentifié (ii) le professeur est autorisé à réserver des salles de conférences.
Post-condition	une salle de conférence est réservée
Situations d'erreur	il n'y a pas de salle de conférences libre
Etat du système en cas d'erreur	le professeur n'a pas réservé de salle de conférences
Acteurs	Professeur
Exécution standard	(1) le professeur sélectionne la salle de conférences (2) le professeur sélectionne la date (3) le système confirme que la salle de conférence est libre (4) le professeur confirme la réservation
Exécutions alternatives	(3) la salle de conférences n'est pas libre (4) le système propose d'autres salles de conférences (5) le professeur sélectionne une autre salle de conférences et confirme la réservation

3.4. Intérêt et définition, Notation

Texte…

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

Texte…

4.1. Diagramme de classes

Texte…

4.2. Diagramme d'objets

Texte…

5. Chapitre 5: Diagrammes UML : vue dynamique

Texte…

5.1. Diagramme d'interaction (Séquence et collaboration)

Texte…

5.2. Diagramme d'activités

Texte…

5.3. Diagramme d'état/transitions

Texte…

6. Chapitre 6: Autres notions et diagrammes UML

Texte…

6.1. Composants, déploiement, structures composite.

Texte…

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

Texte…

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

Texte…

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

Texte…

La partie : Travaux Dirigés

Les méthodes d'analyse et de conception définissent une **démarche** (suite d'étapes) et un **langage de modélisation** (série de modèles) pour mener correctement le développement d'applications logicielles.

Le développement de programmes procéduraux (qui sont définis uniquement par des données et des traitements) utilise seulement des modèles pour les données et des modèles pour les traitements (comme dans le cas de la méthode Merise : MCD-MCT…). Une application objet est caractérisée, par contre, par cinq différentes perspectives :

- Les **fonctions** que l'application doit assurer (diagramme de cas d'utilisation)
- La **structure** des objets de l'application (diagrammes de classes, d'objets)
- Les **interactions** entre objets de l'application pour réaliser les fonctions (diagrammes de séquence, de communication)
- La **dynamique** interne et inter-objet (machine d'état et diagramme d'activité)
- L'**architecture** de l'application (diagrammes de packages, de composants et de déploiement)

Les travaux dirigés et pratiques de la matière Génie Logiciel sont axés sur la **modélisation UML** dans le cadre d'un **développement objet**. Les objectifs, par ordre de priorité, sont les suivants :

- Modélisation UML :

  1. diagramme de cas d'utilisation
  2. diagrammes de classes, d'objets
  3. diagrammes de séquence, de communication
  4. machine d'état et diagramme d'activité
  5. diagrammes de packages, de composants et de déploiement

- Démarche de développement objet :

  1. Méthode d'analyse et de conception objet
  2. Processus

- Pratiques de développement objet :

  1. Principes SOLID
  2. Design patterns

Liens :

- Site officiel du langage de modélisation UML : https://www.uml.org/
- L'outil de modélisation starUML : https://staruml.io/

1. TD N°1 : Introduction à la Modélisation Objet

1ère Partie : Abstraction & Encapsulation

Proposer une abstraction (classe, instance, attribut, méthode, etc.) des éléments suivants :

1. Une personne
2. Une personne qui s'appelle Ali
3. Un compte bancaire
4. La consultation du solde du compte bancaire
5. Un employé
6. Le nom de l'employé
7. L'adresse de l'employé
8. La modification de l'adresse de l'employé
9. La liste des employés
10. L'ajout d'un employé

2ème Partie : Associations, Multiplicités et Rôles

Modéliser les associations exprimées par les règles suivantes :

1. Les étudiants sont inscrits dans une seule classe
2. Une classe occupe une seule salle
3. Une classe regroupe entre cinq et dix étudiants et possède un seul étudiant délégué
4. Un étudiant suit plusieurs matières ; pour chaque matière, l'étudiant obtient une note
5. Un enseignant assure une ou plusieurs matières
6. Une matière possède un enseignant chargé de cours et plusieurs enseignants assistants ; un enseignant peut être chargé de cours dans une matière et assistant dans un autre

3ème Partie : Héritage (Généralisation, Spécialisation & Polymorphisme)

Proposer une modélisation objet des relations suivantes :

1. Les enseignants sont des employés
2. Les enseignants peuvent être des maîtres_assistants, des maîtres_de_conférences ou des professeurs
3. Un enseignant peut être permanent ou vacataire ; les enseignants permanents sont caractérisés par date_recrutement et les enseignants vacataires sont caractérisés par durée_contrat
4. Le calcul du salaire des enseignants permanents est basé sur leurs grade et échelon. Le salaire des vacataires est calculé sur la base de leurs diplômes et du nombre d'heures d'enseignement
5. Un enseignant permanent peut être stagiaire ou titulaire ; un stagiaire qui fait preuve de qualités professionnelles suffisantes est titularisé dans douze mois

4ème Partie : Agrégation & Composition

Modéliser les règles suivantes :

1. Une collection regroupe plusieurs véhicules
2. Un véhicule est composé de carrosserie, moteur et de quatre roues
3. Les pièces sont caractérisées par référence_pièce et désignation_pièce
4. Les pièces peuvent être simples ou composites
5. Les pièces simples sont caractérisées par prix_achat
6. Les pièces composites sont caractérisées par coût_assemblage et peuvent être composées de plusieurs autres pièces (simples et/ou composites)
7. Les pièces composites sont assemblées dans un seul atelier
8. Un moteur est une pièce composite

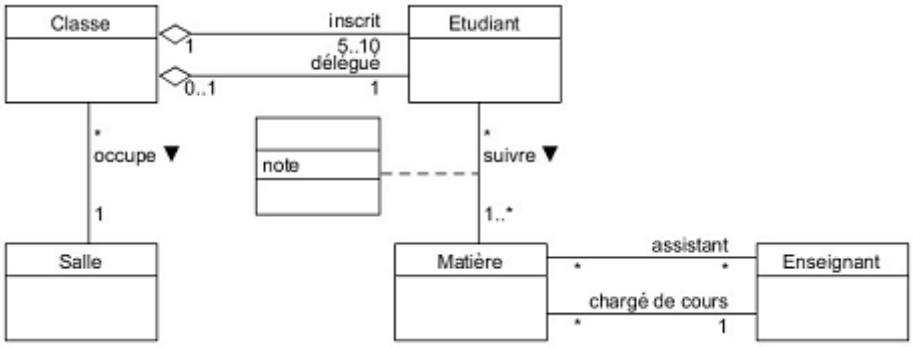
Correction TD N°1

1ère Partie : Abstraction & Encapsulation

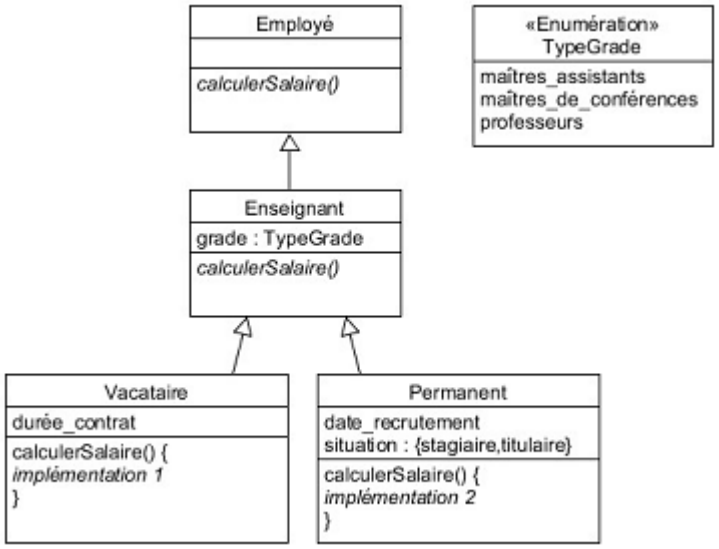
Proposer une abstraction (classe, instance, attribut, méthode, etc.) des éléments suivants :

1. Une personne : classe
2. Une personne qui s'appelle Ali : instance de la classe Personne avec attribut nom de valeur 'Ali'
3. Un compte bancaire : classe
4. La consultation du solde du compte bancaire : méthode dans la classe Compte_Bancaire avec attribut solde
5. Un employé : classe
6. Le nom de l'employé : attribut
7. L'adresse de l'employé : attribut
8. La modification de l'adresse de l'employé : méthode dans la classe Employé
9. La liste des employés : classe (classe container)
10. L'ajout d'un employé : méthode dans la classe Liste_Employés (l'opération désigne l'ajout d'un élément à une liste et non pas la création de cet élément)

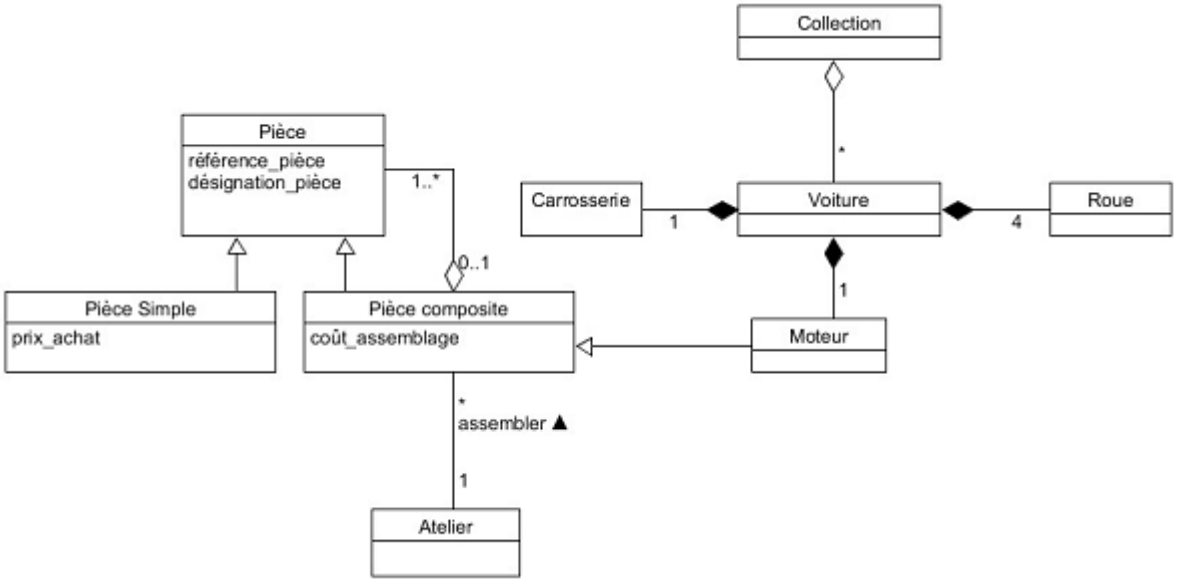
2ème Partie : Associations, Multiplicités et Rôles



3ème Partie : Héritage (Généralisation, Spécialisation & Polymorphisme)



4ème Partie : Agrégation & Composition



La partie : Travaux Pratiques

1. TP N°1: Expression des besoins et analyse

1ère Partie : Description du système - besoins fonctionnels

Une entreprise de location de voiture désire offrir ses services via le web :

Tout client peut consulter le catalogue de modèles de voitures, en explorant l'index des modèles de voitures ou par recherche. Dans le cas de recherche, le client spécifie les détails des modèles auxquels il est intéressé (par exemple la catégorie, la marque, etc.). Les résultats d'exploration d'index ou de recherche sont affichés comme une collection de modèles de voitures avec des informations de base telles que le nom du modèle de voiture. Le client peut alors choisir de voir des informations supplémentaires (la description par exemple) pour un modèle de voiture particulier.

Les clients peuvent être membres ou non membres. Un client membre doit effectuer un logon pour avoir accès aux services supplémentaires : effectuer réservation, annuler réservation, vérifier détails personnels, voir ses réservations en cours, changer le mot de passe du logon, voir ses locations en cours et effectuer un log off.

Les assistants sont impliqués dans les opérations associées aux réservations telles que déplacer les voitures depuis et vers l'espace réservé, c'est-à-dire effectuer réservation et annuler réservation.

Dans l'ordre de voir les détails d'un modèle de voitures, un client doit être en cours de consultation de la liste de modèles de voitures (résultat obtenu par voie d'exploration ou de recherche).

Dans l'ordre de réserver un modèle de voitures, un membre doit être en cours de consultation des détails de ce modèle (un non-membre ne peut pas effectuer une réservation, même quand il est en cours de consultation des détails du modèle à réserver).

Dans l'ordre d'annuler une réservation, un membre doit être en cours de consultation de ses réservations en cours.

Donner :

- la liste des acteurs avec description succincte
- la liste des cas d'utilisation avec description succincte de chaque cas
- le diagramme de cas d'utilisation
- la description détaillée de chaque cas d'utilisation

2ème Partie : Analyse du problème - analyse statique

La phase recueil d'information a permis de dégager la description suivante des données pertinentes du système :

Un modèle de voitures est caractérisé par nom et prix et possède des détails supplémentaires (capacité moteur, description, vidéo et poster). Un modèle de voitures est fabriqué par un ou plusieurs constructeurs (nom). Il est vendu par un seul vendeur (nom). Les modèles de voitures sont classés en catégories (nom) ; un modèle de voiture appartient à une seule catégorie et une catégorie regroupe plusieurs modèles de voitures. Un modèle de voitures peut avoir plusieurs voitures et une voiture (identifiant et distance parcourue) est liée à un seul modèle de voitures. Une voiture possède des détails supplémentaires (code à barres et immatriculation).

Une location est caractérisée par numéro, date début, date fin et montant total. Elle concerne une ou plusieurs voitures et une voiture peut être concernée par une seule location. Un client peut effectuer plusieurs locations et une location est associée à un seul client.

Les clients (nom, numéro téléphone et montant dû) peuvent être membres ou non membres.

Un membre est caractérisé par numéro, position et montant dû. Il possède un compte internet (mot de passe), une adresse (numéro, rue, ville et code postal) et une carte de crédit (numéro, type et date expiration). Les cartes de crédit et les adresses peuvent être partagées par plusieurs membres.

Les non-membres sont caractérisés par leurs numéros du permis de conduire.

Un client peut réserver plusieurs modèles de voitures et un modèle de voitures peut être réservé par plusieurs clients. Les informations numéro, échéance et état réservation caractérisent chaque réservation d'un modèle de voiture par un client.

Donner :

- la liste des classes entités
- les propriétés de chaque classe
- le diagramme de classes

Correction TP N°1

1ère Partie : Description du système - besoins fonctionnels

1) Liste des cas d'utilisation :

- Client : …
- Membre : …
- Non membre : …
- Assistant : …

2) Liste des cas d'utilisation :

- U1: Index de navigation: un client navigue sur l'index des modèles de voiture. (Spécialisé U13, inclut U2.)
- U2: Afficher les résultats: un client voit le sous-ensemble de modèles de voitures qui ont été récupérés. (Inclus par U1 et U4, prolongé par U3.)
- U3: Afficher les détails du modèle de voitures: Un client voit les détails d'un modèle de voitures récupéré, tels que description et annonce. (Prolonge U2, prolongé par U7.)
- U4: recherche: un client recherche des modèles de voiture en spécifiant des catégories, des marques et un moteur. tailles. (Spécialisé U13, comprend U2.)
- U5: connexion: un membre se connecte à iCoot en utilisant son numéro de membre et mot de passe. (Prolongé par U6, U8, U9, U10 et U12.)
- U6: Afficher les détails du membre: un membre affiche certains des détails stockés par iCoot, comme le nom, adresse et détails de la carte de crédit. (Prolonge U5.)
- U7: Faire une réservation: un membre réserve un modèle de voitures lors de la visualisation de ses détails. (Prolonge U3.)
- U8: Afficher les locations: un membre consulte un résumé des voitures qu'il loue actuellement. (Prolonge U5.)
- U9: Modifier le mot de passe: un membre modifie le mot de passe qu'il utilise pour se connecter. (Prolonge U5.)
- U10: Afficher les réservations: un membre consulte les résumés de ses réservations non conclues, comme la date, l'heure et le modèle de voitures. (Prolonge U5, prolongé par U11.)
- U11: Annuler la réservation: un membre annule une réservation non conclue. (Prolonge U10.)
- U12: Déconnexion: un membre se déconnecte d'iCoot. (Prolonge U5.)
- U13: Rechercher des modèles de voitures: un client récupère un sous-ensemble de modèle de voitures du catalogue.

3) La description détaillée de chaque cas d'utilisation

U4: Recherche.

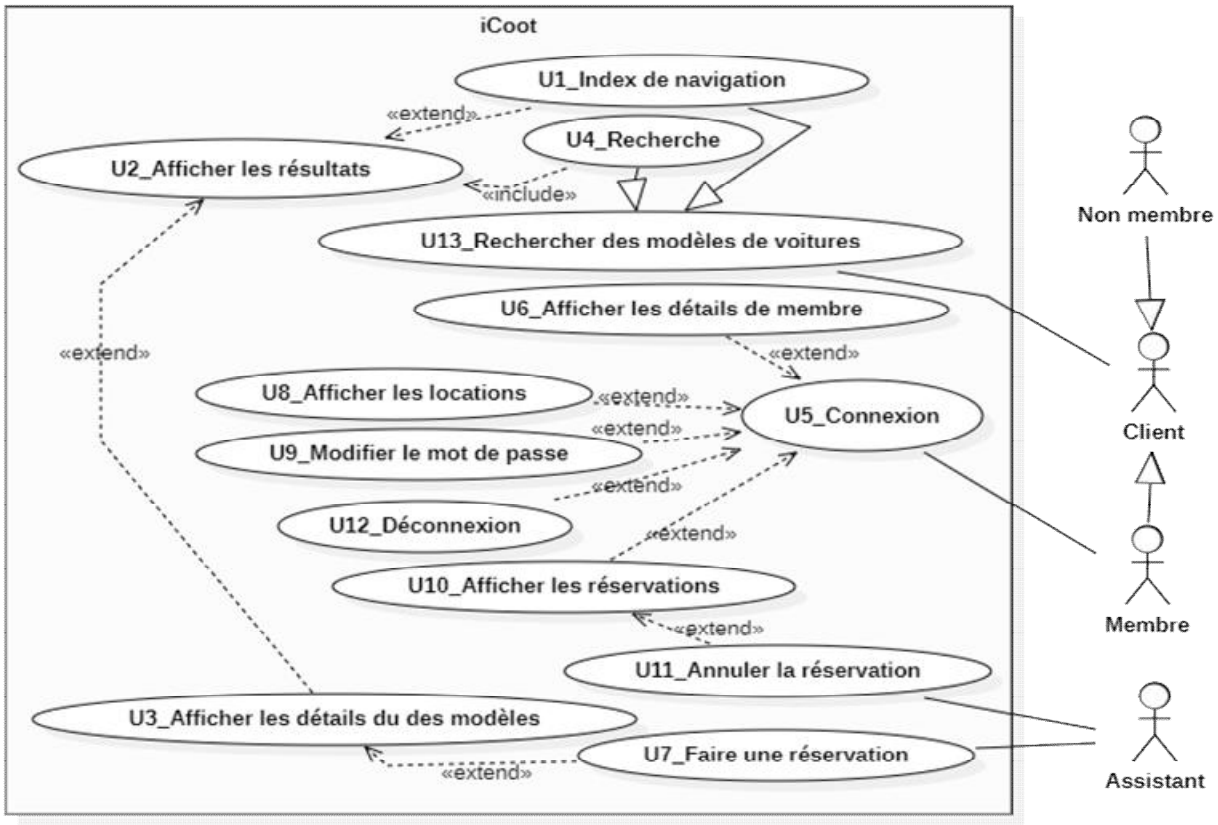
Conditions préalables: aucune.

1. Le client sélectionne les catégories requises (le cas échéant).
2. Le client sélectionne les marques requises (le cas échéant).
3. Le client sélectionne les tailles de moteur requises (le cas échéant).
4. Le client lance la recherche.
5. Incluez U2.

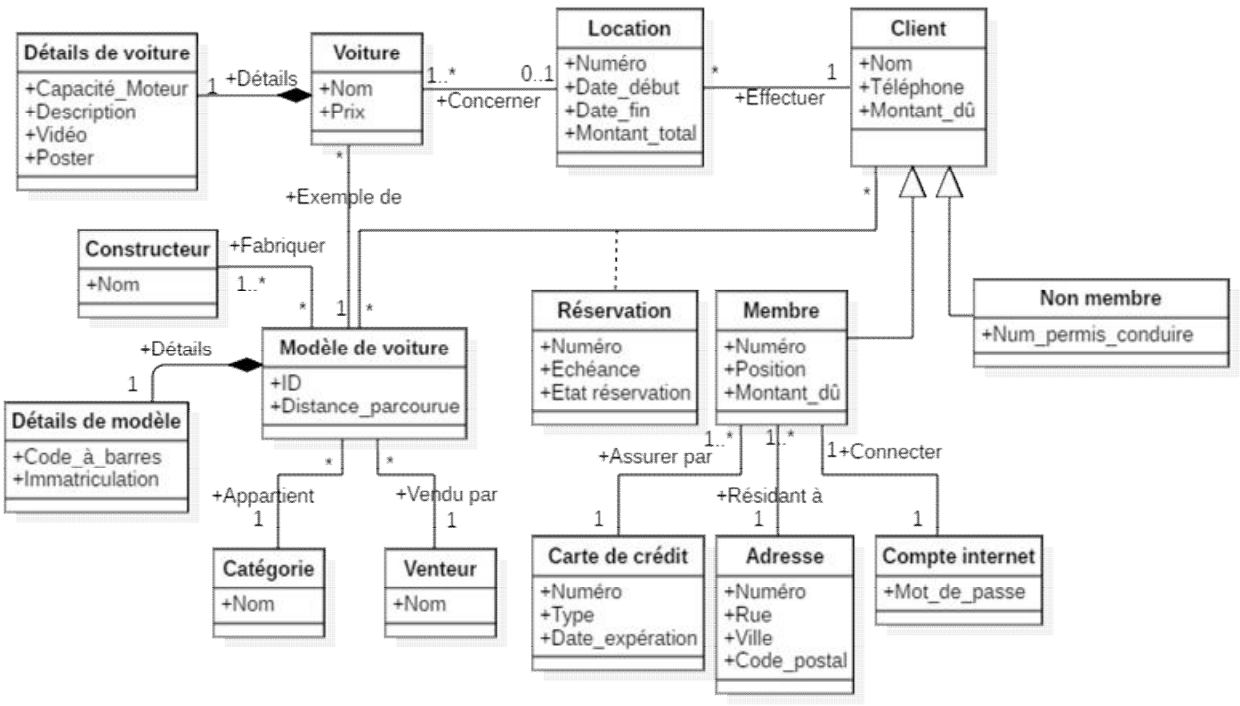
Post-conditions: aucune.

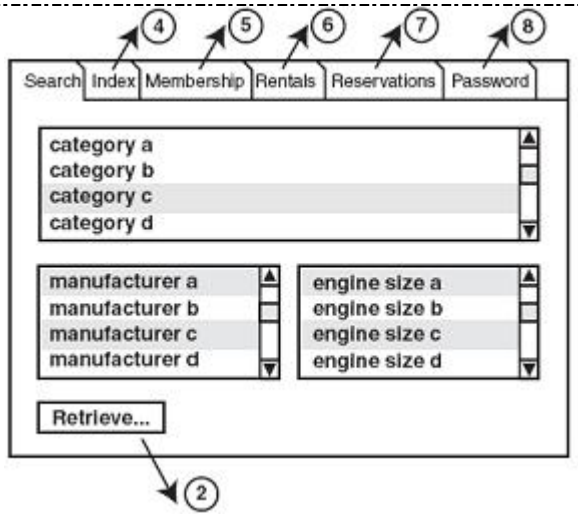
Chemins anormaux: a1. Si le client ne spécifie aucune catégorie, marque ou taille de moteur, plutôt qu'en récupérant l'intégralité du catalogue, iCoot ne doit pas autoriser le lancement de la recherche.

4) Le diagramme de cas d'utilisation:

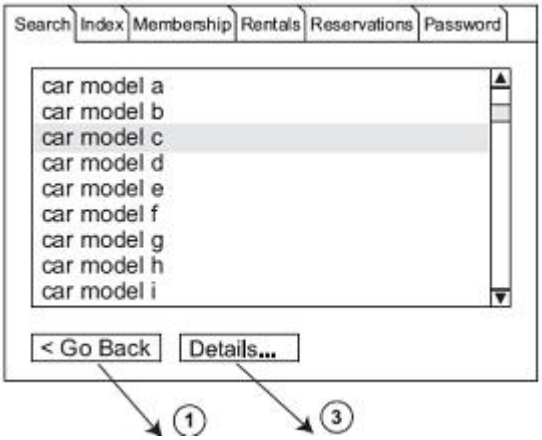


2ème Partie : Description du système - besoins fonctionnels

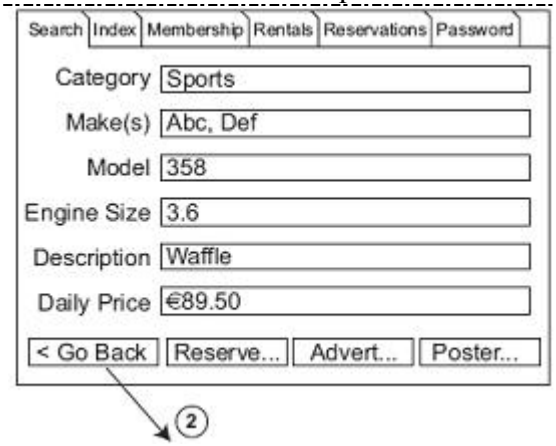




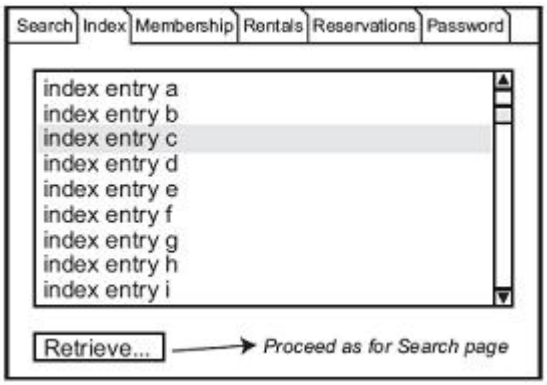
UI1: créer requête



UI2: consulter résultats

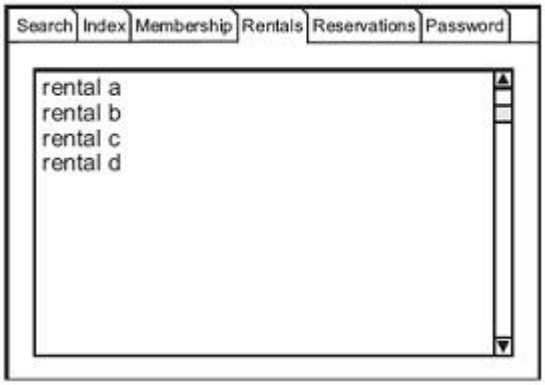


UI3: consulter détails modèle de voiture

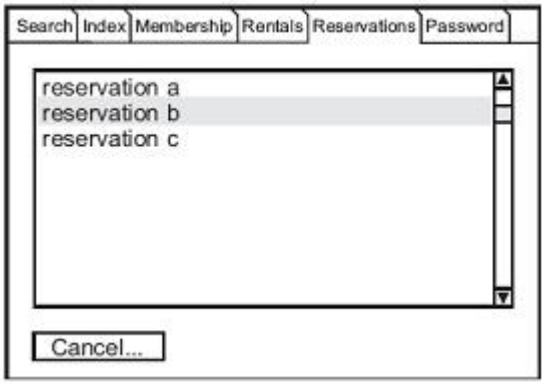


UI4: sélectionner un titre index

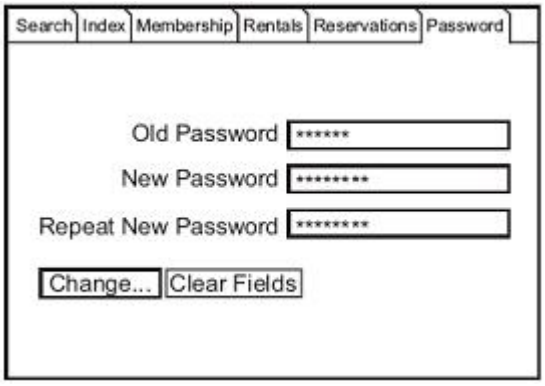
UI5: consulter détails membre



UI6: consulter locations



UI7: consulter réservations



UI8: changer le mot de passe

2. TP N°2 : Installation de StarUML

Pour avoir une version stable de StarUML suivez les étapes suivantes:

1. Télécharger une version StarUML, 7-Zip et le plugin Asar.zip
2. Installer StarUML et 7-Zip.
3. Installer sur 7-Zip le plugin Asar (Créer le sous-dossier «Formats» dans le dossier d'installation de 7-Zip «...\ProgramFiles\7-Zip». Puis, copier à partir de plugin «Asar.zip» le fichier «Asar.64.dll» ou «Asar.32.dll» dans le dossier «...\ProgramFiles\7-Zip\Formats». 7-Zip cherche automatiquement «Asar7z» et l'utilise pour ouvrir des fichiers «.asar»).
4. Extraire le fichier «...\ProgramFiles\StarUML\resources\app.asar» dans un dossier «...\app».
5. Ouvrir le fichier «...\app\src\engine\license-manager.js» par un éditeur de texte puis modifier la fonction checkLicenseValidity () comme suite :

```
checkLicenseValidity () {
    this.validate().then(() => {
            setStatus(this, true)
     }, () => {
            //setStatus(this, false)
            //UnregisteredDialog.showDialog()
            setStatus(this, true) //<-- add this line
    })
}
```

6. Enregistrer les modifications et créer un nouveau fichier «app.azar» par 7-Zip en utilisant le dossier modifié «...\app».
7. Enfin, replacer le fichier «...\ProgramFiles\StarUML\resources\app.asar» par le nouveau fichier «app.azar».
8. Lancer StarUML.