Ministère de l'Enseignement Supérieur et de la recherche scientifique

Université Mohamed Khider Biskra

Faculté de Sciences Exactes et Science de la Nature et de la vie

Département d'informatique

Support de cours pour la matière :

Génie Logiciel 1

Niveau : 2ème année licence

Proposé par :

Dr. Laid Kahloul

Année universitaire 2017-2018

Sommaire

Chapitre III: Le langage UML

Introduction Générale

Introduction générale

Le génie logiciel GL (ou software engineering) est la science dont l'objectif est de proposer des techniques, démarches, approches, langages favorisant la production de logiciels de qualités. Plusieurs qualités sont requises dans un logiciel, mais les plus importantes sont souvent : la fiabilité, la maintenabilité, l'efficacité, etc. D'autre part et du sa complexité, le génie logiciel est considéré aussi comme un art plutôt qu'une science. Cet art a pour objectif de spécifier, concevoir, réaliser et maintenir des programmes ainsi que leur documentation qui accompagnent le développement, l'usage, et la maintenance.

L'objectif de ce support de cours et mettre entre les mains des étudiants informaticiens, novices au domaine de génie logiciel, un document leur introduisant le domaine du génie logiciel. En tant qu'enseignant de cette matière depuis l'année 2004 et presque de manière continue sans interruption, j'ai tenté de rassembler les éléments les plus pertinents à la compréhension et à la maitrise de cette matière par des étudiants de deuxième année licence, où même de troisième année licence ou des étudiants de master ayant besoin de connaissance en génie logiciel pour développer leurs projets de fin d'étude.

Ce support de cours est organisé en trois chapitres, suivis par deux annexes A et B. Le contenu du document se résume comme suit :

Chapitre I : intitulé **« de la crise logicielle au génie logiciel ».** Il commence par une introduction suffisante du domaine de GL. L'objectif de ce premier chapitre est surtout de faire convaincre l'étudiant de l'importance de la matière dans sa formation académique ainsi que de l'importance du GL dans le monde de l'industrie logiciel en pratique. Ce chapitre présente plusieurs raisons de l'apparition du GL et trace son histoire depuis sa naissance dans l'année 1968.

Chapitre II : intitulé « Approches de développement et modèle de gestion de projet logiciel ». Ce chapitre passe à présenter le concept de processus logiciel, du modèle de développement de logiciel, du cycle de vie, des approches de développement et des modèles de gestion de projet logiciel. Les différentes approches les plus étudiées dans la littérature sont discutées, motivées, et critiquées.

Chapitre III : intitulé « langage UML ». C'est le chapitre le plus technique dans la matière. Dans ce chapitre l'étudiant va découvrir le langage UML (unified modeling language). Ce langage est proposé et utilisé pour la description des systèmes orientés objet. Le langage est présenté avec les détails nécessaires permettant à l'étudiant de comprendre son utilité et de pouvoir l'utiliser pour faire des analyses ou des conceptions orientées objet.

Afin de rendre le document encore plus utile, deux annexes comprenant des exercices traités dans la matière ainsi que quelques examens des années précédente. Ces travaux dirigés et examens peuvent donner l'occasion à l'étudiant de tester ses connaissances acquises et de vérifier sa compréhension.

Kahloul Laid

Chapitre I :

De la crise logicielle au génie logiciel

1. Introductions

L'objectif principal de cette introduction est de faire comprendre à l'étudiant, les causes de l'apparition du GL et dans quelles circonstances cette discipline a connu sa naissance. Le cours commence par exposer la crise logicielle, ensuite passe à décrire l'apparition ou la naissance du génie logiciel comme une nouvelle discipline dans le monde de l'informatique.

Cette introduction peut être divisée en deux grandes parties : ***Crise Logicielle***, et ***Génie logiciel***.

2. Crise logicielle et première motivation du Génie Logiciel

Deux éléments principaux seront traités ici : une exposition de l'ère des années soixante et leurs effets dans le monde informatique et l'apparition de la crise logiciel: ses caractère, et ses causes et enfin les solutions qui ont été envisagées.

2.1. Ere des années soixante :

Pour un informaticiens, les années soixante sont caractérisées par deux caractères:

a) L'apparition des machines de la troisième génération : ces machines sont en principe équipées par des architectures à base de circuits intégrés et des microprocesseurs. Ces machines sont caractérisées par : une puissance de calcul importante, une capacité de stockage progressive, et des prix en baisse. L'ordinateur n'est plus la machine privée ou inaccessible.

b) L'apparition des langages de programmation de haut niveau : ces langages ont fait une rupture avec le langage assembleur ou binaire, où le programmeur était trop lié et obligé de connaitre des détails compliquées sur l'architecture de la machine. Plusieurs langages, dont quelques un sont toujours exploités où qui représentaient le noyau de ceux utilisé couramment, ont connu leur apparition dans cette période. Les plus connus sont : Fortran (1954), LISP(1958), COBOL (1959), ALGOL (58, 68, vers le PASCAL 71), APL(1962), SIMULA (1962-67), BASIC (64), PL/I(1964), smaltalk (1970), Prolog (1972), ML(1973),

Ces deux circonstances ont aidé l'informatique à avoir une expansion importante et elle a envahi la vie des humains et de leur société. Plusieurs domaines ont été touchés : militaire, sanitaire, logistique, éducatif, scientifique, spatiale, et même sociale… L'informatique est devenue la solution pour plusieurs problèmes. Les programmes et logiciels sont devenus des axes pertinents dans la vie quotidienne, et tout le monde en dépend.

Cependant, et rapidement on s'est rendu compte que les solutions rapporté par l'informatique à cette époque ne sont pas parfaites. On a remarqué rapidement que le matériel de l'informatique connaît une baisse dans son prix et le rend accessible pour

tout le monde, alors que les logiciels qui pilotent ce matériel connaissent une augmentation importante dans leurs prix. Les logiciels produit ne posaient pas uniquement un problème dans leurs prix, mais les développeurs n'arrivaient plus ni à contrôler les dates de livraison de leurs logiciels, ni à assurer leurs fiabilités !!!!

On a commencé à parler d'une « **Crise Logicielle** ».

2.2. Crise logicielle : caractéristiques et échecs célèbre :

La crise logicielle est la situation dans laquelle, le matériel informatique connaissait une **baisse** dans son prix qui se heurte par une **augmentation dans les prix des logiciels**, un problème dans leur **fiabilité**, un **délai de livraison**, énormément, **dépassés** et un **coût** difficile à **estimer** et à **contrôler**. Plusieurs échecs ont été enregistrés qui prouvent une telle crise :

Problèmes de fiabilité :

1) 1962 : perte de mariner (première sonde américaine) : erreur de transcription
2) 1971 : perte de ballons durant une expérience de météo en France.
3) 1981 : retard du lancement
4) 1990 : problème d'un **switch**: pas d'électricité dans EU et Canada
5) 1994 le célèbre bug du pentium 470 $ de perte
6) 1996 explosion de Ariane 5 : 37 seconde après son lancement ; erreur de transformation de vecteur de bits
7) 2003 : blocage du système de télécom cote Est EU : mise à jour de l'application, et deadlock
8) 2004 : robot de mars ; bloqué (trop de fichier ouvert !!!). http://fr.wikipedia.org/wiki/Spirit_(rover)

Problèmes de délais et de coût :

1) le système d'exploitation d'IBM-360. Un projet prometteur, mais livré tardivement, un prix énorme, et des erreurs importantes, et trop de mémoire pour le stockage
2) projet PL/I 1968 : un projet de rêve jamais achevé !!!!

2.3. Crise logicielle : raison possibles :

Les raisons de cette crise peuvent être classées en deux types :

a) Des raisons dues aux développeurs : les développeurs souffrent de deux problèmes
   - Ils considèrent que la réalisation d'un logiciel est une tâche purement technique (programmation). Et donc, ils ne passent pas le temps nécessaire dans la phase d'analyse et de compréhension. Principe de programmation : Programmer →traquer les erreurs.

- Les développeurs sot d'origine n'ont pas des compétences d'interaction, de communication, de pouvoir écouter, comprendre et convaincre les clients et les utilisateurs de leurs produits

b) Des raisons dues aux caractéristiques du produit logiciel : le logiciel est un produit spécifique qui lui rajoute d'autre problèmes :
   - **Imprévisible** : on ne peut pas facilement avoir une prévision de ce logiciel avant la fin de sa réalisation ;
   - **Flexible** : Un produit trop **souple**, d'où des modifications mineures dans le code source peuvent altérer radicalement son comportement !!!!
   - **Se propage par simple copie** : le **clonage** et la **duplication** de ce produit se fait par simple copie et non pas par une recréation.

2.4. Crise logicielle : solutions envisagées :

Les solutions proposées doivent, bien sur, traiter les deux sources du problème :

a) **Pour les développeurs** : on doit revoir la formation de ces développeurs. Ils doivent avoir conscience que développer un logiciel ce n'est pas une simple programmation, et donc ils doivent prendre leurs temps d'analyse, de compréhension, et de retarder le plus possible cette programmation. Les développeurs doivent avoir aussi d'autre compétences leurs permettent de communiquer et de comprendre les clients et leurs besoins.

b) **Pour le produit logiciel** : en général, l'intervention la plus importante peut prendre place sur la prévision du logiciel. L'idée est d'avoir des visions préalables, partielles sur ce logiciel avant même de sa réalisation. On parle de **prototype**, de **maquette**, de **modèle** … qui sont des représentations intermédiaires permettent de prédire la forme et le fonctionnement du futur logiciel.

   Ces propositions ont été prises en considération de manière plus claire, en proposant toute une nouvelle science dont l'objectif est de développer des logiciels, c'est le Génie Logiciel.

3. Naissance du Génie logiciel et ses objectifs primaires

3.1 Naissance

Le GL a connu sa naissance sous le nom de « «software engineering » dans la période 7-11/10/1968 durant une réunion d'un ensemble de spécialiste dans le développement et l'exploitation des logiciels, sous le parrainage de l'OTAN. Dans la ville de Garmisch-Partenkirchen (Allemagne). Une nouvelle science pour assurer la fiabilité des logiciels produits, contrôler leurs coûts et respecter leurs dates de livraison.

3.2 Définition du logiciel

Ensemble de programmes, procédures et des données manipulées par ces programmes pour faire fonctionner un ordinateur, ainsi que les documentations nécessaire pour le développement et la maintenance de ces programmes.

Logiciel=programmes+données+documentations

3.3 Définition du GL

Définition 1. L'art de spécifier, concevoir, réaliser et faire évoluer, dans des délais raisonnables et avec des moyens, des programmes des données et de documentations de qualités.

Définition 2. La science dont l'objectif est de mettre en place : des **outils**, **langages**, **méthodes**, **approches** favorisant la production de logiciels de qualité.

3.4 Production des Logiciels selon le GL : Processus Logiciel

Le GL considère que le logiciel est un produit de « manufacturing», qui ne se réalise pas d'un seul coup, mais qui se réalise en plusieurs étapes, et qui exige plusieurs personnes (une équipe). On parle d'un **processus logiciel**, où le logiciel nait, évolue, et meurt (être rejeté) à une certaine phase. Par cela, l'objectif du GL se résumera encore dans deux points :

- La définition de ce processus logiciel
- La gestion de ce processus logiciel

Dans ces objectifs le GL propose ce qui est connue par **Approches de développement de logiciel**, et **Modèle de Gestion de projet Logiciel**.

a) **Les approches de développement** : une approche de développement est une démarche ou une politique (théorique) permettant de définir les étapes à suivre pour développer un logiciel. Il y'a une variété d'approches :
   - L'approche de la cascade
   - L'approche de prototypage
   - L'approche de la programmation exploratoire
   - L'approche de la réutilisabilité ;
   - L'approche des transformations formelles

b) **Les modèles de gestion de projet logiciel** : ici on met l'accent, en plus de la définition des activités, sur leurs gestion surtout. Par modèle, on veut dire une image abstraite d'un système ou d'une situation. Donc un modèle de gestion est

une image qui décrit l'évolution de la construction d'un projet logiciel. On trouve par exemple :

- Modèle de la cascade
- Modèle des incréments
- Modèle de la Spirale :

4. Conclusion

Le GL est la science proposée pour résoudre la crise logicielle, et assurer le développement de logiciels de qualité. Cette discipline doit établir un ensemble d'approches de développement, et de modèle de gestion gérant le processus logiciel. Du côté qualités requise, on peut citer les plus exigées comme suit :

- Fiabilité ;
- Efficacité : le logiciel doit utiliser rationnellement ses ressource : ne doit pas consommer trop de mémoire, ni gaspiller ses cycles d'horloge ;
- Interface appropriée : l'interface doit être facile à utiliser, et à apprendre ;
- Maintenabilité

Chapitre II :

Approches de développement
&
Modèles de Gestion de Projet Logiciel

1. Introduction

L'objectif de cette partie du cours est de présenter, en grosso-modo (sans trop de détails), les approches de développement et les modèles de gestion de projets logiciels, les plus reconnus dans le monde de génie logiciel. L'accent est, surtout, mis sur l'approche (ou le modèle) de la cascade.

2. Approche de la cascade

2.1. Naissance et idée de cette approche

C'est la première approche proposé en 1969 par Royce (donc connu sous le nom de Modèle de Royce). Cette approche est basée sur les principes suivants :

- Le processus est considéré comme une suite séquentielle d'activité ;
- Chaque activité a une date début et une date fin prédéfinies ;
- Chaque activité une fois terminée, **délivre** (fournit) un **délivrable** (un document) à l'activité qui suit : pour cette raison on parle aussi d'une approche à base de **délivrable**.

2.2. Schémas du modèle : activités

L'approche est schématisée sur la figure 1.

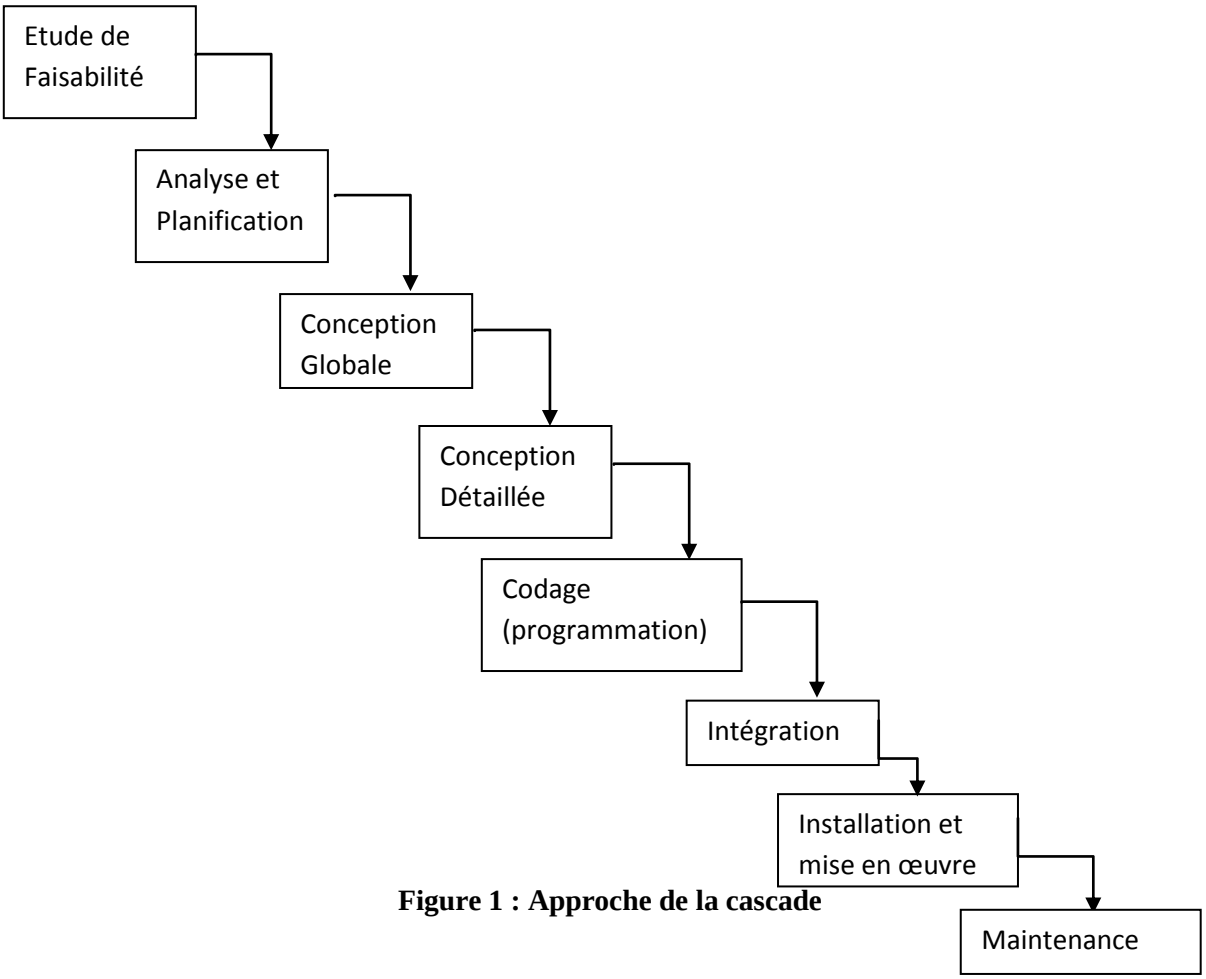


Figure 1 : Approche de la cascade

- **Etude de faisabilité:** dans cette activité, les développeurs (en général des ingénieurs système bien expérimentés et disposant de connaissances profondes et des compétences multidisciplinaires) commencent par se poser des questions comme : est ce que le système est réalisable ou non ? (on donne toujours l'exemple d'un logiciel pour la détection des prochains tremblements de terre qui reste comme un projet trop ambitieux mais non faisable), quelles sont les parties soft de celles hard que le système doit comporter ?, et enfin des décisions seront prises sur l'achat de composants nécessaires, le développement d'autres composants, les approches de développement à suivre, …
- **Analyse et planification**:

Analyse: il s'agit de l'analyse des besoins du client ou utilisateur du futur logiciel.

Objectif de cette activité : de connaitre les objectifs et les buts du client. En général les buts et objectifs sont des énoncés louable (et donc non mesurables : par exemple : joli, rapide, efficace, …), l'analyste doit donc transformer ces buts et objectifs en des besoins mesurables (vitesse de lancement<4 seconde, interface avec la couleur bleue, …). Donc cette activité doit identifier une liste de besoins. Il s'agit donc de répondre à la question **Quoi** ? et non pas à la question **comment** ?.

Personnel requis : les analystes (des développeurs), le client, l'utilisateur, le chef projet logiciel, un maitre d'ouvrage pour rédiger le cahier de charge, …

Outils utilisés : en général dans cette activité, les analystes font appel à des questionnaires, des entretiens, des interviews avec leurs clients et utilisateur pour faire sortir le maximum de besoins et pour se rapprocher de ce que les clients souhaitent avoir dans leur logiciel.

Planification: c'est de planifier le reste du processus. Des gestionnaires travaillent dans cette activité pour préparer le budget, le calendrier, ainsi que les moyens et personnel du processus.

- **Conception globale**:

Objectif de cette activité : définir l'architecture globale du système, et donc identifier les services que le système doit offrir comme réponses aux besoins identifier en analyse. Une liste de services est identifiée à ce niveau. Le système est décomposé en un ensemble de sous-systèmes. Un sous-système sera composé d'un ensemble de modules, et chaque module peut contenir plusieurs unités.

Personnel requis : des concepteurs (des programmeurs connaisseurs d'outils de description de haut niveau, des langages de modélisation, …).

Outils utilisés : langage de modélisation : DFD (diagramme de flux de données), DES (diagrammes entités association, diagrammes de contrôle, UML (Unified Modelling Language), …

- **Conception détaillée**: Pratiquement c'est une phase de raffinement et d'enrichissement des résultats obtenus auparavant. Cette activité doit aboutir à offrir une description assez détaillée du logiciel en construction. Cette description doit être trop proche d'une implémentation, et donc faciliter la tâche de programmation. La description finale peut être sous forme d'un ensemble d'algorithmes détaillés avec des structures de données (dans le cas d'une conception fonctionnelle). Dans cette activité, on utilise souvent des langages de description de programmes comme le langage inspiré de ADA (l'un des langages de programmation offrant une haute lisibilité et clarté des codes).

- **Codage (programmation)**: A ce niveau, il s'agit de choisir un langage de programmation, des environnements de développement, et de commencer le codage (l'implémentation) de la conception détaillée offertes par l'activité précédente.

- **Intégration**: En effet, dans les grands systèmes, le systèmes est composé de plusieurs sous-systèmes qui peuvent être développés, conçus et programmés par différentes équipes et en parallèle. A la fin de la programmation de tous ces sous-systèmes, la tâche d'intégration de tous ces sous-systèmes est importante pour former une seule unité. L'intégration nécessite, surtout, de définir les différentes interfaces entre les différents sous-systèmes réalisés.

- **Installation et mise en œuvre**: L'installation consiste à prendre le système (déjà développé sur la machine de la compagnie de développement), et de le mettre dans la machine du client, et ainsi de le configurer avec cette nouvelle configuration. La tâche d'installation doit rendre le logiciel opérationnel vis-à-vis la configuration matérielle et logicielle du client. Après cette installation, le client commence l'exploitation de son système (donc c'est la mise en œuvre), et de là le client explore les services qu'il souhaite retrouver, et éventuellement des erreurs commencent à être détecter.

- **Maintenance**: elle a pour objectif général de garder le système opérationnel, le plus longtemps possible. Cette activité est une activité permanente. On peut distinguer entre trois types de maintenance :

    - **Corrective** : dont l'objectif est de corriger les erreurs découvertes ;

    - **Adaptative** : dont l'objectif est d'adapter le système aux nouvelles technologies matérielles et logicielles ;

- **Perfective** : dont l'objectif est d'améliorer le système est d'augmenter ses performance.

2.3. Critiques de cette approche

L'approche de la cascade est la plus ancienne. Elle a reçu certaines critiques et plusieurs améliorations et remédiassions ont été proposées. Parmi les critiques:

- Un modèle trop idéal et ne reflète pas la réalité : les activités nettement séparées, sans retour en arrière, sans chevauchement. En réalité, ces activités ne sont pas assez séparées ;
- La gestion des délivrables: ces documents exigent des lectures, des corrections, des mises à jour. Ces opérations consomment du temps, du personnel et de l'argent aussi.
- Apparition des délivrables artificiels : quelques fois les développeurs risque de produire des délivrables artificiels pour seulement présenter de rapports à leurs gestionnaires, sans que ces délivrables reflètent le développement réel.

Pour répondre à ces critiques et à d'autres, les chercheurs ont proposé d'autres approches de développement ainsi que d'autres modèles de gestion de projets logiciels.

La proposition de nouveaux modèles de GPL et de nouvelles approches sont aussi motivées par des données empiriques qui ont montré la distribution des coûts sur les différentes phases de développement et pour différents types de système. Le tableau ci-dessous résume une étude réalisé après plusieurs années de développement de logiciels en utilisant la cascade classique.

Systèmes\ ctivité	Analyse et Conception	codage	test
Système de contrôle	16%	20%	44%
Aéronautique	34%	20%	46%
Système d'exploitation	33%	17%	50%
Système de Gestion	44%	28%	28%
Système de calcul scientifique	44%	26%	30%

Tableau 1 : Distribution des coûts de développement sur les activités du processus logiciel

Le tableau 1 montre que dans les systèmes de contrôle, aéronautique et d'exploitation, les activités de test sont les activités les plus gourmandes en termes de coût. Ce coût concerne les trois niveaux : Personnel, budget et calendrier. Les systèmes de gestion sont plutôt exige plutôt une phase d'analyse plus coûteuse car il est souvent important de passer une longue période en la récupération des données. En général, les activités de codage sont les activités les moins couteuses par rapport aux autres activités et ceci pour tout genre de système.

Suite à ces résultats, les chercheurs ont proposés différentes amélioration à la cascade pour couvrir les limites et les surcoûts des différentes activités.

3. Représentation en V (représentation améliorée de la cascade)

3.1 Principe

Une autre représentation de la cascade est le modèle en V. Ce modèle mets l'accent sur certains aspects non présents dans la représentation de la cascade :

- Activité de test : avec différente variante ;
- Relation entre les différentes activités

Dans cette représentation, on distingue entre deux types d'activités :

1) Les activités constructrices : dont l'objectif est de développer le système. On parle de construction du système aussi : Faisabilité, Analyse, Conception, et codage

2) Les activités destructrices : dont l'objectif est de trouver les erreurs commises durant les activités de construction. Donc, c'est une forme de destructions du travail construits. En effet, il s'agit d'une suite d'activités de test qui réussissent quand elles identifient des erreurs dans les produits des premières activités : Test unitaire, test d'intégration, test d'acceptation, et enfin le test système

La représentation en V montre explicitement les relations entre ces différents types d'activités: les premières activités de construction préparent les dernières activités de test. Par exemple, ce qu'on doit faire dans un test système doit être préparé et planifiée durant la phase d'analyse elle-même.

3.2 Schéma du V

La figure 2 montre le modèle en V avec ses différentes activités

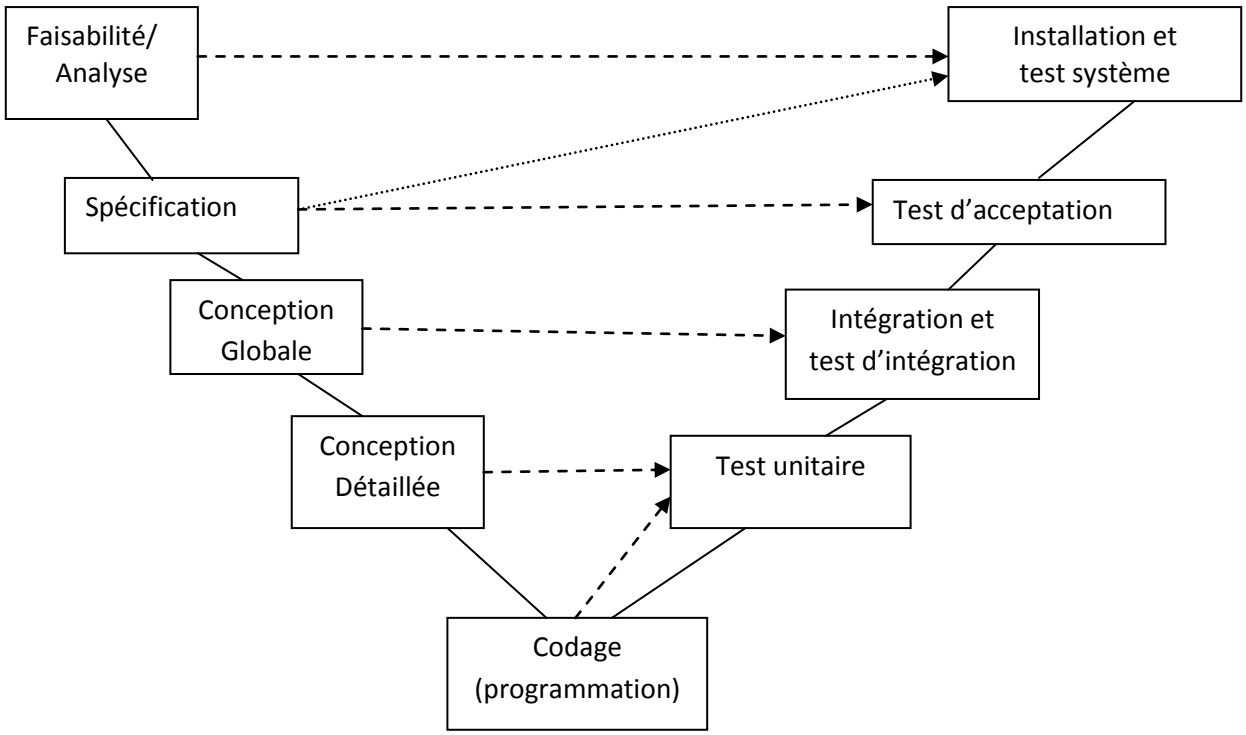


Figure 2 : Représentation en V

Dans ce schéma, certaines activités sont incluses :

1) L'activité de spécification : dans cette activité, les analystes ou ***spécifieurs*** prépare une spécification des besoins. Une spécification doit être une description assez concise, précise, sans ambigüité. Une telle spécification peut être même formelle (décrite dans des langages mathématique et logique si le

système le nécessite). Une spécification diffère d'un cahier de charge en termes de précision.

2) Le test unitaire : cette activité doit permettre de tester les unités du logiciel de façon séparée, unité par unité. Elle est liée aux activités de codage et de conception détaillée.

3) Le test d'intégration : cette activité s'intéresse à tester les unités intégrées et leurs interfaces de communication. Elle est liée à la conception globale du système.

4) Le test d'acceptation : il est réalisé à la présence du client chez la compagnie du développement et sur sa configuration matérielle et logicielle.

5) Le test système: il est réalisé à la présence chez le client chez sur la configuration matérielle et logicielle cible.

Remarque : dans cette représentation, les arêtes décrivent l'ordre du déroulement d'activités, et les arcs discrétisés décrivent les relations de préparation et de dépendance entre activité de développement de test.

4. Modèle des incréments

Objectifs:

L'idée de ce modèle est de considérer que les différentes activités du processus logiciel peuvent être chevauchées (ou mise en pipeline) quand elles n'exigent pas les même entrées ni les mêmes personnes. La phase d'analyse et la phase de conception peuvent chevauchées si cette conception n'a plus besoin des résultats de l'analyse courante. Ce chevauchement d'activité permet de rendre le processus logiciel semi-parallèle ou en pipeline. Ceci permet, certainement, de minimiser le temps de développement.

Principe: chevaucher (paralléliser ou mettre en pipeline) les activités qui peuvent être faites en même temps.

Étapes:

En grosso-modo, les étapes d'un tel développement se présentent en quelques simples étapes consistant en :

1) Une **première phase d'analyse** qui doit identifier le **noyau du système** et un ensemble de **composants proportionnellement** indépendants;

2) Réalisation complète du noyau du système;

3) Le reste des composants se réalisent en parallèle, autant que ceci est possibles (moyens disponibles, pas de relations causales entres activités, etc);

4) Chaque composant est intégré une fois terminé avec les composants déjà intégrés.

La figure 3 montre une représentation de ce modèle et comment les constructions des différents composants sont chevauchées.

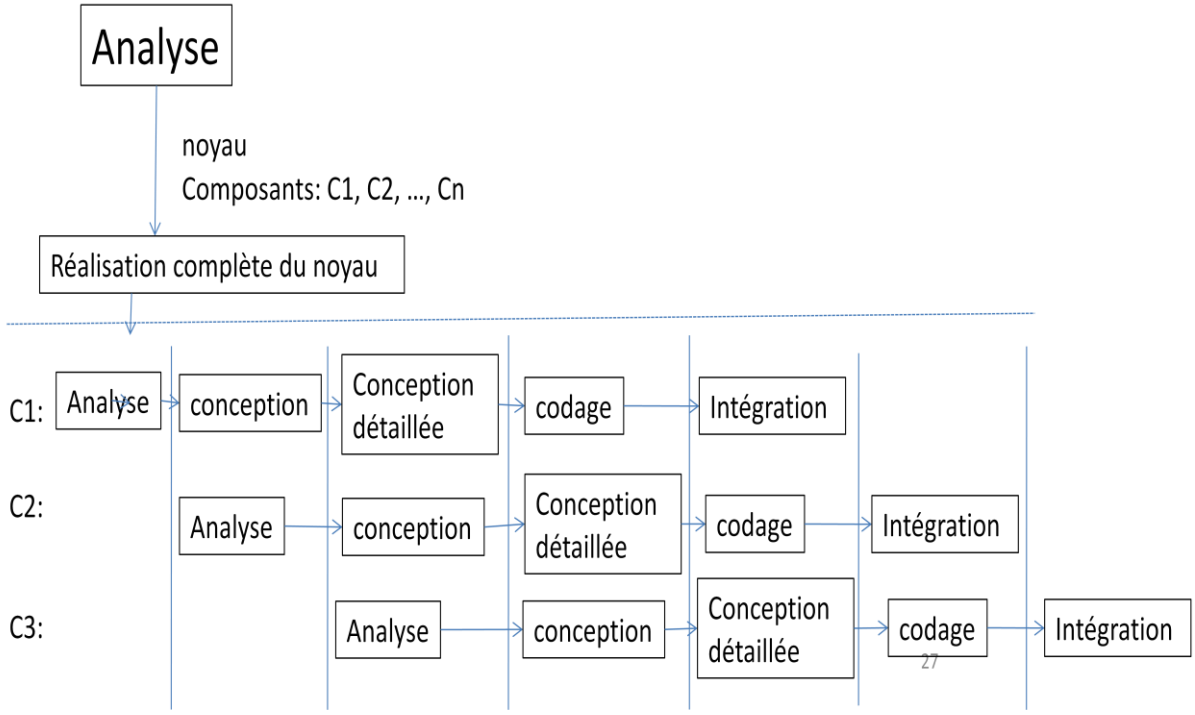


Figure 3 : Modèles des incréments

Avantage:

En plus du gain en temps de développement, le modèle offre les avantages suivant :

- Possibilité d'avoir des livraisons partielles;
- Une bonne gestion de l'équipe

Limites:

Cependant, les incréments peuvent être limités pour les raisons suivants :

- Les systèmes ne sont pas toujours décomposable (facilement);
- L'intégration peut poser souvent des problèmes.

5. Autres approches de développement

A l'opposition des modèles de gestion de projet logiciel, les approches de développement représentent des démarches, plutôt théoriques, dont l'objectif est de fournir une liste d'activité nécessaires pour réaliser un logiciel. Ces approches ne sont pas trop accentuées sur l'aspect gestion et ne donnent pas trop de détails. Ces approches peuvent être combinée et peuvent même être appliquées ensemble dans le processus logiciel et dans de le même modèle de GPL.

Dans la littérature, il existe cinq majeures approches : Approche de la cascade (liée directement au modèle de la cascade, déjà présenté dans la partie précédente), l'approche basée sur le prototypage, l'approche basée sur la programmation exploratoire, l'approche de réutilisabilité, et enfin l'approche de transformations formelles. On examine ci-dessous, les quatre approches qui n'étaient pas encore présentées.

5.1 Approche basée sur le prototypage

Objectif:

Afin d'éviter les mal-compréhensions des descriptions des besoins effectuée, souvent, en langage naturelle, l'approche de prototypage propose l'usage de prototypes logiciels. Un prototype peut être vu comme une version initiale réalisée rapidement pour **explorer** les besoins de l'utilisateur à travers et **faciliter** la phase d'analyse. Le prototype peut traiter plusieurs aspect d'un logiciel : aspect structurel (interface), aspect dynamique (fonctionnalités), …etc. On distingue entre deux types de prototypes : Jetable qui servira qu'à l'exploration et sera jeté une fois l'analyse est achevée, et Evolutif : qui peut évoluer de version initiale vers une description très complète, ainsi un logiciel complet.

Étapes:

Les étapes se résument comme suit :

- Découvrir les besoin de l'utilisateur via un prototype.
- Ce prototype sera rejeté (**prototype jetable**), une fois les besoins sont identifiés.
- Le prototype peut évoluer (**prototype évolutif**) pour devenir lui-même le système final

Limites:

Cette approche peut poser certains problèmes aux développeurs comme :

- Elle exige des **outils de prototypage spécifiques** (comme les anciens langages de prototypage dits : langages de 4ème génération)
- Quelques fois **le prix du prototype** devient aussi important que le prix du système lui-même.

Exemple et discussion :

A titre d'exemple, avec des langages modernes supportant le développement rapide des interfaces dans les environnements de développement rapide (RDE), cas du C++ builder, Delphi ou Java, les programmeurs peuvent rapidement prototyper les interfaces des systèmes : informatiques, mécaniques ou électroniques. En plus de ces interfaces, la dynamique de ces systèmes peut être simulée avec des procédures, méthodes ou des interactions objets.

La figure 4 montre un vrai distributeur de billets automatique, qui est un système hybrides combinant plusieurs compétences : informatiques, mécaniques, et automatiques. La figure 5 monte un prototype rapide de l'interface de DBA, réalisé en quelques minutes avec un langage comme Delphi, C++ Builder ou Java.

Figure 4 : Un distributeur de billet automatique

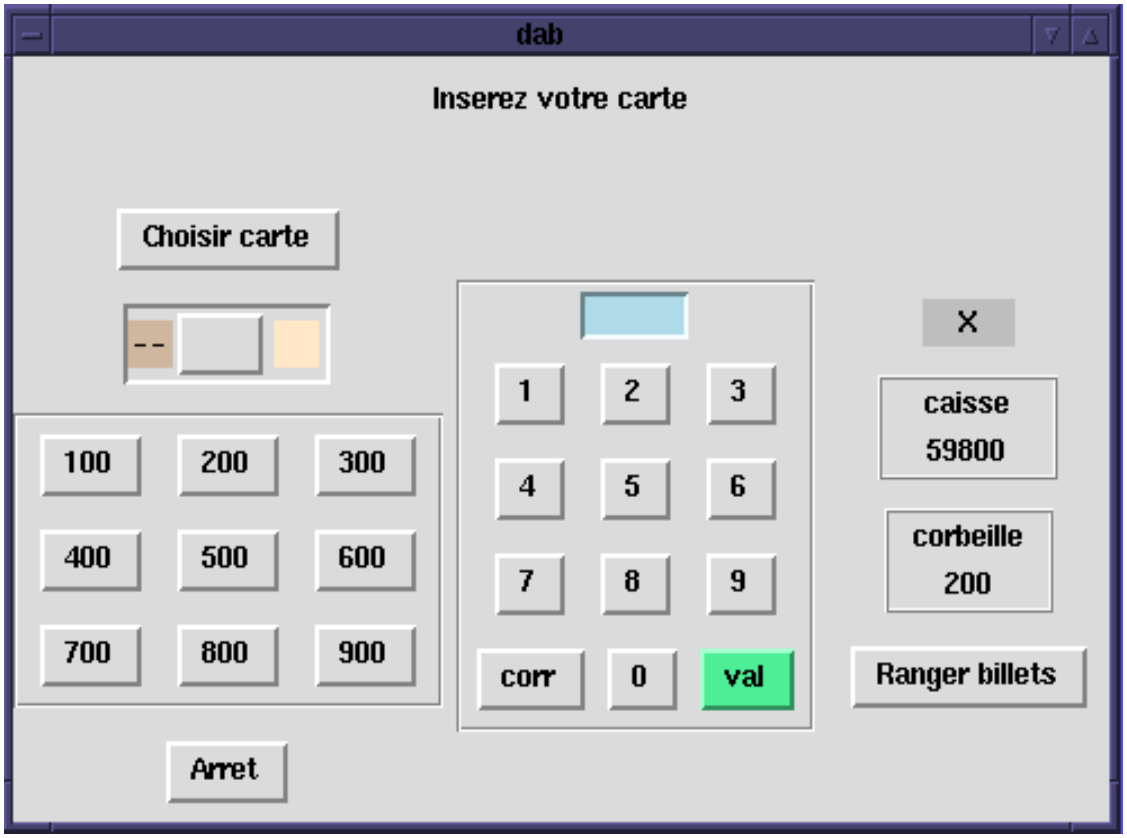


Figure 5 : Un prototype rapide et simulateurs d'un distributeur de billet automatique

5.2 Approche basée sur la programmation exploratoire

Objectif:

L'objectif de cette approche de présenter rapidement une version opérationnelle au client. Il s'agit d'anticiper le développement et d'aller directement au codage afin de voir l'adéquation des idées faites par le développeur avec celles des clients. La programmation exploratoire vise à explorer les besoins du client via la programmation plutôt que des interviews ou des questions en analyses.

Étapes:

Les étapes se résument comme suit :

1) Réaliser un résumé de la spécification;
2) Programmer la spécification;
3) **Si** le programme satisfait l'utilisateur **alors**
   Livraison du logiciel

Sinon Aller à 2

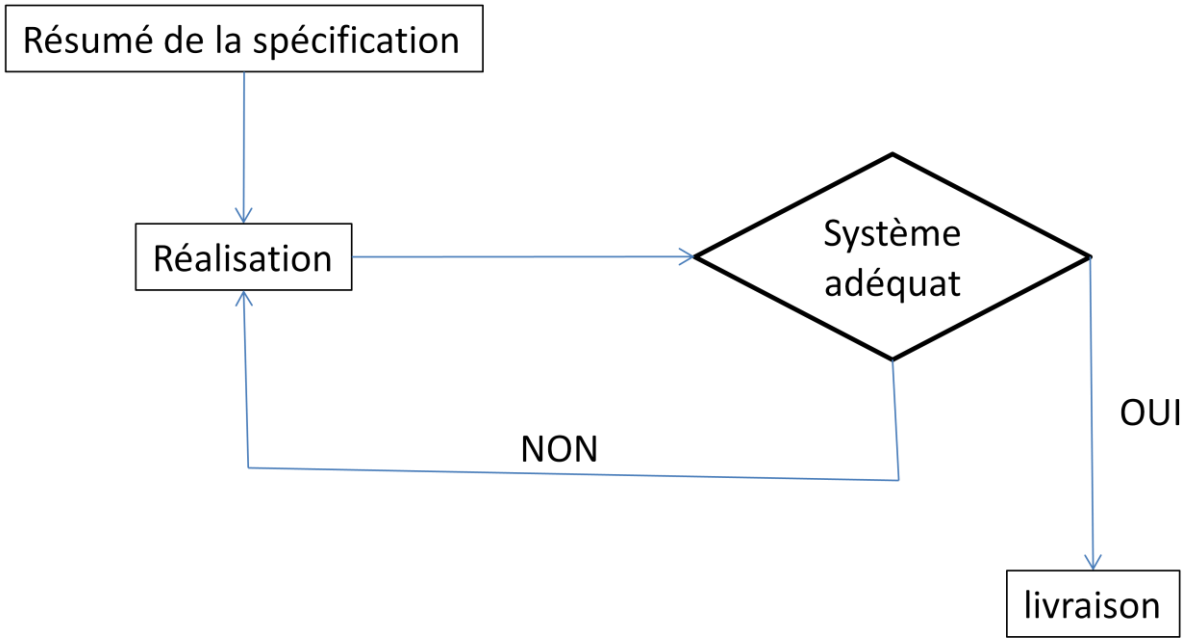


Figure 6 : La programmation exploratoire

Limites de cette approche: malgré son efficacité apparente, cette approche peut rencontrer certains problèmes pour être appliquée. Les deux inconvénients majeurs de cette approche sont donc :

- Elle exige des **langages particuliers** (le logiciel change trop); Donc, des langages classiques comme le C, Java ou tout langage impératif ne peut pas être exploités ici.
- Problème de gestion de l'équipe: cette approche nécessite une petite équipe. donc trop de personnes seront mises en **chômage**.

En effet, cette approche est plutôt dédiée aux systèmes experts de l'intelligence artificielle, où la capacité du système peut accroitre en ajoutant de la connaissance sous forme de règles logiques. L'ajout de ces règles peut être fait à n'importe quel moment et leurs ordre n'influence pas sur le fonctionnement du système.

5.3 Approche basée sur la réutilisabilité

Objectif:

Dans un système logiciel compliqué, il souvent impossible de réaliser tous les composants à partir de zéro. L'informatique moderne favorise l'usage et le ré-usage des composants déjà réalisés et certifiés par des compagnies ou des organismes. Il est toujours préférable de profiter du nombre de composants et codes disponibles dans le marché informatique qu'ils s'agissent de freeware (des composants gratuits) ou de shareware (des composants payant).

En effet, en utilisant l'approche de réutilisabilité, on s'attend à une baisse importante dans le **coût** et le **temps** de la réalisation. Cette régression dans le temps et le coût est due à l'évitement de plusieurs phases incorporant toutes les activités d'analyse, conception, codage et test.

Étapes:

Le système n'est pas réalisé à zéro, mais il est **composé de plusieurs** autres composantes déjà existantes. La figure 7 montre une simple intégration de plusieurs composants pour bâtir un seul système. Ces composants sont considérer disponibles et déjà implémentés, testés, et exploités ailleurs.

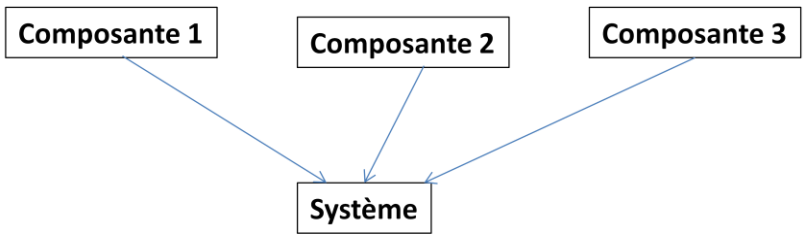


Figure 7 : La programmation exploratoire

Limites:

Autant qu'elle offre plusieurs avantages, cette approche peut avoir de grand problème dans deux aspects très cruciaux :

- Disponibilité et coût des composants? Ces composants peuvent être très coûteux, que le développeur préférera les réaliser lui-même.
- Intégration des composants. Des composants qui viennent de plusieurs fournisseurs peuvent poser des problèmes d'intégration s'ils ne sont pas tous soumis aux mêmes standards.

5.4 Approche basée sur les transformations formelles

Objectif:

En effet l'un des problèmes majeurs de la crise logicielle était le manque de fiabilité des logiciels. Ce manque de fiabilité se manifestera en des bugs qui ont marqué le développement logiciel durant des années. Il était toujours considéré que ce manque de fiabilité est du aux langages utilisés dans la spécification des besoins des utilisateurs. Ces spécifications sont souvent écrites en langages naturels (français, anglais, …). Les langages naturels se caractérisent par : l'ambigüité, la confusion, le manque de clarté, … ce qui peut causer des omissions de besoins, leur complétudes, et même leurs contradiction et leurs ambigüité dans les phases ultérieures à la phase d'analyse. Pour palier à ces limites des spécifications en langages naturels, les chercheurs proposent l'usage des langages de spécification et de modélisation formelle. Ces langages sont considérés plus convenables pour éviter les limiter des langages naturels. Les langages formels sont basés sur les fondements mathématiques et logiques et peuvent être utilisés pour décrire rigoureusement les besoins d'un système, vérifier et valider ces besoins, raffiner ces besoin et identifier des services les assurant, raffiner ces services successivement jusqu'à aboutir à un code complet du système. Ces raffinements successifs sont dits aussi Transformations formelles qui mènent d'une représentation à une autre représentation.

L'usage d'une approche de transformations formelle est supposé garantir d'avoir des systèmes **fiables** et **corrects**. Ainsi, il est possible d'utiliser la **preuve** de correction. La preuve de correction permet donc de m**inimiser** le coût des tests.

Étapes:

Les étapes peuvent se résumées comme suit :

1. Réaliser une spécification initiale S0 à partir des descriptions informelles;
2. Transformer cette spécification en une autre spécification S1, en appliquant une transformation T0
3. Plusieurs transformations seront faites jusqu'à aboutir au programme final.

La figure 8 montre une représentation graphique du processus des transformations formelles

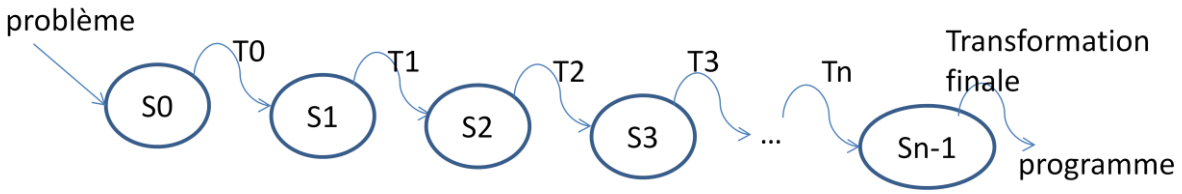


Figure 8 : Les transformations formelles

Avantages:

En plus des objectifs de cette approche qui sont de garantir la fiabilité des systèmes réalisés, il est que cette approche offre les avantages suivants :

- Possibilité d'automatiser tout le processus de transformation et donc minimiser les interventions humaines : gain du temps et de fiabilité (manque d'erreurs humaines en transformations) ;
- Ces approches sont les plus adéquats, surtout, pour les systèmes critiques (où la panne est non permise)

Limites:

Malgré les avantages de cette approche, elle souffre de deux problèmes qui limitent son large usage en génie logiciel :

- Elle exige des compétences spécifiques dans les domaines liés à la vérification formelle: **preuve**, **logique**, **algèbre,** etc.

- Elle peut être une approche couteux et temps et en effort personnel dans les activités de **rédaction** et **preuve** des spécifications.

6. Pourquoi une variété d'approches

Il est important d'avoir une telle variété d'approches et de modèles pour plusieurs raisons. Cette variété se justifie par les motivations suivantes :

1. **Pas** d'approche idéale : Comme était présenté, chaque approche a ses points forts et ses points faibles. D'où, aucune approche ne peut assurer la réussite du processus logiciel de manière absolue.

2. À chaque **système** ses **spécificités** et ses **qualités** : Il est claire qu'un système d'exploitation favorise une approche de réutilisabilité, alors qu'un système critique préfère une approche de transformation formelle, ainsi qu'un système de gestion se trouve mieux placé dans une approche de la cascade, etc.

3. Les **compétences** de l'équipe et les **outils** et les **moyens** de développement: le choix de l'approche est souvent guidé par ce qu'on a comme : compétence personnel, comme outils de développement et comme moyen financier.

7. Le génie logiciel dans la réalité : Industrie logicielle

En pratique, les approches de développement déjà présentée restent plutôt dans un cadre académique et pédagogique. L'industrie logiciel était souvent artisanal et basée sur des acquis expérimentaux et empiriques plutôt que sur des formations purement théoriques et académique. Pour cerner la situation dans l'industrie logicielle, en 1991 et aux états unis, le DoD (department of defense) a proposé 5 niveaux de maitrise dans le développement de logiciel. Ces 5 niveaux sont définis comme suit :

1) **Niveau initial: aléatoire (processus non contrôlé).** Ici, le développeur ne peut ni contrôler le délai, ne le coût ni la fiabilité.

2) **Niveau artisanal:** le développeur peut contrôler le délai.

3) **Niveau défini:** le développeur peut contrôler deux paramètres uniquement : délais & fiabilité ou délais & coût.

4) **Niveau géré:** tout est contrôlé.

5) **Niveau optimisé:** un stade très avancé ou la compagnie fait des recherche dans le processus logiciel, proposer de nouvelles approches, …etc.

En faisant des statistiques sur les compagnies de développement de logiciels eu EU, ils ton découvert que : 85% des développeurs sont dans le niveau 1, 15% dans le niveau 2, et de rares (des exceptions) qui sont au niveau 3.

Ces résultats restent valides même après plusieurs années de ces premières statistiques. Les figures suivantes (obtenu du site de l'organisme CHAOS) schématisent des données plus récentes qui montrent les échecs –toujours- en occurrence dans l'industrie de logicielle. A noter que le mot « ***challenged*** » veut dire le budget et le calendrier n'est pas respecté.

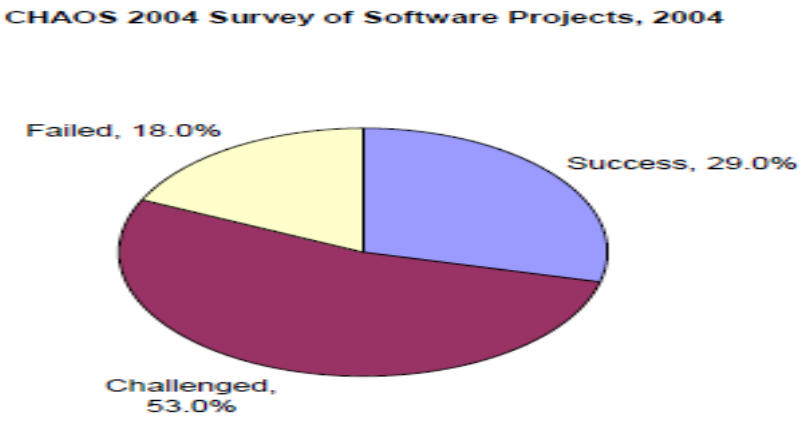


Figure 9 : Echec en production de logiciel 2004

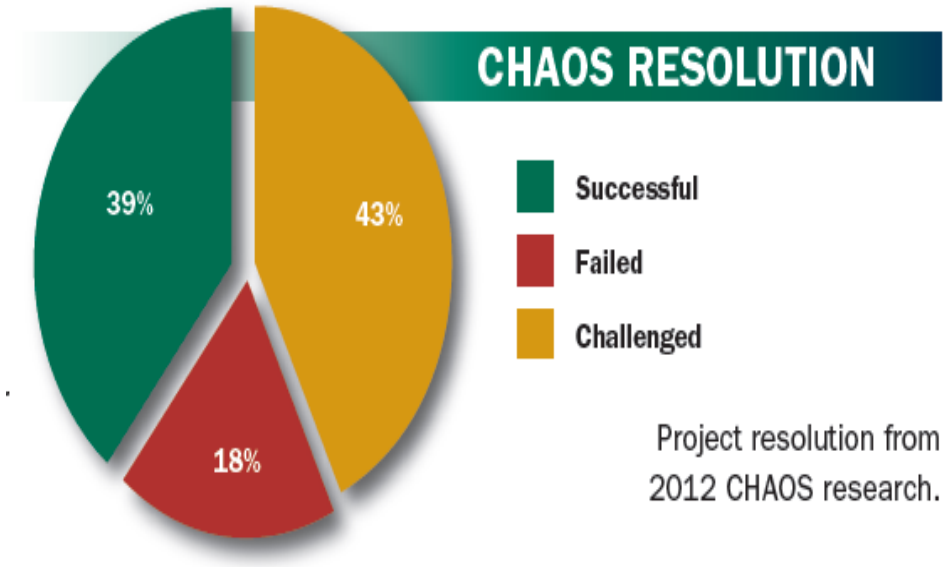


Figure 9 : Echec en production de logiciel 2012

RESOLUTION

	2004	2006	2008	2010	2012
Successful	29%	35%	32%	37%	39%
Failed	18%	19%	24%	21%	18%
Challenged	53%	46%	44%	42%	43%

Project resolution results from CHAOS research for years 2004 to 2012.

Figure 10 : Echec en production de logiciel 2004-2012

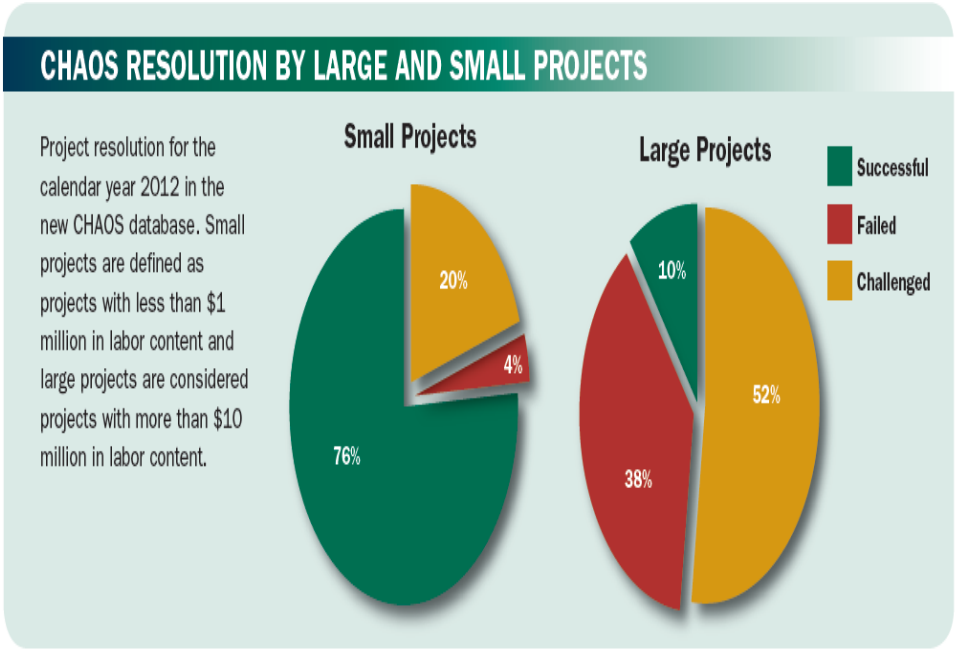


Figure 10 : Echec en production de logiciel (petit projet vs grand projet) 2012

8. Conclusion

Affin de développer d'assurer le développement de logiciels de qualité (fiable, maintenable, efficace, etc), les chercheur en GL ont proposé plusieurs approches de développement et des modèles de gestion de projets logiciels (GPL). Le modèle le plus intuitif, simple et appliqué est celui de la cascade. Pourtant, ce premier modèle a montré ses limites en terme de : temps important de développement, retard de livraison de version préalable, découverte tardives des erreurs, consommation de ressources énormes en analyse et test, …D'autres approches et modèles ont été proposés pour pallier à ces limites.

L'existence d'une variété d'approches et de modèles se justifie par les différents types de systèmes à développer, par les compétences, outils et moyens disponibles dans les équipes de développement, et par les qualités souhaités dans chaque système.

En pratique, et même selon des statistiques publiées dans les années récentes, le GL en pratique a toujours de grands défis à vaincre. Les logiciels souffrent toujours des problèmes dans leurs délais, leurs coûts finaux et leur fiabilité.

Chapitre III : Le langage UML

1. Naissance de UML

UML (Unified modeling langage) est un langage unifié de modélisation de systèmes orientés objet. Ce langage a connu sa vraie naissance en 1997, suite à l'unification des notations utilisées dans plusieurs langages de modélisation et approches de conception orientées objets.

Les figures 11 et 12 montrent la chronologie d'évolution des différentes notations et celles qui ont influencé le langage UML.

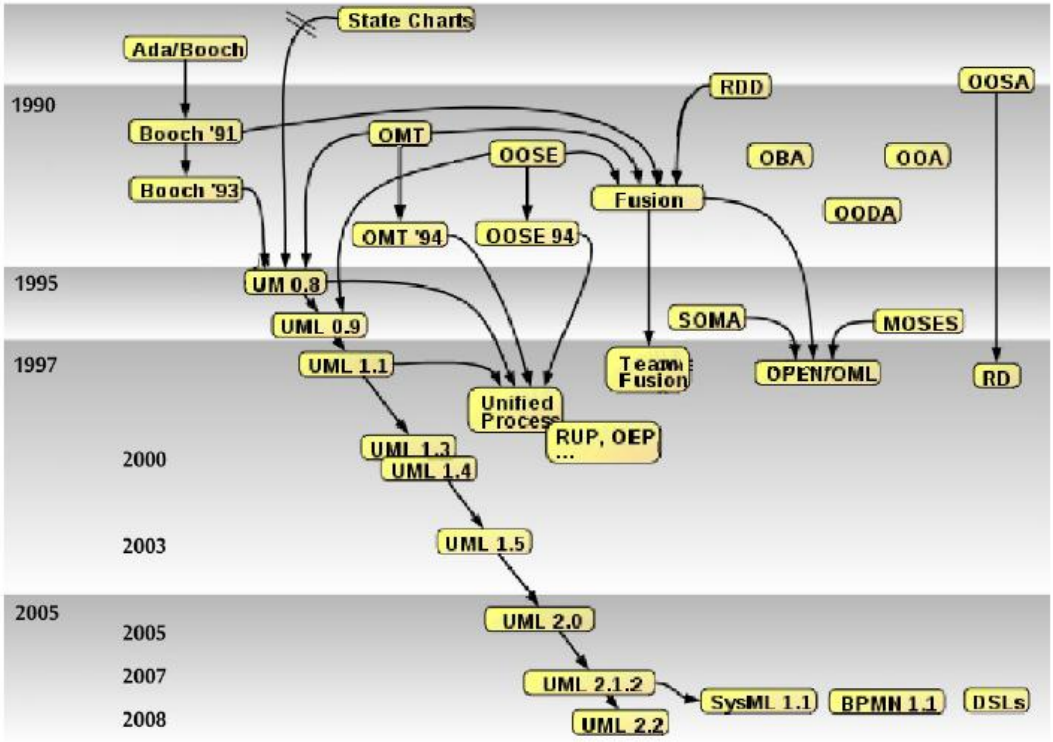


Figure 11 : Naissance et évolution d'UML vers 2008

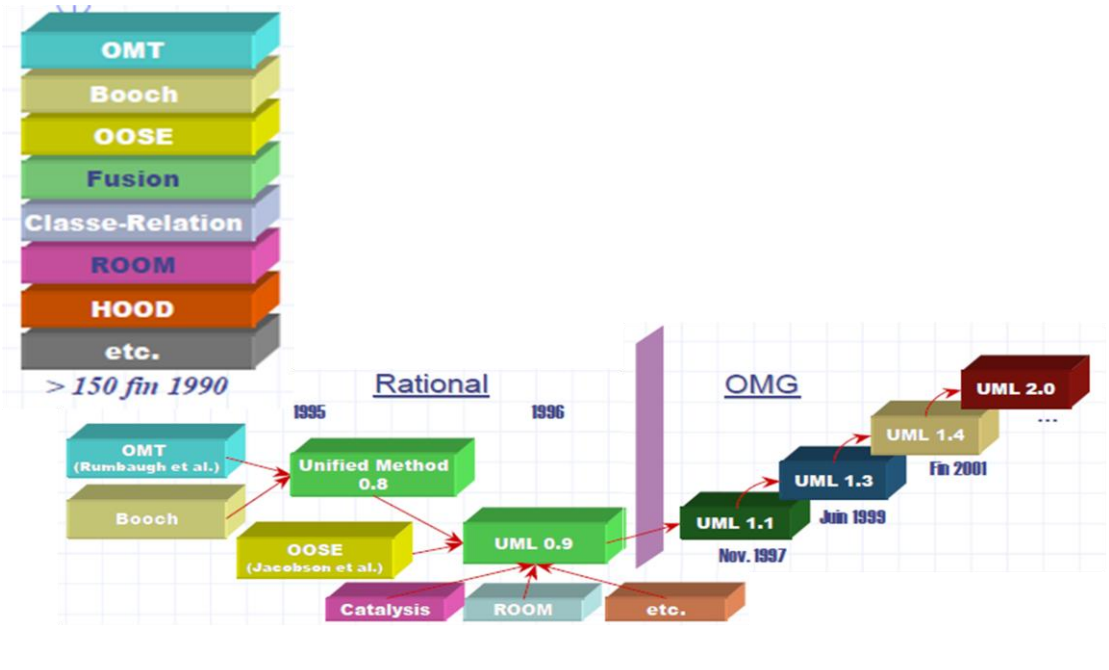


Figure 11 : Unification des notations en UML

En effet, les trois approches et noations qui ont influencé UML sont : OMT (Object Management Techniques), BOOCH, et OOSE (Object Oriented Software Engineering), comme schématisé sur la figure 12. Actuellement, UML est maintenu toujours par OMG: http://www.uml.org/.

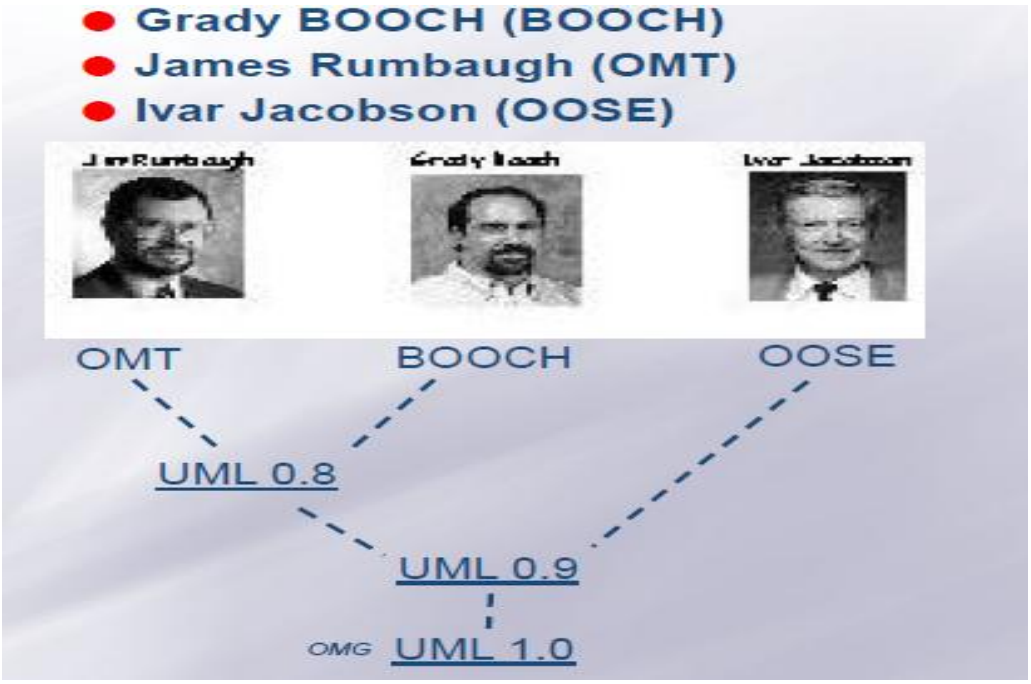


Figure 12 : de OMT, BOOTCH, OOSE vers UML

2. Définition de UML

C'est un langage visuel dédié à la spécification, la construction et la documentation des artefacts d'un système logiciel, [selon OMG].

Notez bien :

UML est un langage de modélisation et n'est pas une approche de conception, analyse ou de développement, car Elle **ne propose pas**:

- **un processus** de développement,
- **ni ordonnancement** des tâches,
- **ni répartition** des responsabilités,
- **ni règles** de mise en œuvre

Certains ouvrages basés sur UML ajoutent cet aspect fondamental en méthodologie)

3. Diagrammes de UML

UML est un langage de modélisation qui propose un ensemble de diagrammes qui peuvent être utilisés pour la description des activités d'analyse et de conception et des leurs résultats. Ces diagrammes peuvent être classés en trois grandes familles :

Modèles descriptifs vs prescriptifs:

- Descriptifs: Décrire **l'existant** (domaine, métier)
- Prescriptifs: Décrire le **futur** système à réaliser

Modèles destinés à différents acteurs:

- Pour l'utilisateur: Décrire le **quoi?**
- Pour les concepteurs/développeurs: Décrire le **comment?**

Modèles statiques vs dynamiques!

- Statiques: Décrire les aspects **structurels**
- Dynamiques: Décrire **comportements et interactions**

La figure 13 présente un récapitulatif de l'ensemble des diagrammes UML proposés pour la modélisation d'un système, de ses besoins et de son déploiement.

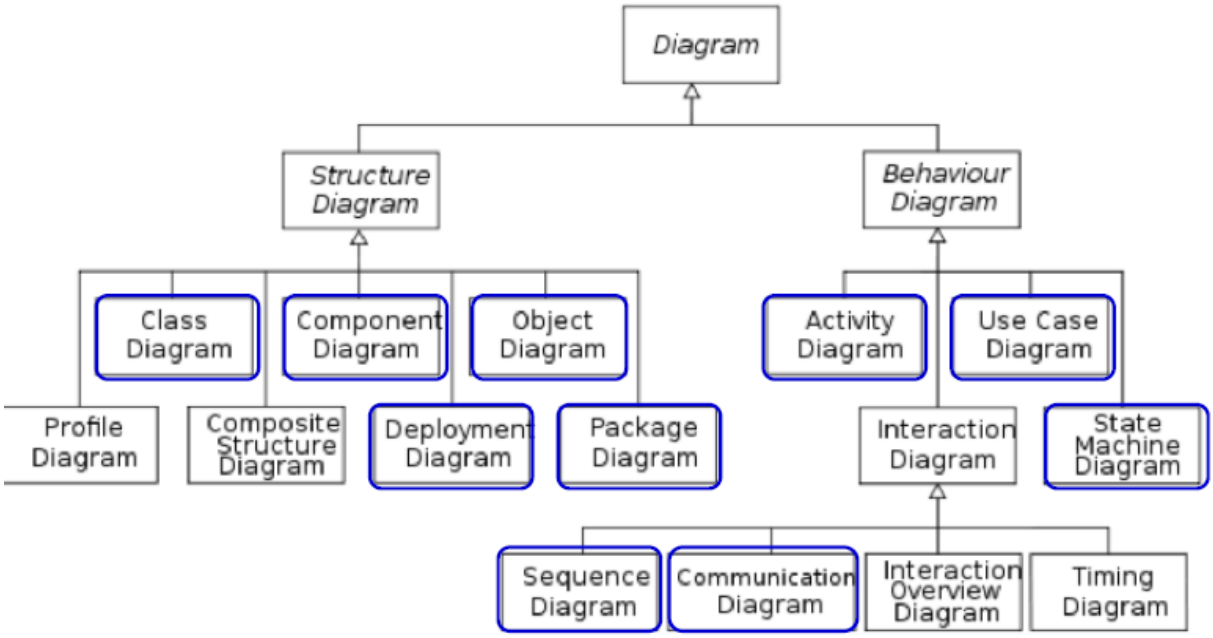


Figure 13 : les diagrammes UML

Les diagrammes les plus utilisés dans le processus de développement et qui seront présentés dans la formation académique en deuxième années sont les suivants:

Diagramme dédiés pour la collecte des besoins des utilisateurs:

- Diagramme des cas d'utilisation

Diagramme dédiés à la structure statique:

- Diagramme de classes
- Diagramme objet

Diagramme dédiés à la dynamique interne des objets:

- Diagramme états-transition
- Diagramme d'activités

Diagramme dédiés dynamique aux interactions entre objets:

- Diagramme de séquence
- Diagramme de collaboration

Diagramme dédiés dynamique dédiés à la réalisation et au déploiement:

- Diagramme de composants
- Diagramme de déploiement

4. Démarche possible d'usage de UML

Même si UML n'est pas une approche de développement et il ne donne pas le processus d'application des différents diagrammes, mais on peut appliquer un processus ad-hoc pour rendre la modélisation plus efficace et rationnelle :

- Spécifier le système: use-cases
- Modéliser des objets communicants: objets, communication
- Modéliser la structure de l'application: Classes
- Modéliser le comportement des objets
- Modéliser les traitements
- Modéliser l'instanciation de l'application

5. Diagramme des cas d'utilisation : Use Case Diagram

5.1 Pourquoi ce diagramme ?

- Structurer les **besoins** des utilisateurs et les **préoccupations** "réelles" des utilisateurs;
- Montrer les **objectifs** du système;
- Identifier les **utilisateurs** (acteurs) + **interactions** (utilisateur-système);
- Vision générale sur les **fonctionnalités** du système

5.2 Concepts et Modélisation

Avec un UCD, l'analyse peut présenter les différents acteurs du système, leurs cas d'usage du système, les différentes relations entre cas d'utilisation et aussi les relations entre acteurs du système. Aussi, il est possible d'ajouter des commentaires, de créer des paquetages, etc. Ci-dessous, on présente les deux concepts clés utilisés en UCD : Use cases et les acteurs :

- **Use case**: **séquences d'actions** réalisées par le système et **produisant un résultat** observable intéressant pour un acteur particulier. **Exemple:** Achat billet, Ouverture compte, Livraison Client, Traiter commande, établir facture…

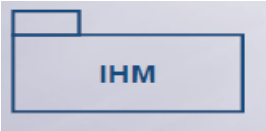
- **Acteur**: une **entité externe** qui **agit** sur le système (opérateur, autre système...). Qui peut **consulter** |**modifier** l'état du système. Le système fournit un service qui correspond au besoin d'un acteur. Les acteurs peuvent être **classés**

Des notations graphiques servent à faire cette modélisation comme présenté ci-dessous :

- Acteur: 
- Cas d'utilisation: 

- Commentaire : 

- Package:



5.3 Liens entre use cases et liens entre acteurs

Plusieurs types de liens peuvent exister entre les nœuds d'un UCD. Entre acteurs et use cases, on a des liens dits de communication; Entre cas d'utilisation, on trois types de lien: généralisation, inclusion, extension. Et enfin entre acteurs, on a deux types de liens : généralisation, extension.

Exemple 1 :

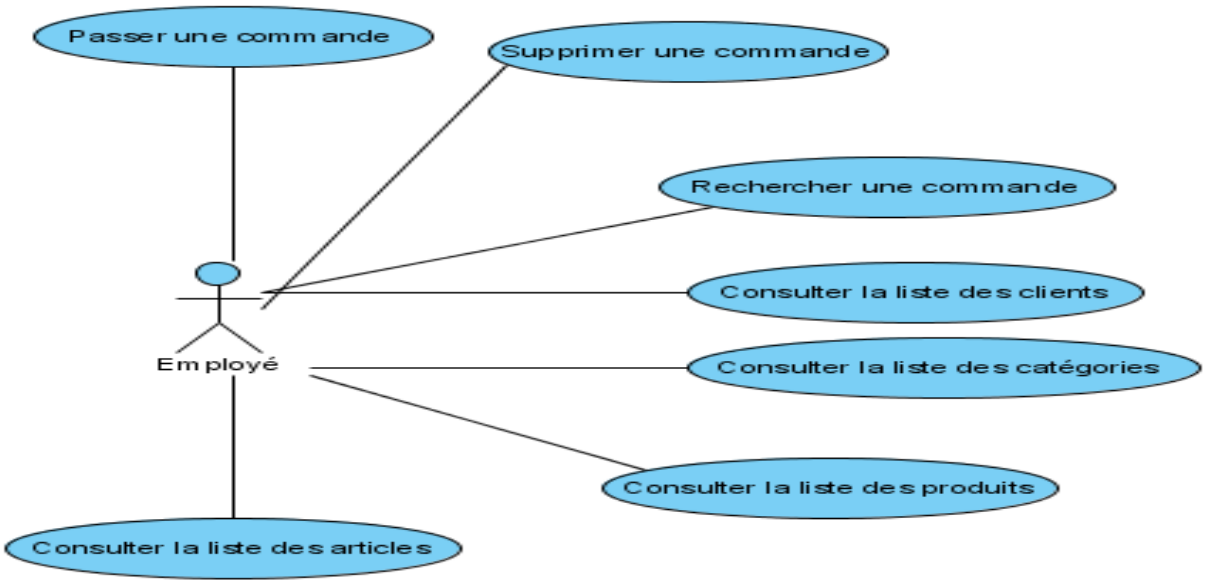


Figure 14 : Exemple 1 d'un diagramme use case

Sur l'exemple de la figure 14, un simple diagramme composé d'un seule acteur dit « employé » connecté à plusieurs cas d'utilisation : passer une commande, supprimer une commande, … etc.

Sur l'exemple de la figure 15, un diagramme UCD plus élaboré est présenté. Dans ce UCD, quatre acteurs sont définis : client, caissier, gérant et administrateur system. Quatre ***use cases*** sont présents : traiter une vente, ouvrir la caisse, traiter un retour, et gérer la sécurité. Chaque acteur est lié avec les cas d'utilsiation dont il est concerné par. En plus de ces composantes, le modèle présente deux autres acteurs spécifiques : service d'autorisation des paiements et système des ressources humaines.

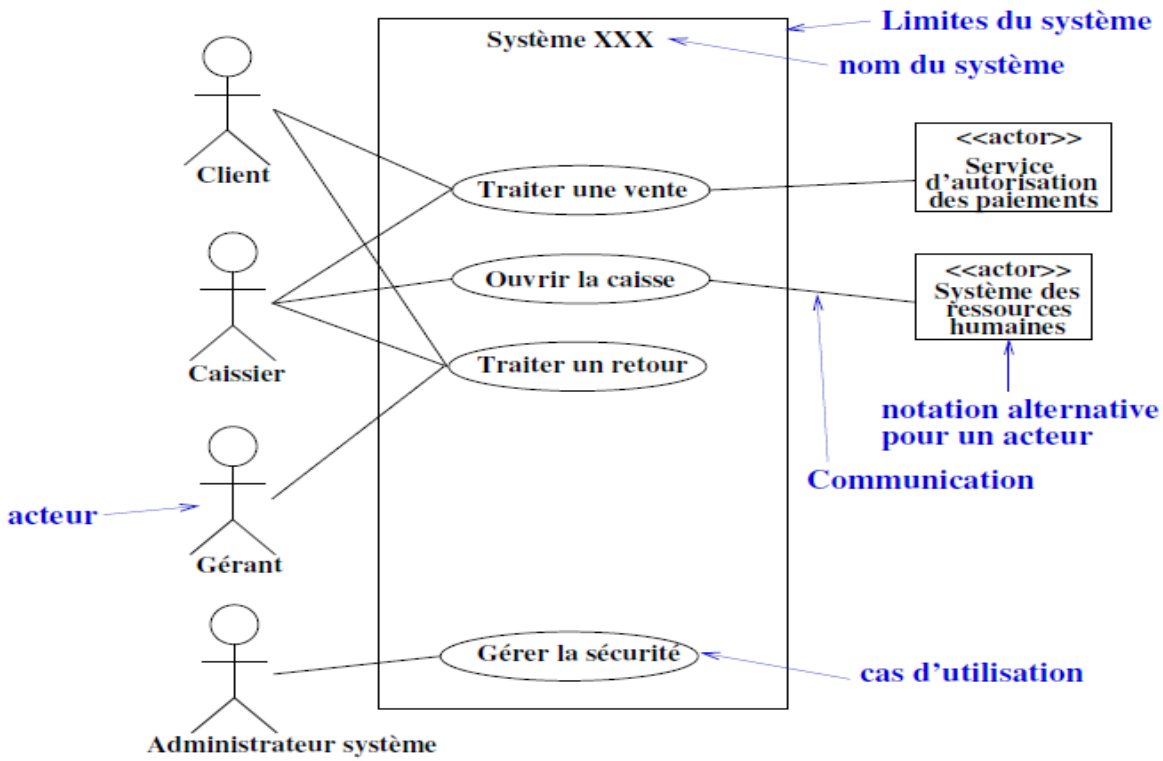


Figure 15 : Exemple 2 d'un diagramme use case

La figure 16 montre une relation de généralisation possible entre deux acteurs.

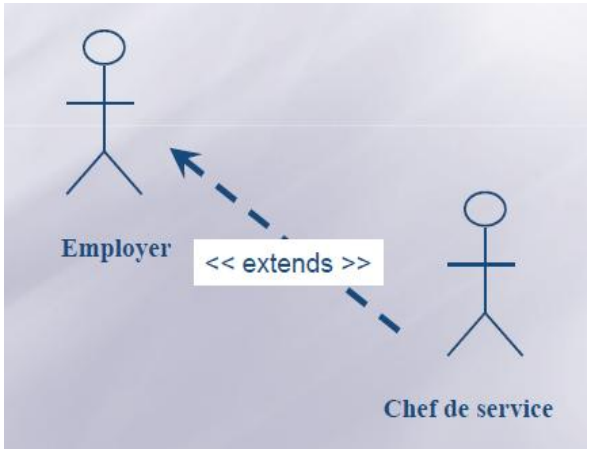


Figure 16 : Exemple 1 de relation entre acteurs

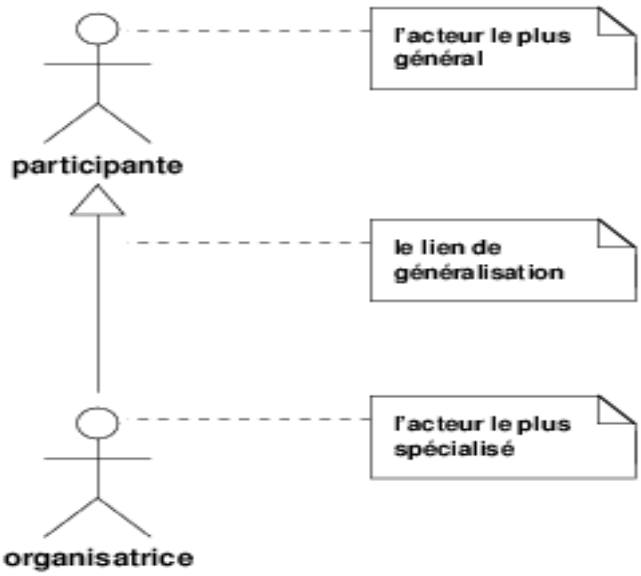


Figure 17 : Exemple 2 de relation entre acteurs : généralisation

Sur la figure 16, l'acteur employé est vu comme une généralisation de l'acteur chef service, ou l'acteur chef service est vu comme une spécialisation de l'acteur employé. Une telle spécialisation permet à l'acteur chef service de disposer de toutes les capacités de l'acteur employé et encore d'avoir plus de compétence ou de savoir faire. Sur la figure 17, la relation de généralisation est présentée encore avec trois acteurs qui se relient entre eux.

Sur la figure 18, un modèle UCD est présenté avec une relation de généralisation entre deux acteurs :

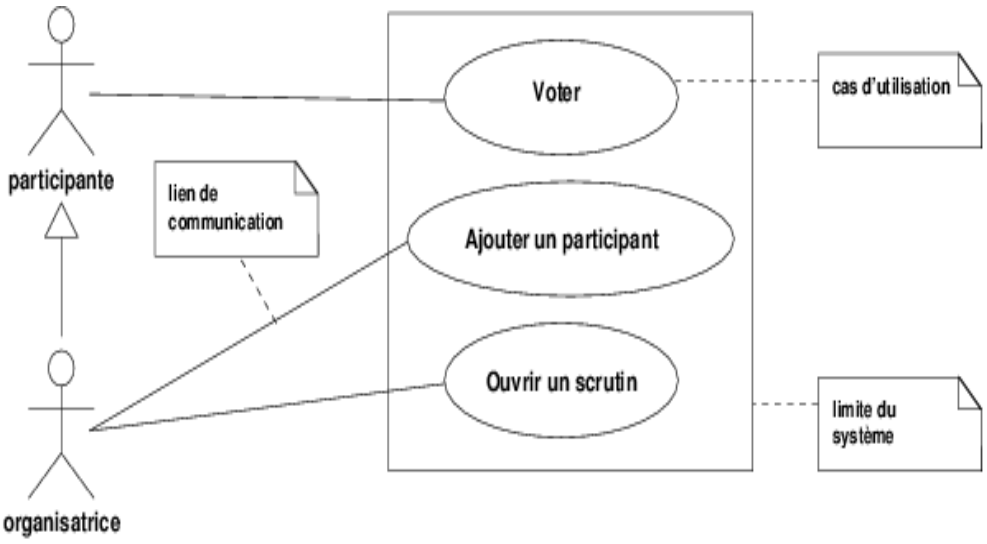


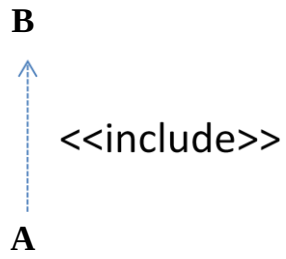
Figure 18 : Exemple 3 de relation entre acteurs : généralisation

L'acteur participant est l'acteur généralisant l'acteur organisateur. Dans ce cas, chacun des acteurs a accès à des cas d'utilisation qui correspondent à son rôle dans le système. Il est intuitif que tout cas d'utilisation accessible à un acteur général sera accessible à tous ses acteurs spécialisés. Ici, l'organisateur d'un processus de vote s'occupera d'ouvrir la session de vote ainsi que c'est lui qui ajoutera les participants. Un participant de son tour, ne peut que voter.

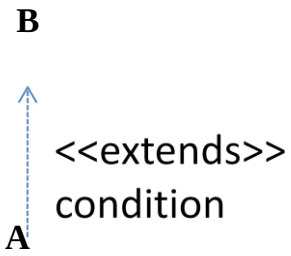
5.4 Liens entre « *use cases* »

Entre deux cas d'utilisation A et B, on peut avoir trois types de relations possible : l'inclusion, l'extension et enfin la généralisation. Ci-dessous, des détails sur ces types de relations sont fournis.

- **Inclusion:** le cas d'utilisation A inclut le cas d'utilisation B si A contient B dans sa définition. Ceci est noté par :



- **Extension:** le cas d'utilisation A étend le cas d'utilisation B si A ajoute des fonctionnalités facultatives à B. On peut avoir une condition que doit vérifier B avant d'avoir l'extension A. Ceci est noté par :



- **Généralisation:** le cas d'utilisation A est une généralisation du cas d'utilisation B, si A peut être substitué par B pour un cas précis. Par exemple, la spécialisation de certaines actions du cas d'utilisation d'origine. Comme exemple, voir la figure 19. Sur la figure 19, un cas d'utilisation générique est dit ***payer*** et deux cas d'utilisations spécifiques qui représentent des spécialisations du cas d'utilisation payer sont: ***payer par carte crédit*** et ***payer en espèce***.

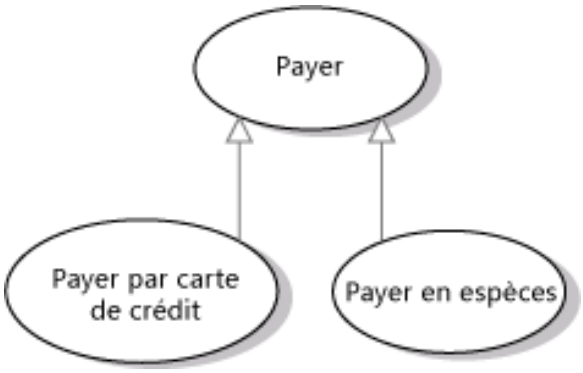


Figure 19 : Exemple de relation entre cas d'utilisation : généralisation

5.5 Comment créer un DCU?

Comme tout autre diagramme d'UML, la création d'un DCU se fait plutôt de manière intuitive et s'apprend avec la pratique et l'application. On peut donner quelques directives à suivre pour créer ce diagramme comme suit :

1. Identifier les cas d'utilisation: chaque comportement du système attendu par l'utilisateur.
2. Définir le périmètre du système : Paquetage limitant les services du système et le séparant de ses utilisateurs et des autres systèmes avec lesquels il peut fonctionner.
3. Identifier les acteurs : par exemple, ceux qui utiliseront le futur logiciel.
4. Évaluer les besoins de chaque acteur en CU.
5. Regrouper ces CU dans un diagramme présentant une vue synthétique du paquetage.
6. Possibilité de documentation des CU complexes (car ça servira comme cahier de charge).
7. Et enfin, à ne pas descendre trop bas ans le raffinement, car on ne cherche pas à réinventer l'analyse fonctionnelle

5.6 Autres exemples de DCU?

L'exemple de la figure 20 montre encore avec plus de détails un système de guichet automatique. Ce guichet permet de retirer de l'argent par des clients. Le retrait d'argent implique la modification du stock. Cette opération de retrait d'argent peut être étendu en permettant de retirer des devises ou des euros. Ici les conditions d'extension ne sont pas montrées. D'autre part, le guichet est accessible par un technicien qui ravitaille le coffre d'argent. Cette dernière opération, certes, changera le stock comme indiqué sur le DCU.

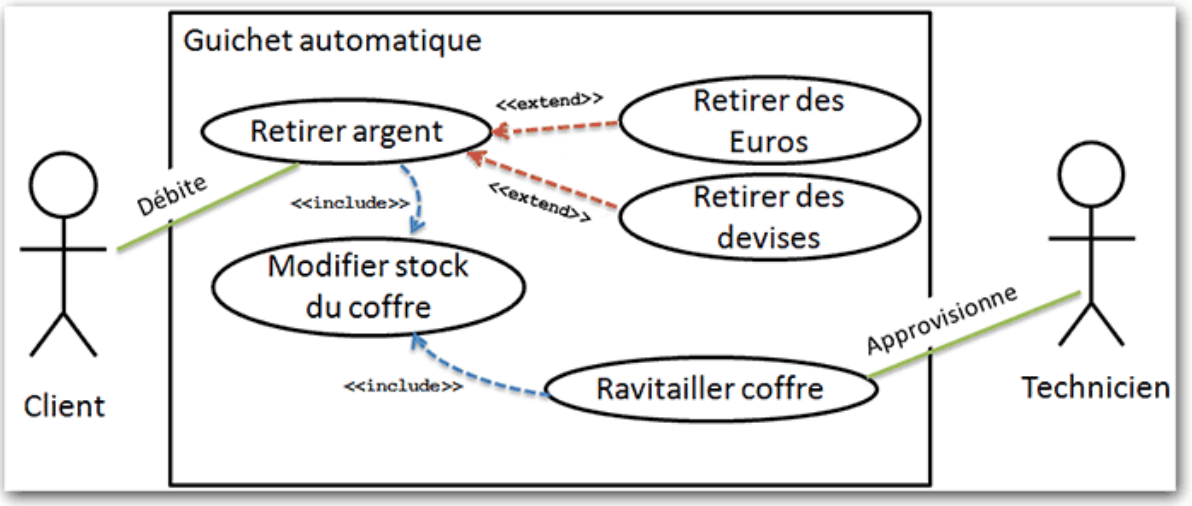


Figure 20 : Exemple 1 d'un DCU

L'exemple de la figure 21 montre un autre DCU, où on peut trouver les différents types de relations : généralisation entre acteurs (client et client fidèle), généralisation entre cas d'utilisation (paye et payer par chèque ou payer en ligne), inclusion (entre commander un article et payer), et enfin extension conditionnelle entre consulter ses points fidèles et éditer un chèque de fidélité.

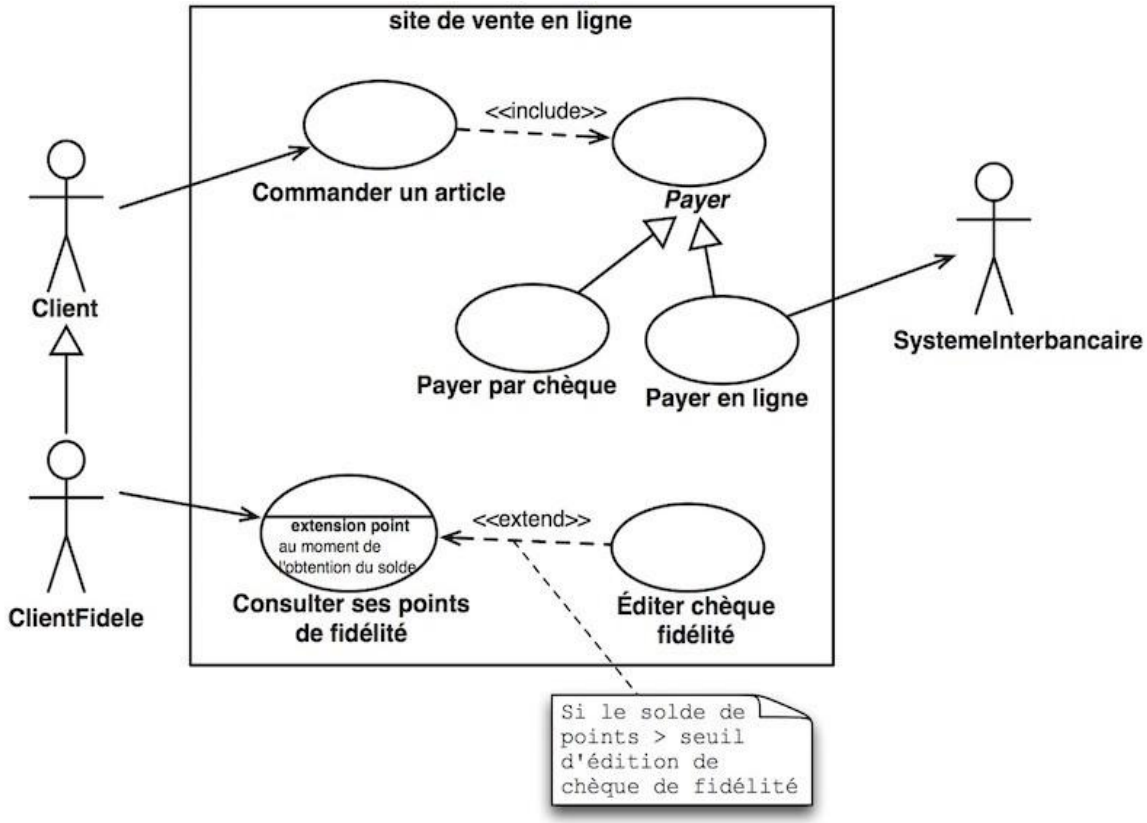


Figure 21 : Exemple 2 d'un DCU

6. Diagramme de la structure

Les diagrammes de cas d'utilisation (DCU) ou les ***use case diagrams*** (UCD) sont utilisés préalablement comme une technique d'analyse des besoins et d'identification de services, acteurs, liaisons entre acteur et services, liaisons possibles avec d'autres systèmes, …. Ces diagrammes sont souvent utilisés en phase d'analyse.

La conception du système commence à poser la question du « comment réaliser le système ? ». En **conception orientée objet**, les concepts clés de la réalisation sont les classes et les objets. Cette section présente respectivement: Diagramme d'Objets et le Diagramme de Classes.

6.1 Diagramme d'objets

L'objectif de ce diagramme et de représenter les objets et leurs liens à un instant donné, durant la vie du système. C'est une vision statique instantanée du système à un moment donné de son exécution. Donc, il s'agit de l'état global du système à instant donné. Cet état est défini par : le nombre d'objets existants, leurs états interne, leurs interactions instantanées, etc. Le diagramme d'objets peut être réalisé suite à un diagramme de DCU et il représente une instance possible du diagramme de classe qu'on va voir dans la prochaine section.

Un diagramme d'objets est un graphe ; ainsi composé d'un ensemble de nœuds liés par des arcs ou des arêtes.

• **Nœuds du graphe** = Objets. Possibilité de supprimer le nom, la classe et/ou les attributs (objet anonyme, non typé ou d'état inconnu)

• **Arêtes du graphe** = Liens entre objets. Ils peuvent être des liens binaires: entre 2 objets, des liens n-aire : entre n objets. Il est possible de nommer les liens d'interaction.

La figure 22 montre comment un objet sera présenté :

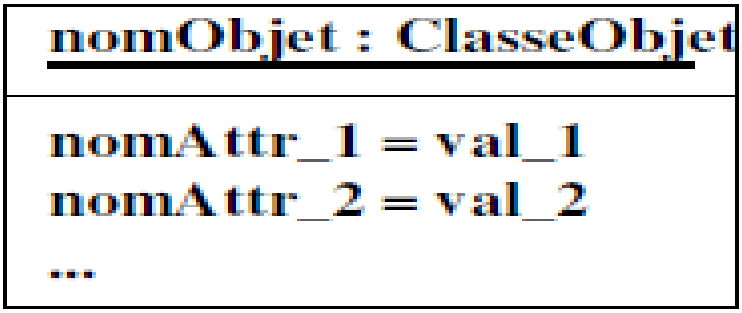


Figure 22 : représentation d'un objet dans un diagramme objets

La figure 23 montre un diagramme d'objets où 6 objets sont liés. On voit un objet avec son identifiant et sa classe (maVoiture de la classe Voiture), des objets anonymes qui sont de la classe Roue, et enfin un objet Moteur dans on ne connait pas la classe.

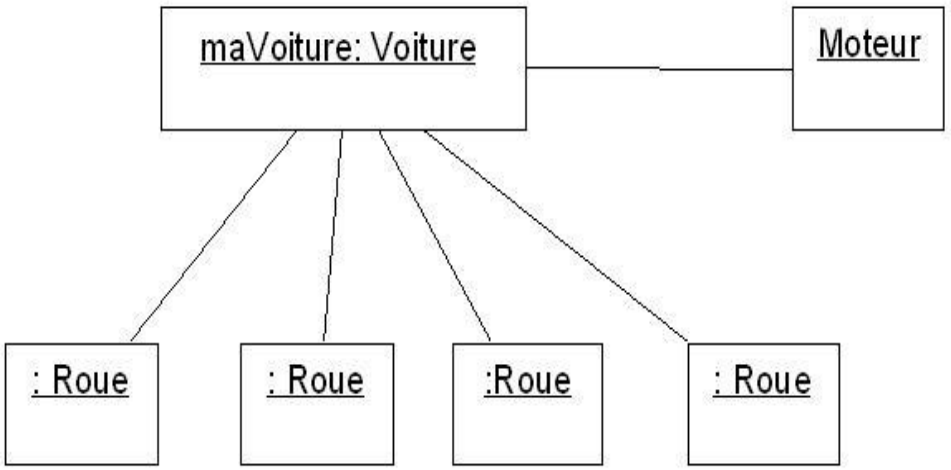


Figure 23 : Exemple 1 d'un diagramme objets

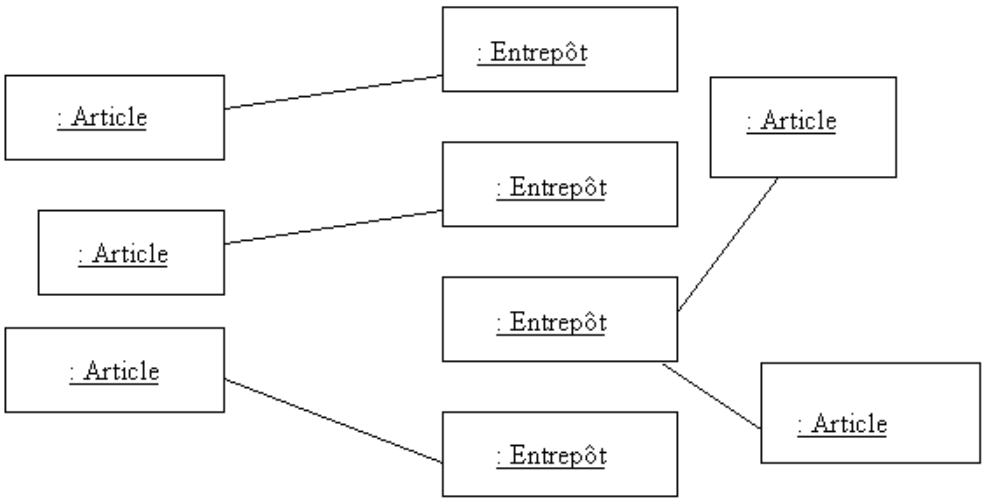


Figure 24 : Exemple 2 d'un diagramme objets

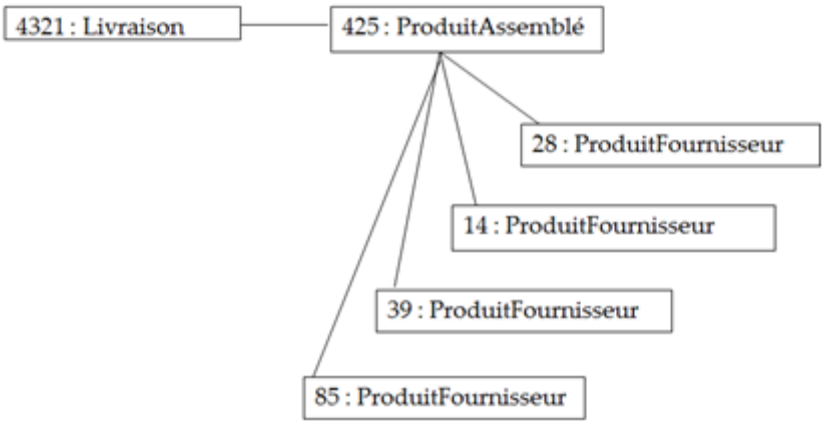


Figure 25 : Exemple 3 d'un diagramme objets

Sur la figure 26, on a un diagramme d'objet avec des interactions ayant des étiquettes (noms d'interaction).

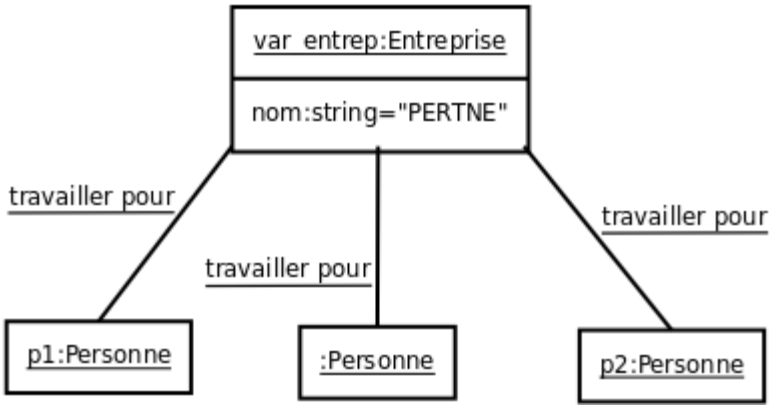


Figure 26 : Exemple 4 d'un diagramme objets

6.2 Diagramme de classes

Le diagramme de classes modéliser la structure rigide et inchangeable du système orienté objet. C'est une vision statique indépendante du temps. C'est une généralisation et une abstraction des diagrammes d'objets. Un diagramme de classe peut être utilisé pour dériver une infinité de diagramme d'objet. Et à partir d'un diagramme d'objet, il est toujours possible de créer un diagramme de classe.

Ce diagramme sert à:

- Montrer les **classes** du système: **nom**, **propriétés**, **méthodes**, **modificateurs,** etc.

- Montrer les **relations** ou **associations** entre ces classes: **héritage**, **agrégation**, **composition**, et **interaction.**

Représentation de la classe :

Une classe sera schématisée comme présenté sur la figure 27. Elle peut être documentée ou non ; ainsi il est possible d'avoir uniquement le nom de la classe, comme il est possible de raffiner la description et d'avoir plus de détails sur cette classe.

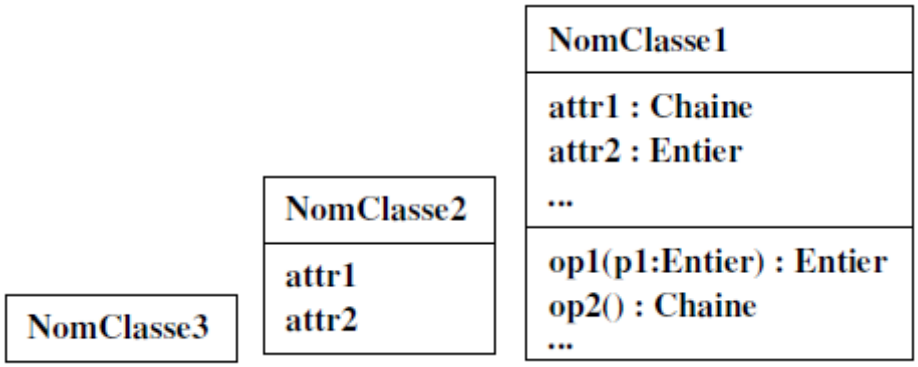


Figure 27 : représentation d'une classe dans un diagramme de classes

Format de description d'un attribut d'une classe :

Chaque attribut de classe sera écrit selon la forme :

[Vis] Nom [Mult] [":" TypeAtt] ["=" Val] [Prop]

où

- **Vis** : + (public), - (privé), # (protégé), (package)
- **Mult** : "[" nbElt "]" ou "[" Min .. Max "]"
- **TypeAtt** : type primitif (Entier, Chaîne, ...) ou classe
- **Val** : valeur initiale à la création de l'objet
- **Prop** : {gelé}, {variable}, {ajoutUniquement}, ...
- Attributs de classe (statiques) doivent être **soulignés**
- Attributs dérivés précédés de "/" : ces attributs peuvent devenir des méthodes par la suite.

Exemples :

- # onOff : Bouton - x : Réel coord[3] : Réel + pi : réel = 3.14 {gelé}
- inscrits[2..8] : Personne /age : Entier

<u>Format de description d'une opération :</u>

Chaque méthode de classe sera écrite selon la forme :

[Visibilité] Nom ["(" Arg ")"] [":" Type]

Où

- **Visibilité** : + (public), - (privé), # (protégé)
- **Arg** : liste des arguments selon le format

  [Dir] NomArgument : TypeArgument

  où Dir = in (par défaut), out, ou inout
- **Type** : type de la valeur retournée (type primitif ou classe)
- Opérations abstraites/virtuelles (non implémentées) en ***italique***
- Opérations de classe (statiques) : **<u>soulignées</u>**
- Possibilité d'annoter avec des contraintes stéréotypées : «precondition» et «postcondition», programmation par contrats, etc.
- Possibilité de surcharger une opération : même nom, mais paramètres différents
- Stéréotypes d'opérations : «create» et «destroy»

La figure 28 montre une classe avec ses attributs, méthodes et modificateurs possibles.

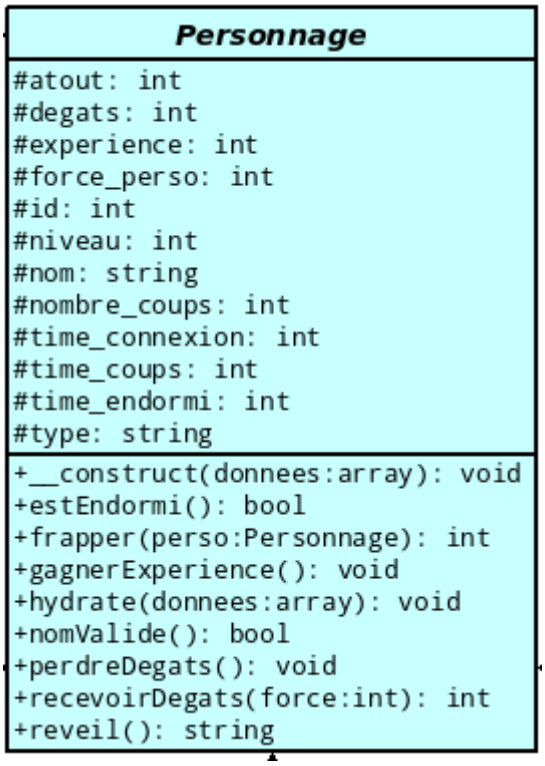


Figure 28 : Exemple complet d'une classe en diagramme de classes

Association entre classes:

Les associations entre classes seront, souvent, déduites des associations exprimées sur le diagrammes d'objets. Après classification d'objets en un ensemble de classes, les associations inter-objets exprimées sur le diagramme d'objets seront transformées en des associations entre classes avec leurs cardinalités nécessaires et ajoutées au diagramme de classes. Comme exemple de construction de diagramme de classes à partir de celui des objets, on considère l'exemple des figures 29 et 30.

Sur la figure 29, un diagramme d'objets est présenté. Plusieurs objets anonymes appartenant à 4 classes différentes sont interconnectés par plusieurs liens. A base de ce diagramme, on déduit :

1. L'existence de 4 classes dans le diagramme des classes ;
2. Les associations et leurs cardinalités sont déduites des liens du diagramme d'objets.

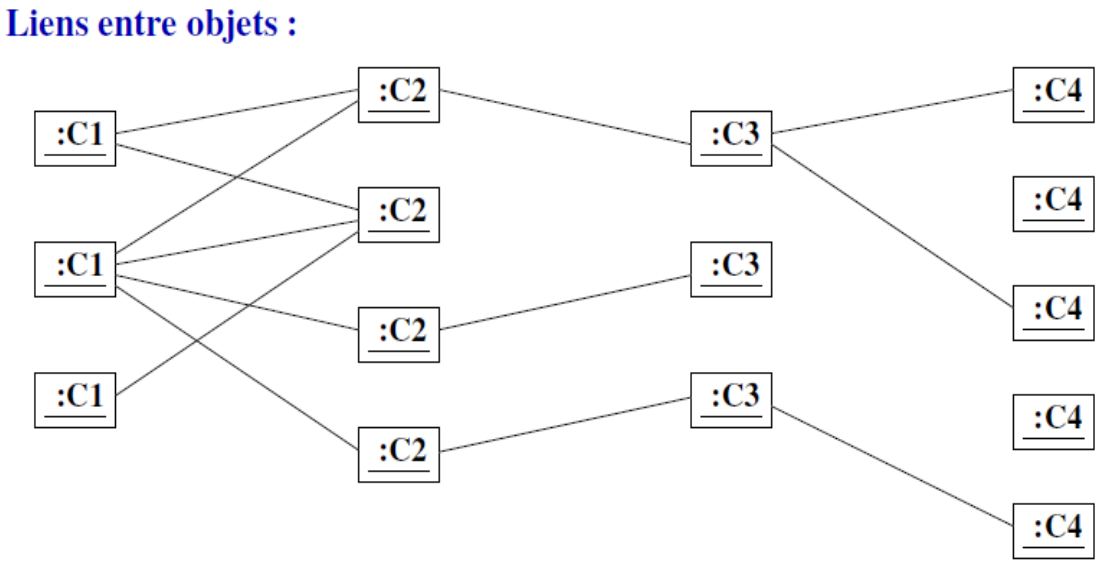


Figure 29 : liens entre objets dans un diagramme d'objets.

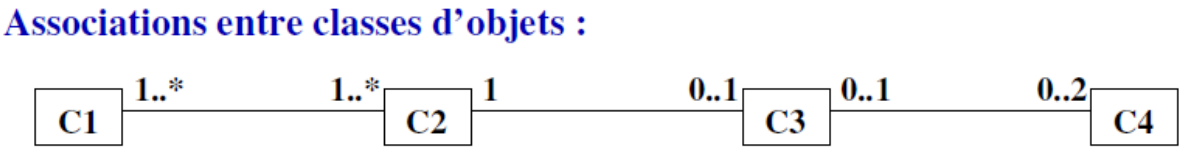


Figure 30 : liens entre classes, déduite des liens dans un diagramme d'objets.

Il est, aussi, possible d'annoter les associations entre classes avec leurs intitulées, leurs ajouter les rôles des différentes instances des classes associée. Ceci est présenté sur les figures 31, 32.

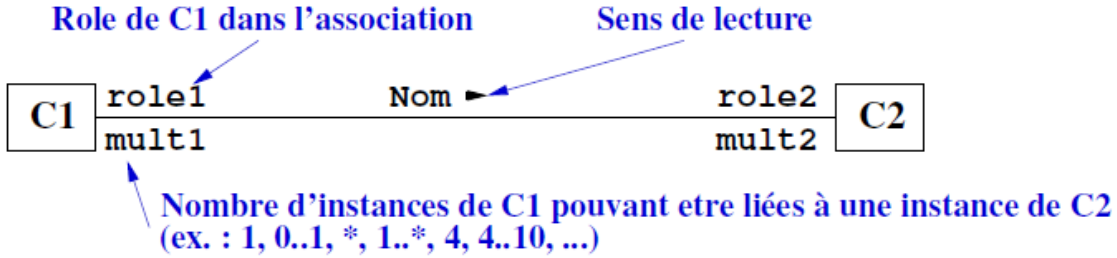


Figure 31 : liens entre classes avec annotations.

Exemple :

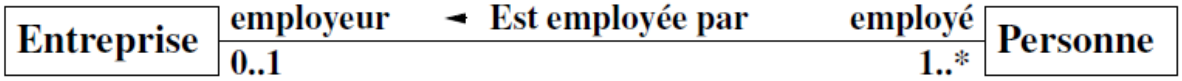


Figure 32 : Exemple de liens entre classes avec annotations.

Les associations entre classes peuvent être représentées graphiquement comme c'est le cas sur l'exemple de la figure 32 comme elles peuvent être représentées explicitement par l'ajout d'attributs dans les classes en association. La figure 33 montre une association exprimée sous forme d'attributs inclus dans l'une des classes en association.

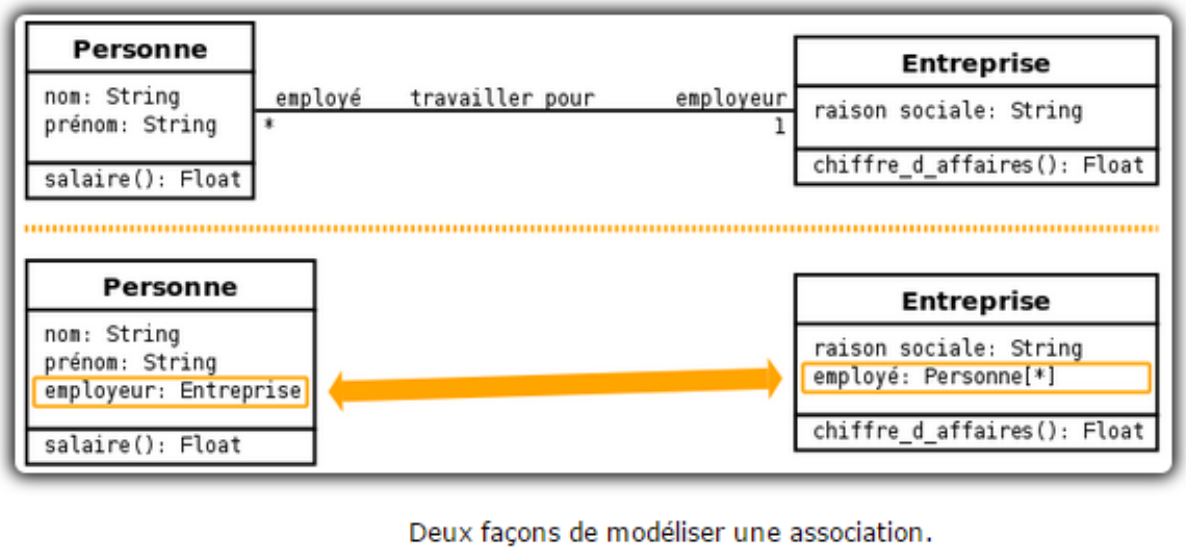


Figure 33 : Association sous forme d'attribut de classe.

Navigabilité d'une association entre deux classes C1 et C2 :

La navigabilité est la capacité d'une instance de C1 (resp. C2) à accéder aux instances de C2 (resp. C1)

- **Par défaut :** Navigabilité dans les deux sens. C1 a un attribut de type C2 et C2 a un attribut de type C1

- **Spécification de la navigabilité : Orientation de l'association:** C1 a un attribut du type de C2, mais pas l'inverse (Figure 34).

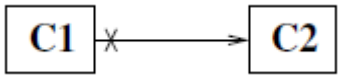


Figure 34 : Navigabilité dans un seul sens.

Remarque : Dans un diagramme de classes conceptuelles, toute classe doit être accessible à partir de la classe principale.

Contraintes sur les associations

Se sont des contraintes sur une extrémité de l'association. Exemple : {***ordered***}, {***variable***}, {***frozen***} , {***addOnly***}, etc. Ces contraintes augmentent l'expressivité du diagramme et ajoutent des informations indispensables pour le raffinement et l'implémentation des classes.

- Ordonnée : ***ordered :*** les valeurs sont ajoutées en ordre précis.
- La variabilité (***variable)*** indique si des données peuvent être mises à jour (ajoutées, modifiées, supprimées) ou non. Par défaut, une donnée peut être mise à jour.
- la contrainte {***frozen***} indique qu'une donnée, une fois créée, ne peut plus être mise à jour.
- La contrainte {***addOnly***} indique qu'on ne peut qu'ajouter des données.

Exemples : dans un historique, les faits seront ajoutés uniquement, sans possibilité de les supprimer, et aussi ils sont ordonnés.

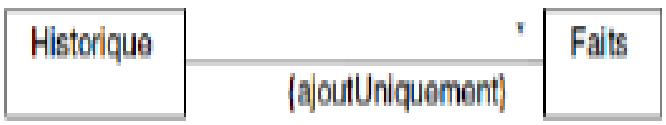


Figure 35 : Contrainte sur une association.

Contraintes entre associations

Se sont des contraintes entre associations liées à la même classe. C'est le cas par exemple de :

- {subset} : quand les l'une des instances d'une classe C1 sont un sous ensemble des instances d'une deuxième classe C2. C1 et C2 sont en association sous contrainte « subset » et elles sont liées à une seule classe C. Voir l'exemple ci-dessous.

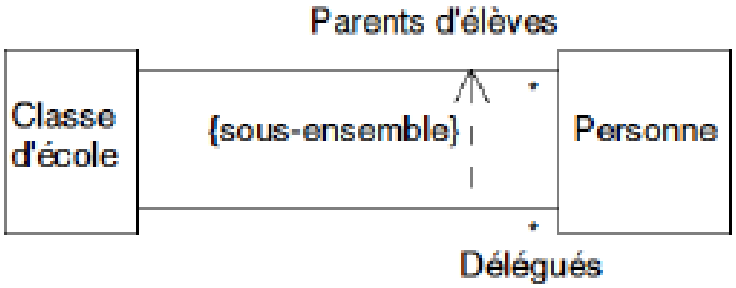


Figure 36 : Contrainte *subset* entre deux association.

- {xor} : Dans ce cas, l'une des deux associations existe seulement. Voir la figure de l'exemple ci-dessous. Un PC portable fonctionne ou bien de sa batterie ou bien du secteur.

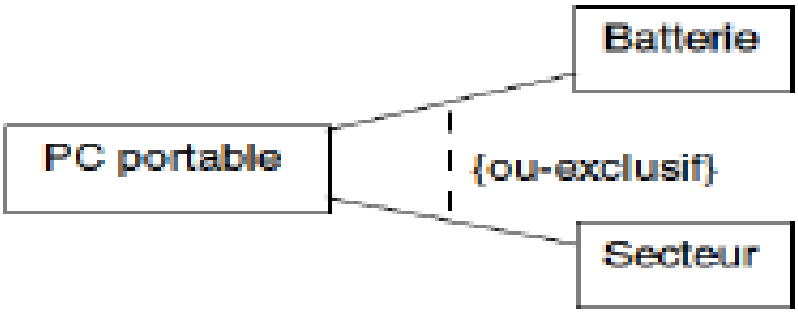


Figure 37 : Contrainte *xor* entre deux association.

Associations particulières: composition & agrégation

Se sont deux associations structurelles entre les classes d'un système. Les deux permettent à une classe dite Composite (ou agrégation) de contenir des instances d'autres classes dites Composantes ou (agrégées). La différence consiste en la puissance de la composition par rapport à la relation d'agrégation. Dans une relation de composition, les classes composantes n'existent plus (et n'ont aucun sens) si la classe composite disparaisse. C'est le cas d'association existante entre une Banque (comme classe composite) et les Comptes Bancaires (comme classes composante). A l'opposition de cette association, l'agrégation est une relation plus faible et c'est las plus rencontrée dans les structures de classes.

Notation graphique de la composition :

- Une relation transitive et antisymétrique
- La création (copie, destruction) du composite (container) implique la création (copie, destruction) de ses composants
- Un composant appartient à au plus un composite

Elle sera représentée comme la figure 38. Un livre se compose de plusieurs chapitres. Chaque chapitre se compose de plusieurs paragraphes.

Figure 38 : Notation graphique de la composition.

Notation graphique de l'agrégation :

Sur la figure 39, un polygone est composé de 3 à n point. Un point peut faire parti d'un nombre infini de polygone.

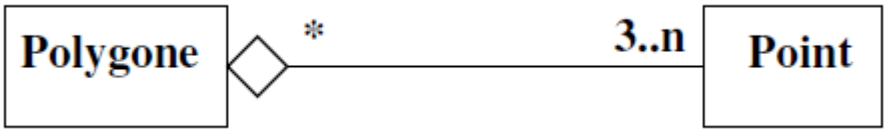


Figure 39 : Notation graphique de l'agrégation.

Associations particulières: Héritage

Un mécanisme proposé par les langages de programmation Objet: "B hérite de A" signifie que B possède :

1. Toutes les propriétés de A (attributs, méthode, associations, contraintes) avec:

- Possibilité de redéfinir les opérations de la sous-classe
- Polymorphisme

2. Ainsi que des nouvelles propriétés qui lui sont propres

Cette héritage peut être simple (une classe ne peut hériter que d'une seule classe mère) ou multiple (une classe peut avoir plusieurs classes mères). La figure 40 montre l'héritage simple entre trois classes filles (vélo, voiture, bateau) de leur classe mère (et abstraite) MoyeLocomotion. Ensuite, il y'a un héritage multiple de la classe fille Amphibie des deux classes mères : Voiture et Bateau.

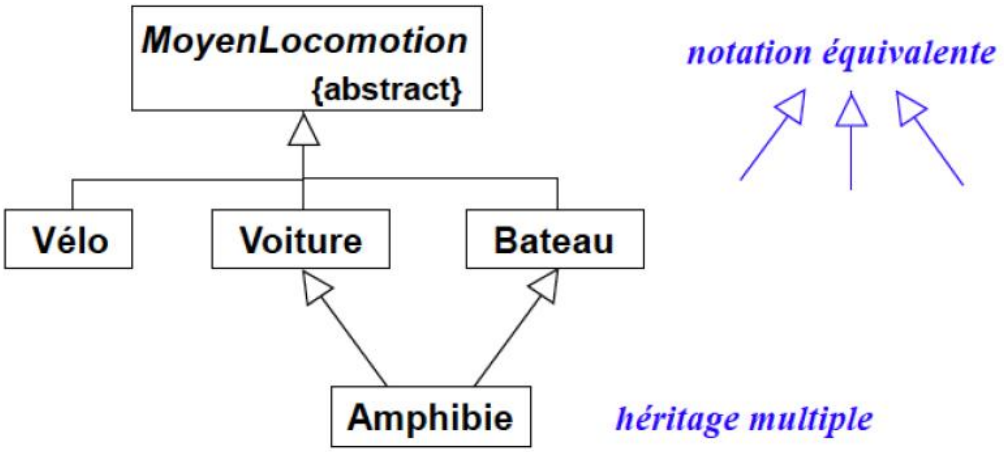


Figure 40 : Notation graphique d'héritage entre classes

Héritage: contraintes

Il est possible d'avoir des contraintes sur les relations d'héritages entre classe. Par exemple des héritages des classes C1 et C2, de leurs mères C, peuvent être disjoints, si les instances appartenant à C1 et les instances appartenant à C2 sont deux ensembles disjoints. La figure 41 montre un exemple où les relations d'héritage sont disjointes. Dans cette figure, les deux classes Agaricus et Boletus sont caréments disjointes. Ceci impose que toute classe déscendante de Boletus ne pourra jamais hériter de Agaricus et vise versa.

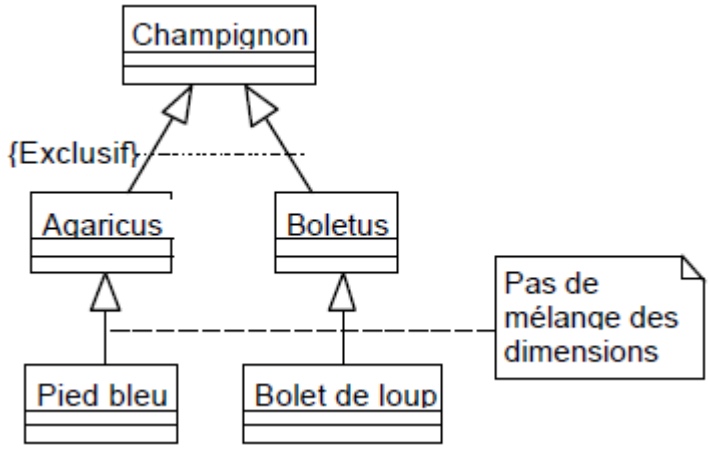


Figure 41 : contrainte sur héritage entre classes

La figure 42 montre une relation d'héritage avec une contrainte « inclusif ». Dans ce cas, il est possible d'avoir dans les classes descendantes (comme Pétrolette) un héritage multiple des classes (comme le cas de : A moteur et Terrestre) dont leurs héritages est plutôt inclusif.

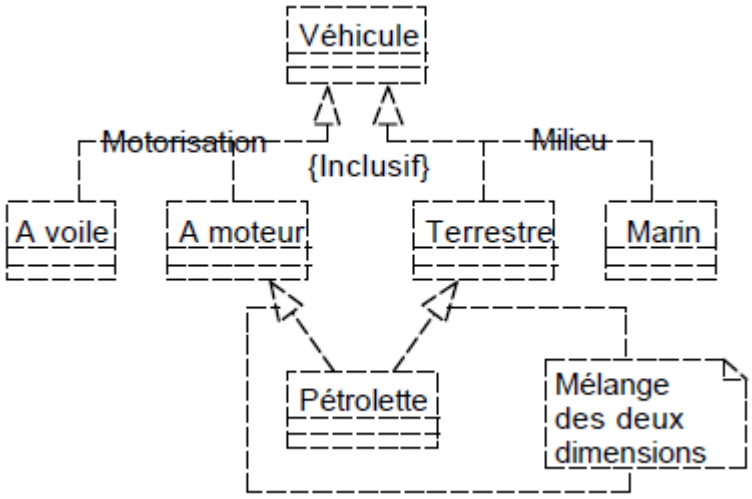


Figure 42 : contrainte « inclusif » sur héritage entre classes

Classes abstraites :

Une classe abstraite est une classe qui ne peut pas être instanciée. C'est une classe qui sert seulement comme un regroupement d'attributs et de méthodes et qui sera héritée par d'autres classes non abstraites qui vont être instanciées par la suite. Sur la figure 43, on présente l'exemple d'une classe abstraite et quelques classes descendantes.

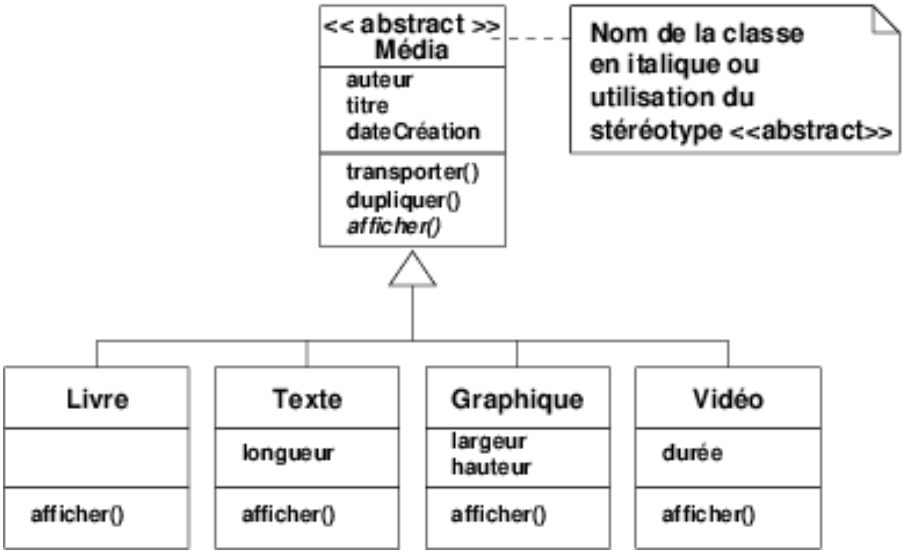


Figure 43 : la classe abstraite

Interface :

Qu'est-ce qu'une interface ?

Classe sans attributs dont toutes les opérations sont abstraites, caractérisée aussi par :

- Ne peut être instanciée,
- Doit être réalisée (implémentée) par des classes non abstraites,
- Peut hériter d'une autre interface

Pourquoi des interfaces ?

- Utilisation similaire aux classes abstraites
- En Java : une classe ne peut hériter que de plus d'une classe, mais elle peut réaliser plusieurs interfaces

En effet, une classe peut utiliser (use) ou implémenter une interface. Ces deux cas peuvent être exprimés avec UML comme présenté sur les figures 44, 45, 46.

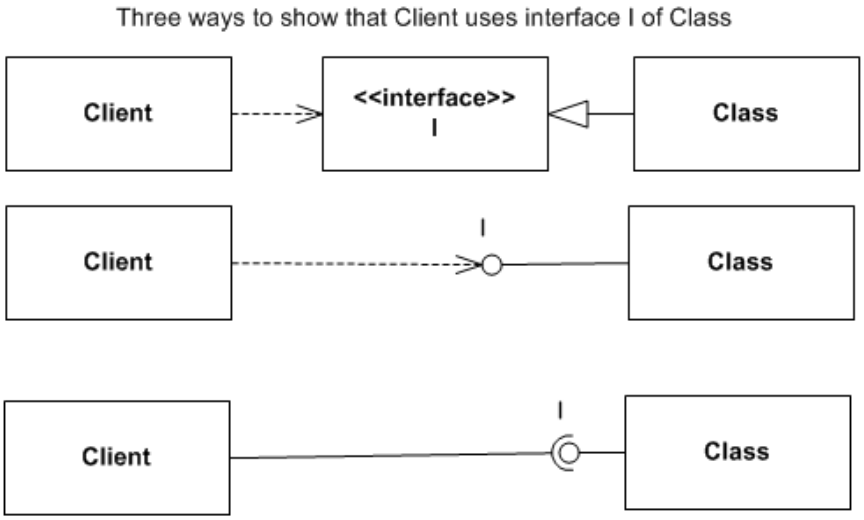


Figure 44 : classes utilisant une interface

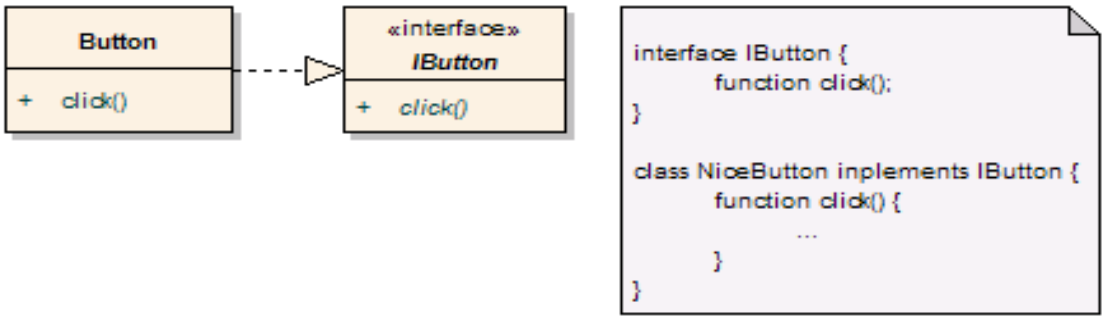


Figure 45 : une classe implémentant une interface

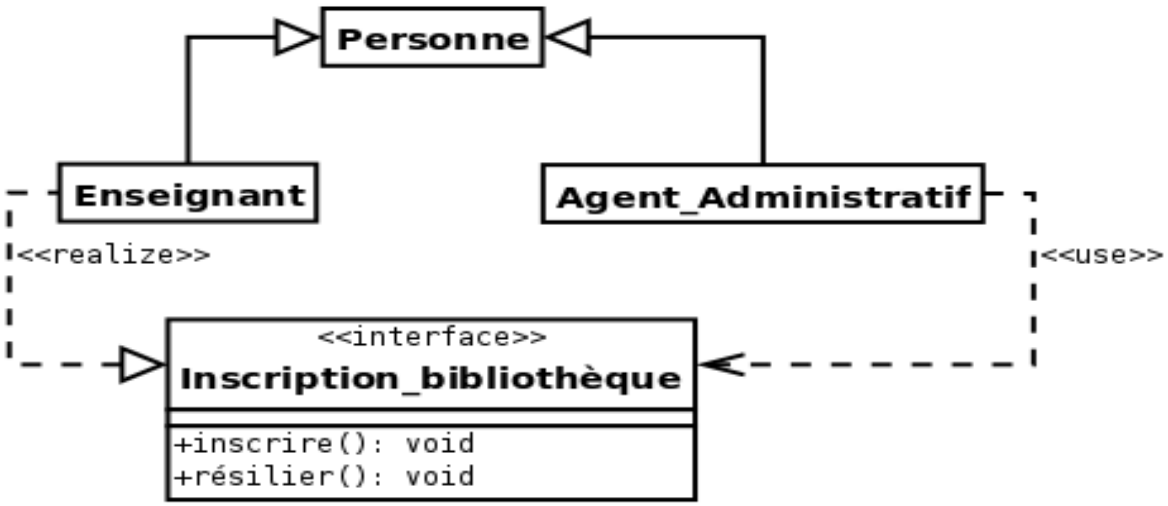


Figure 46 : classes utilisant et implémentant une interface

7. Diagramme de la dynamique du système orienté objet

Sur le côté dynamique et fonctionnement d'un système orienté objet, on peut distinguer entre deux niveaux de comportements : un niveau intra-objet qui montre comment un objet s'exécute et donc passe d'un état vers un autre état suite à l'exécution de ses méthodes, et le niveau inter-objets qui focalise plutôt sur un aspect plus abstrait concernant les interactions entre les objets sous forme d'échanges de message ou d'invocation de méthodes. Plusieurs diagrammes UML sont proposés pour ces objectifs. Ci-dessous, on montre deux diagrammes les plus exploités : diagrammes de séquence pour interaction inter-objets et les state-charts utilisés pour la modélisation des comportements internes des objets.

7.1 Diagramme de séquences

Le diagramme de séquence peut être utilisé soit en phase d'analyse ou en phase de conception. En phase d'analyse, ce diagramme peut être utile pour montrer :

- **Interactions** entre **acteurs** et **système**
- Décrire les **scénarios** des cas d'utilisation

En phase de conception, ce diagramme peut servir surtout à montrer :

- **Interactions** entre **objets**
- Réfléchir à **l'affectation de responsabilités** aux objets:

1. Qui crée les objets ?
2. Qui permet d'accéder à un objet ?
3. Quel objet reçoit un message provenant de l'IHM (Interface homme machine) ?

Ce diagramme montre explicitement une r**eprésentation temporelle** des interactions entre les objets et donc rend clair la c**hronologie** des messages échangés entre les objets et avec les acteurs. La figure 47 montre un simple exemple de diagramme de séquence avec trois objets en interaction.

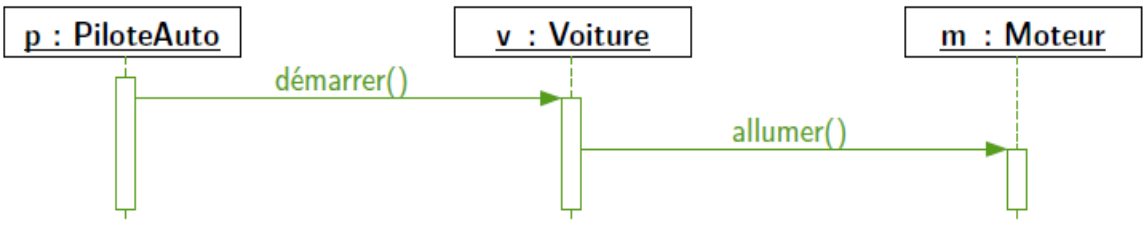


Figure 47 : Simple exemple de diagramme de séquence

Représentation graphique d'un diagramme de séquence :

Les éléments du diagramme de séquence sont les suivants :

- **Acteurs :** les utilisateurs en interaction avec le système ;
- **Objets** (instances)
- **Messages** (cas d'utilisation, appels d'opération, etc)

Et Il s'agit d'une représentation graphique de la chronologie des **échanges de messages** avec le système ou au sein du système. On va exprimer:

- « **Vie** » de chaque entité représentée verticalement. Ceci est fait par une ligne de vie de chaque entité intervenant dans l'interaction. Voir figure 78.
- **Échanges** de messages représentés horizontalement

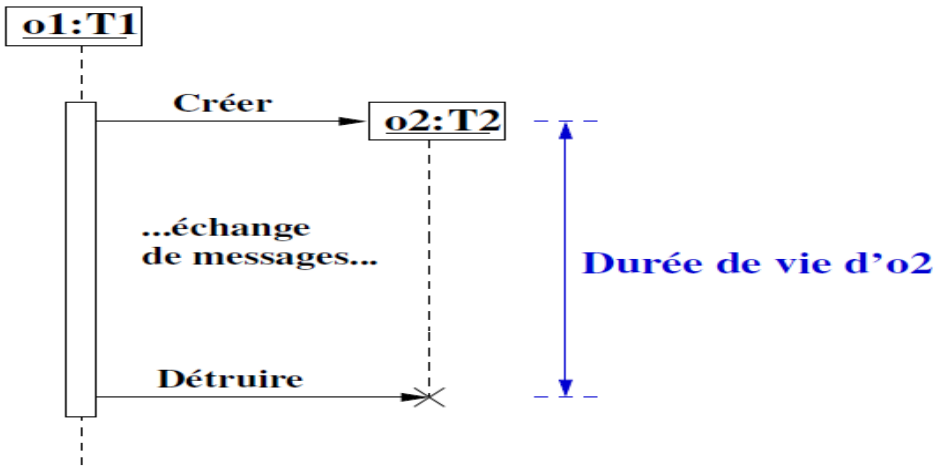


Figure 48 : ligne de vie d'entité en diagramme de séquence

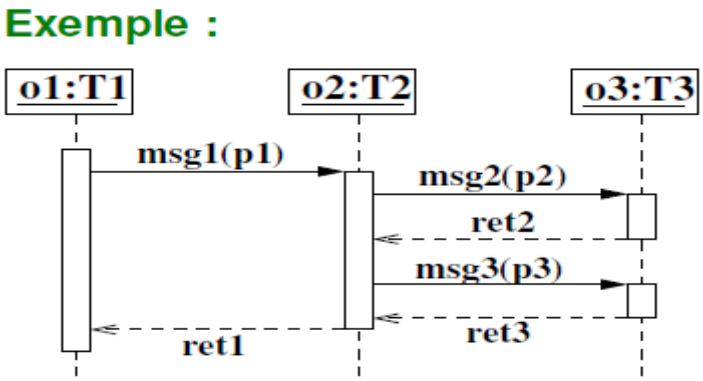


Figure 48 : échange de messages en diagramme de séquence

Sur la figure 48, un objet o1 de la classe T1 envoie un message synchrone msg1 vers l'objet o2 de la classe T2. o2 doit répondre à la fin avec une réponse ret1. Les flèches de messages montre la direction et aussi la nature de message : envoi, réponse, synchrone, asynchrones, …les rectangles horizontaux représentent les périodes d'activité des objets quand ils reçoivent des messages ou quand ils envoient des messages. Les différents détails que peut comporter un diagramme de séquence sont encore montrés sur les figure 49, 50, 51, et 52.

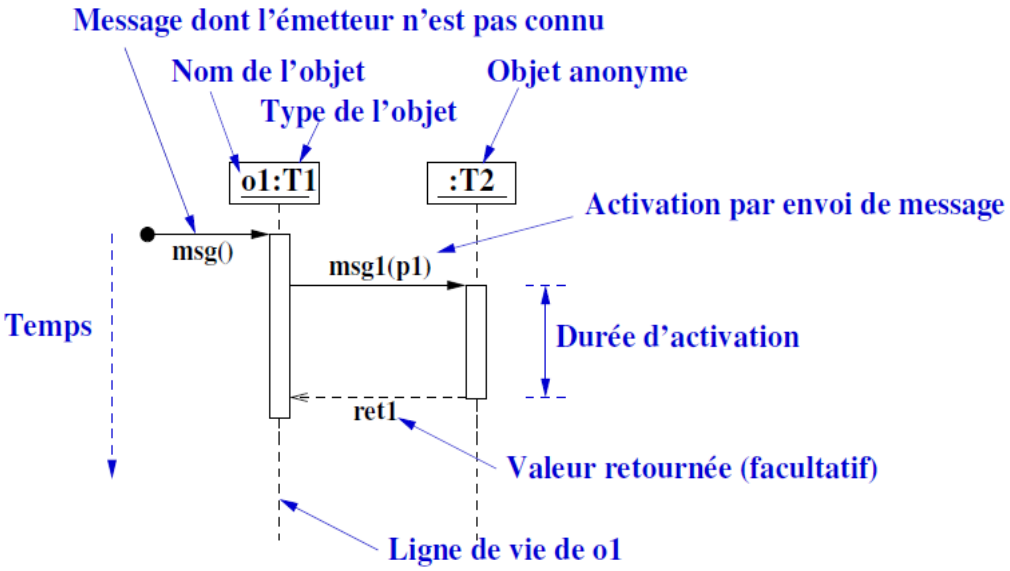


Figure 49 : diagramme de séquence et ses détails 1

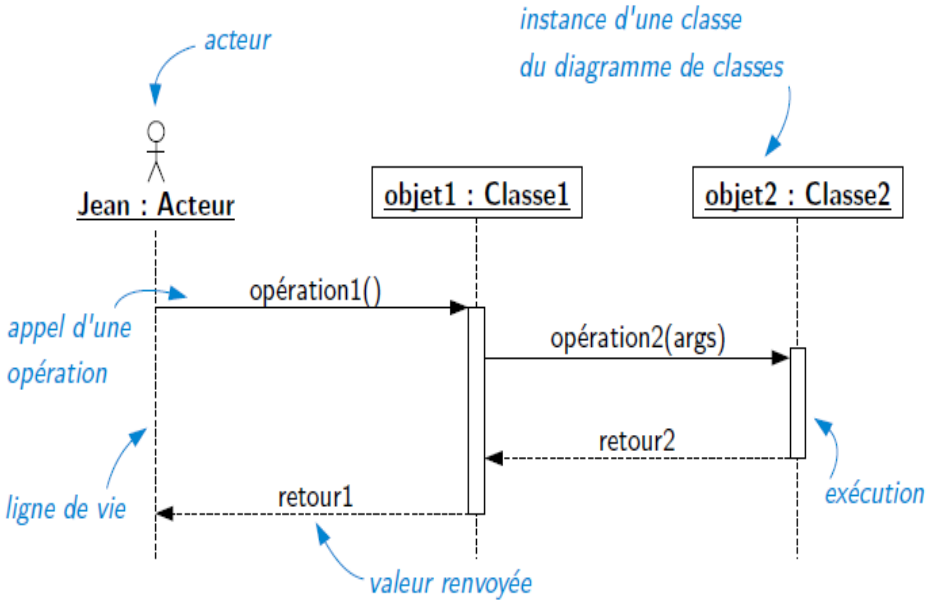


Figure 50 : diagramme de séquence sur un exemple et ses détails 2

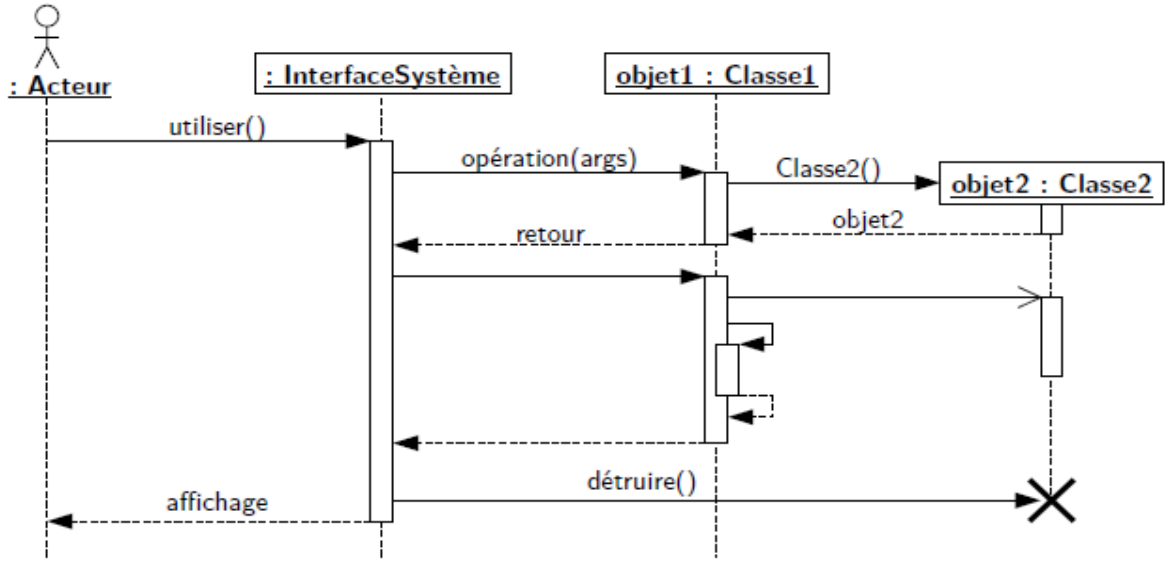


Figure 51 : diagramme de séquence sur un exemple et ses détails 3

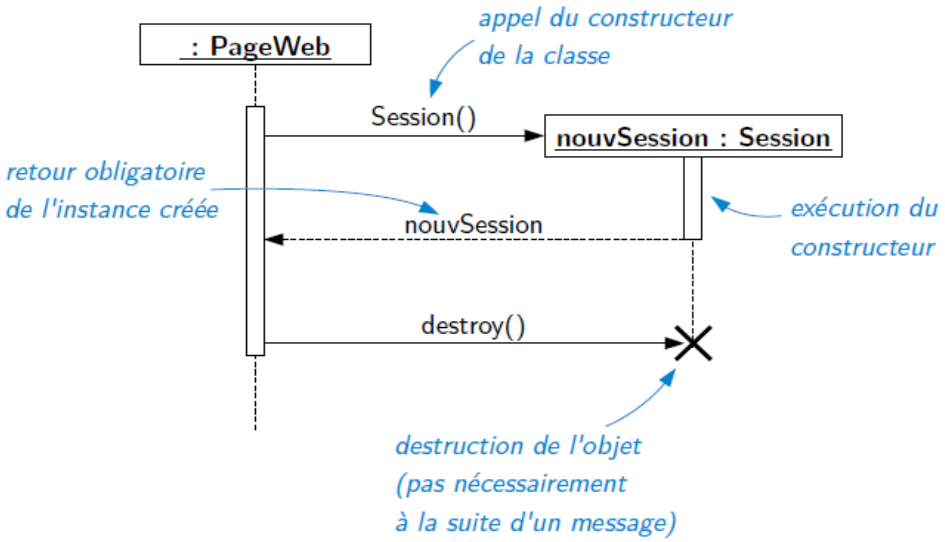


Figure 52 : diagramme de séquence sur un exemple et ses détails 4

Communication synchrone vs asynchrone:

Dans la communication synchrone, l'émetteur et le récepteur seront bloqués jusqu'à la fin de la communication. Ceci est modélisé par la couleur rouge comme sur la figure 53.

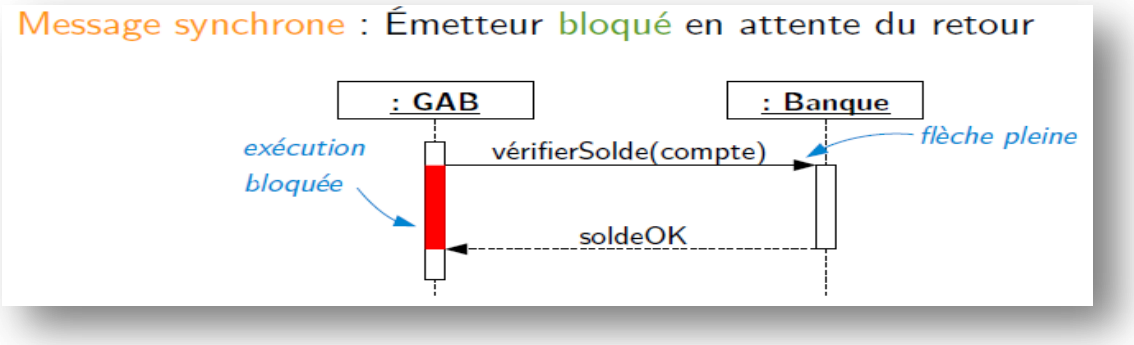


Figure 53 : communication synchrone

Sur la figure 54, la communication est asynchrone et sans blocage.

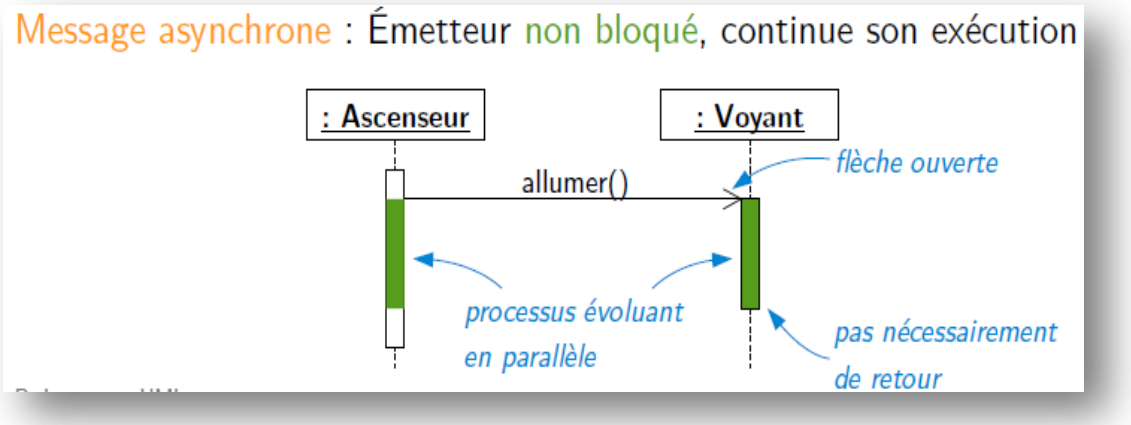
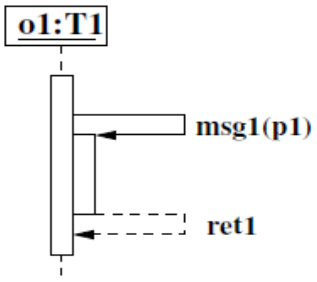


Figure 54 : communication asynchrone

Message réflexif:

Un objet qui envoie des messages à lui-même (appel d'une opération interne) se modélise comme sur la figure 55.



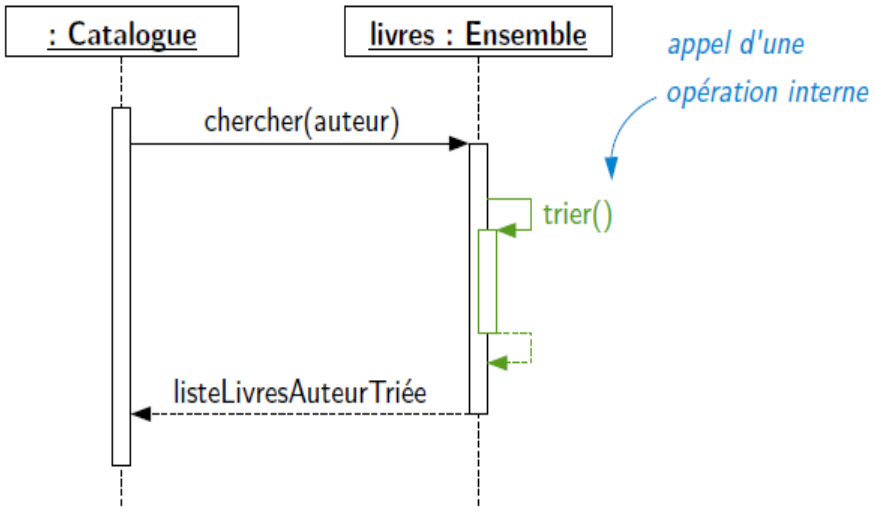


Figure 55 : message réflexif

Message réflexif:

Il est possible de montrer les délais de propagation que peut nécessiter l'envoi-réception d'un message entre deux objets comme sur la figure 56.

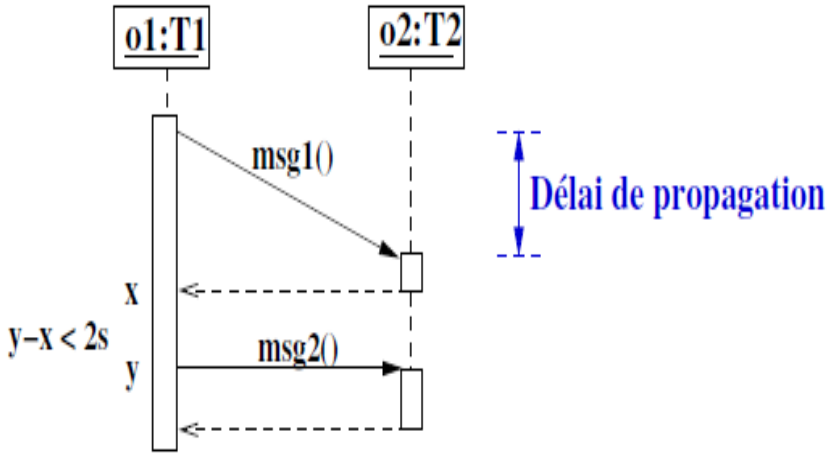


Figure 56 : délai de propagation

Structuration d'interaction:

Dans un diagramme de séquence, on peut exprimer plusieurs structures comme :

- **alt** : fragments multiple alternatifs (si alors sinon) ;
- **loop** : le fragment s'exécute plusieurs fois ;
- **ref** : référence à une interaction dans un autre diagramme ;

- **par** : fragment parallèle (traitements concurrents) ;
- **opt** : fragment optionnel ;
- **region** : région critique (un seul thread à la fois) ;

Alternance :

La figure 57 montre une structure conditionnelle. Deux possibilités sont possibles soit que les portes soient ouvertes ou qu'elles soient fermées. L'une des deux interactions sera possible selon la satisfaction de l'une des deux conditions.

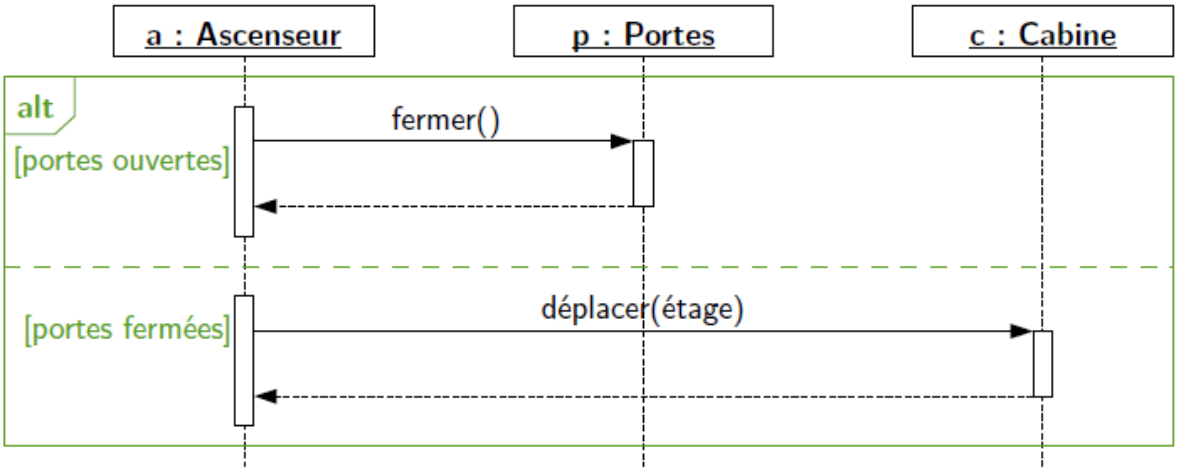


Figure 57 : Alternance

Sur la figure 58, une interaction itérative est montrée. Le mot clé loop est utilisé dans ce cas.

Itérative:

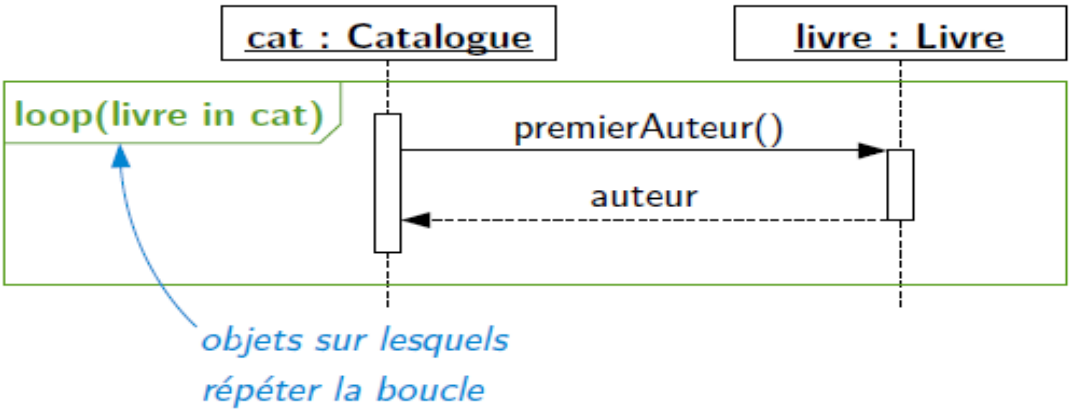


Figure 58 : la répétition

Sur la figure 59, un diagramme imbriqué est présenté. Au sein de ce diagramme une interaction itérative est exprimée. Cette interaction représente une dernière alternative en plus de deux autres avant elle.

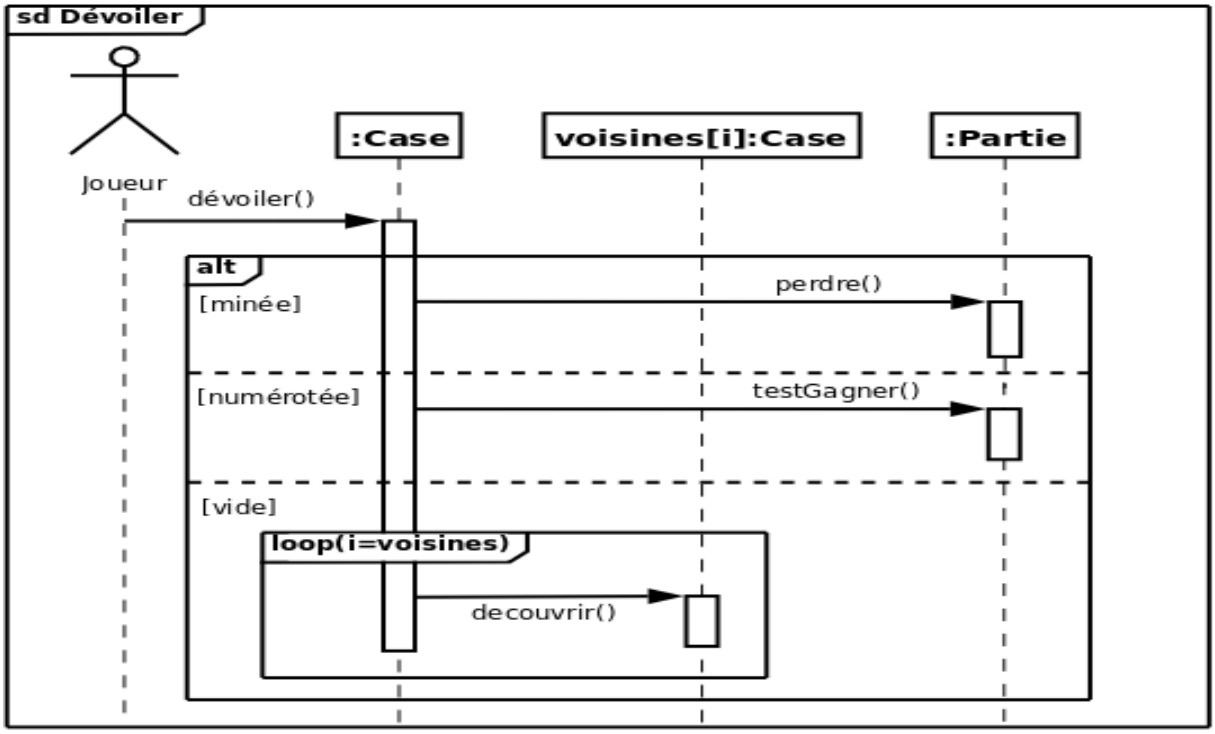


Figure 59 : la répétition incluse dans une alternance.

Référence :

Utilisé pour simplifier la présentation des DS et éviter la répétition des structures. Sur la figure 60, on montre une interaction nommée xx, qui sera par la suite repris dans un autre diagramme (figure 61) sans être obligé de la tracer complètement. Il s'agit d'une simple référence.

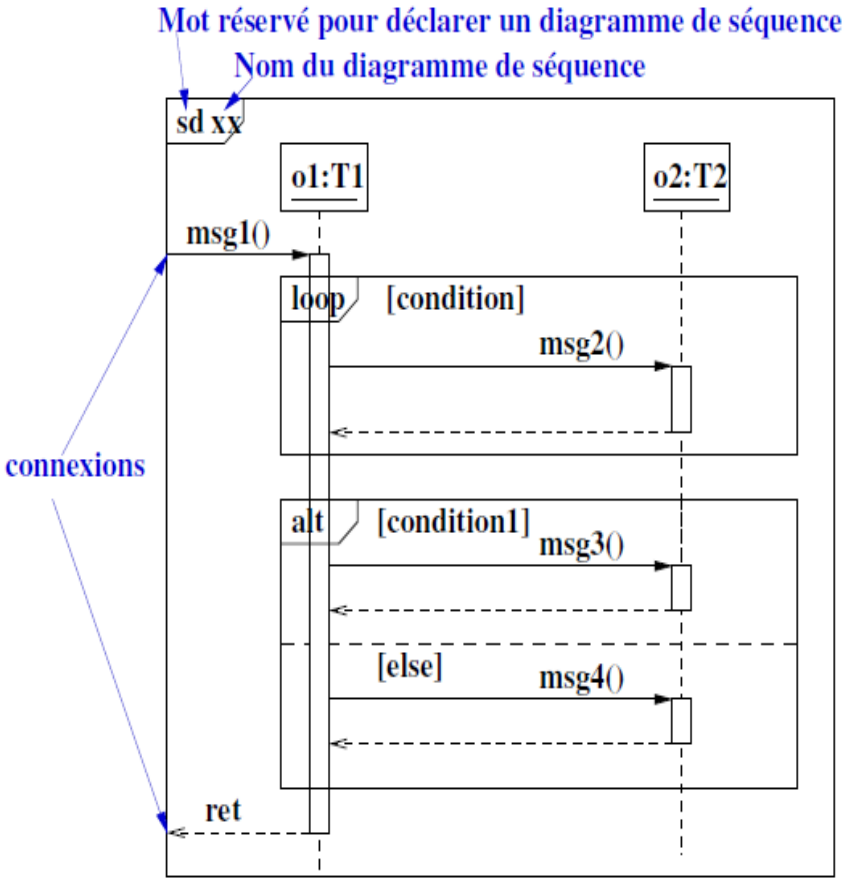


Figure 60 : une interaction nommée xx.

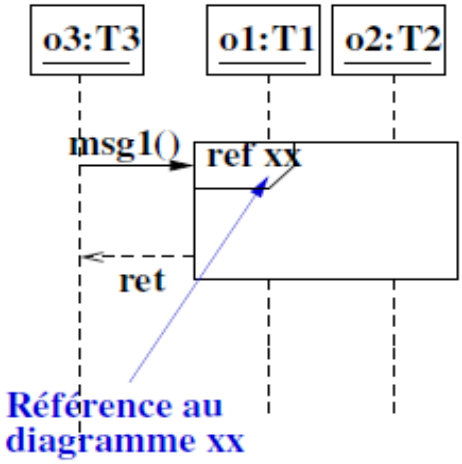


Figure 61 : référence à l'interaction xx.

Parallélisme :

Il est possible de montrer des interactions parallèles, en utilisant le mot clé PAR comme modélisé dans la figure 62.

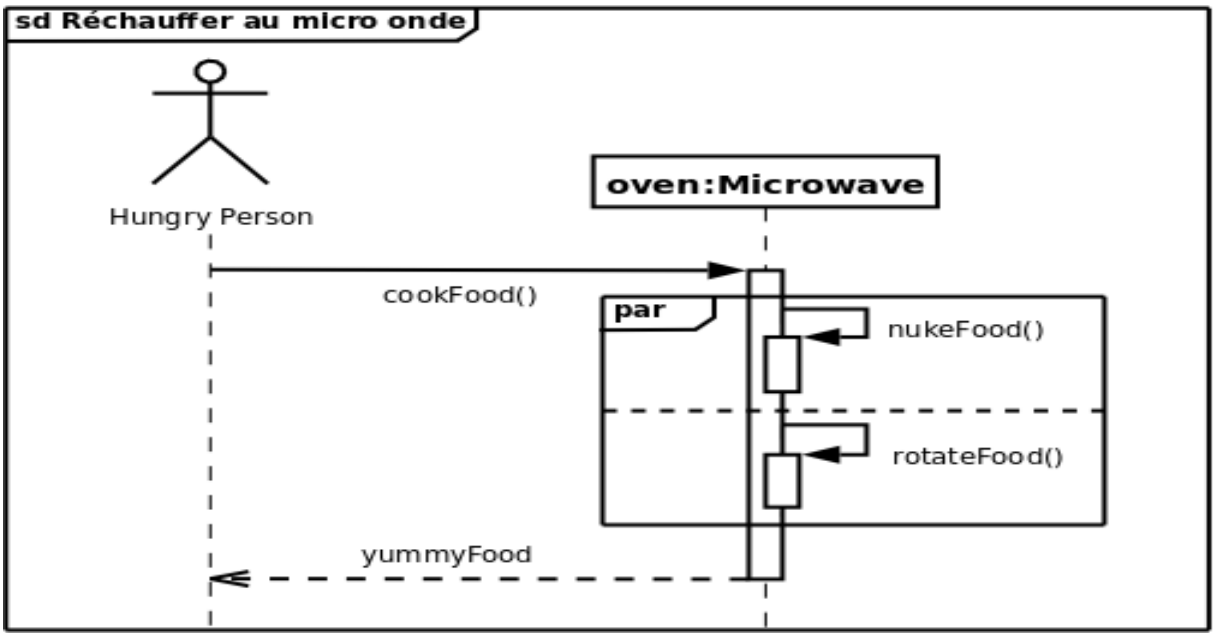


Figure 62 : Interactions parallèles.

7.2 Diagramme state-chart (état-transition)

L'objectif de ce diagramme est de montrer le comportement interne des objets et non pas leurs interaction externe. Donc, ce diagramme sert à :

- Décrire le comportement dynamique d'une entité (logiciel, composant, objet...).
- Décrire le comportement par des états et des transitions entre ces états.

Les deux concepts clés sont donc :

- État : abstraction d'un moment de la vie d'une entité pendant lequel elle satisfait un ensemble de conditions.
- Transition : modélise le changement d'état.

A titre d'exemple, la figure 63 montre un objet lampe qui bascule entre deux états : allumée et éteinte.

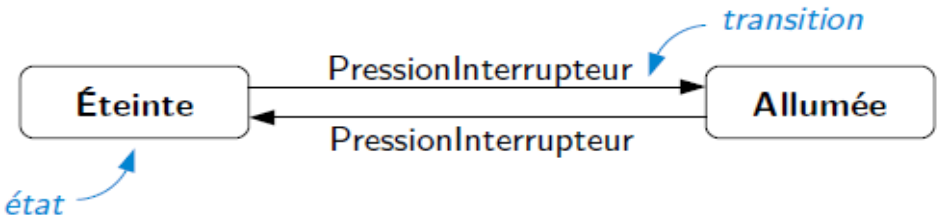


Figure 63 : exemple 1 de diagramme état-transition.

Il décrit quoi donc?

Comment **l'entité change d'état** suite à: l'occurrence d'un fait significatif ou remarquable qui peut être par exemple :

- Réception d'un **signal**
- Réception d'un **message**
- Expiration d'une **temporisation**
- ...

Les **événements déclenchent les transitions** d'un état vers un autre. Mais, il est à noter que :

- L'événement sera «perdu» si aucune transition spécifiée pour lui
- Il y a un **état initial**, mais **pas toujours d'état final**

La transition

Elle est étiqueté par (voir figure 64):

- Un **événement**;
- Une **condition:** sous forme: **when (condition)**
- Et une **action**

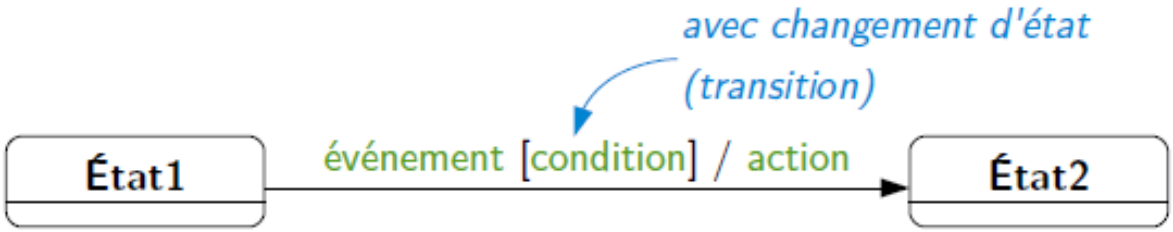


Figure 64 : une transition.

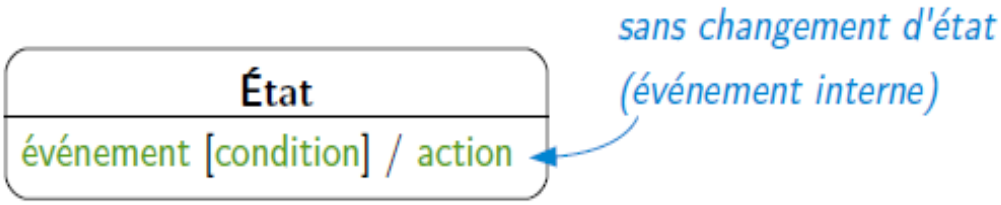


Figure 65 : un état.

Les états d'un diagramme state-chart

Trois états par lesquels un objet passe :

- **État initial** : Initialisation du système, exécution du constructeur de l'objet. Noté comme un point noir :

- **État final** : Fin de vie du système, destruction de l'objet. Notée comme suit :

- **États intermédiaires** : étapes de la vie du système ou de l'objet. Il peut être étiqueté par: événement à l'entrée : **Entry,** événement à la sortie : **Exit,** événement sans changement d'état : **event** et une Activité : **Do.**

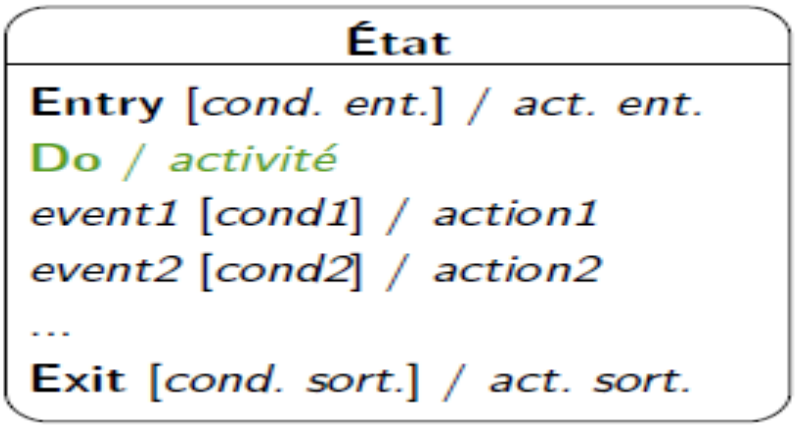


Figure 65 : un état intermédiaire.

Exemples d'un diagramme SC :

La figure 66 montre un exemple de state-chart simple où un objet passe par 3 états, état initial, inactif et actif.

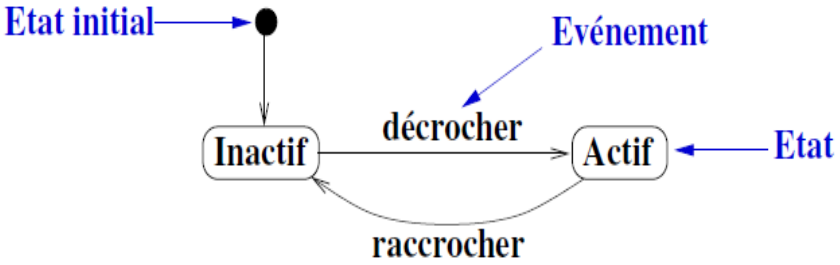


Figure 66 : exemple d'un diagramme état-transition.

La figure 67 montre un exemple de state-chart simple plus concret.

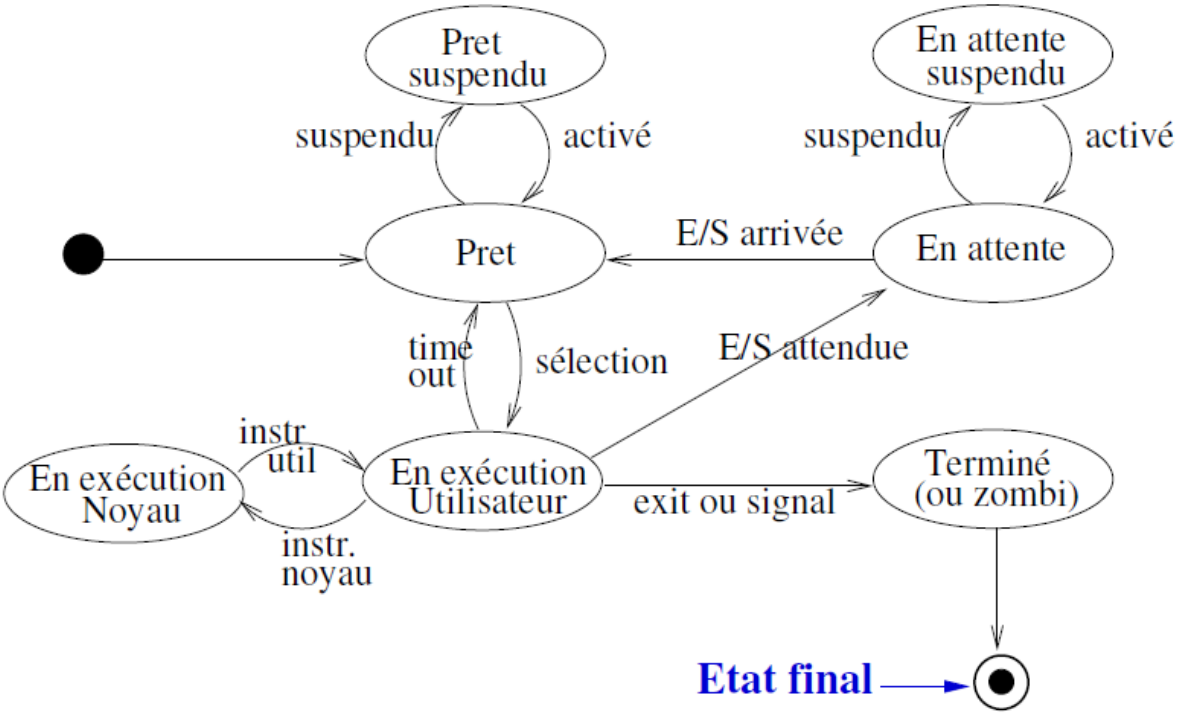
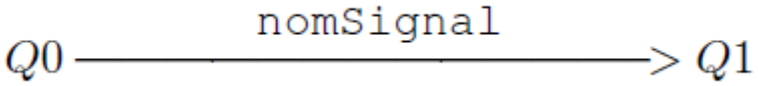


Figure 66 : exemple 2 d'un diagramme état-transition.

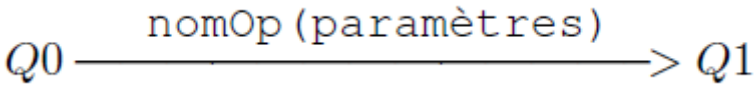
<u>Notation graphique d'événements possibles dans un SC :</u>

Le passage d'un état vers un autre état est fait suite à un événement. Dans un diagramme SC, on distingue les événements suivants changeant les états :

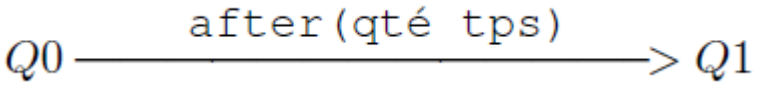
- Signaux :



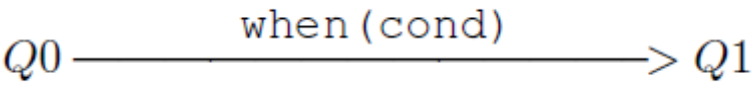
- Appels d'opérations :



- Evénements temporels : on passe de l'état Q0 vers Q1 suite à l'écoulement d'une quantité de temps.



- Evénements de changement : Passage dans l'état Q1 quand condition devient vraie.

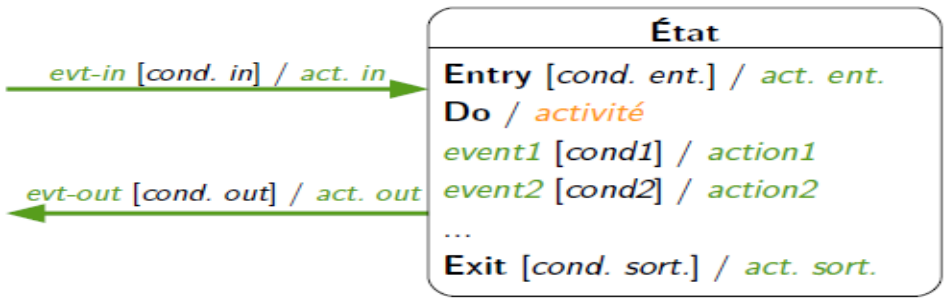


Activité vs action dans un SC

- L'activité peut être interrompue à cause d'un événement.
- Un événement interne interrompe l'activité, sauvegarde le contexte de l'état ensuite il déclenche l'action si la condition est satisfaite.

Dynamique interne d'état

En effet, objet sera dans un état suite à un événement evt-in, dans cet état il va faire une certaine activité. Cette activité peut être interrompue à tout moment par des événements internes : event1, event2, … suite à ces événements l'objet peut entrprendre des actions comme : action1, action2, … sans qu'il change d'état. L'objet ne peut sortir de cet état que suite à un événement Exit. Voir les figures 67, 68.



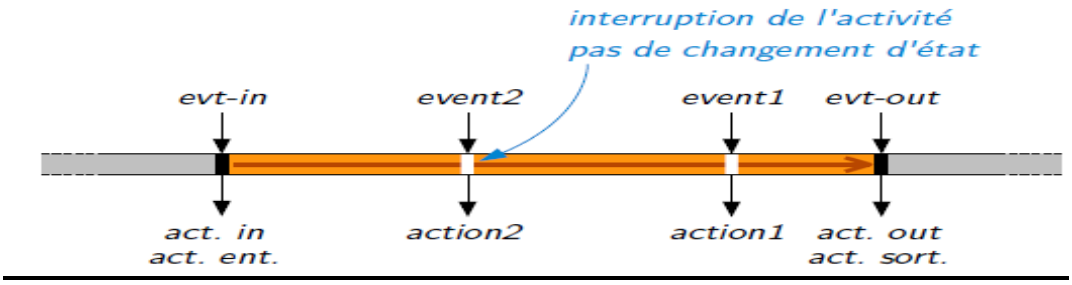


Figure 67 : dynamique interne 1 d'un état.

Sur la figure 68, on a introduit un événement evt-self qui peut se produire durant l'existence de l'objet dans un état. Un tél événement ne change pas l'état, mais oblige l'objet à exécuter l'action de sortie, ensuite l'action d'entrée.

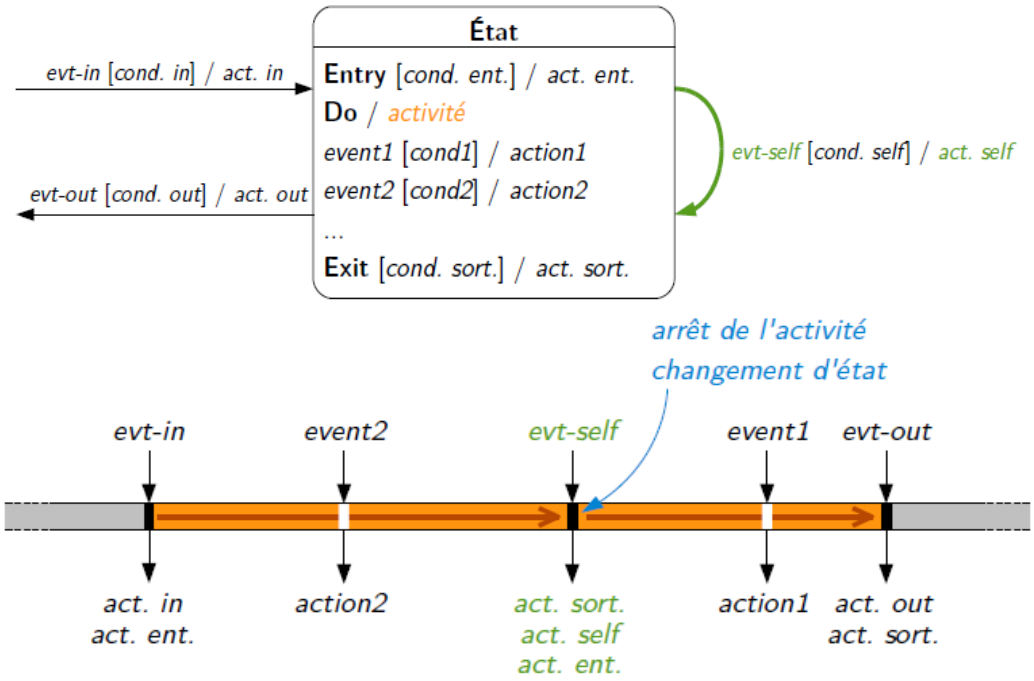


Figure 68 : dynamique interne 1 d'un état.

Exemple d'un d'état intermédiaire

Sur la figure 69, l'objet entre à l'état fourAllumé et exécute l'action : mettre thermostat à T. Ensuite une activité permanente sera de rendre le voyant allumé. Toute en gardant le voyant allumé (activité non interrompue), l'action chauffer s'exécute si sa condition est satisfaite. L'objet quitte cet état (sans action de sortie) si un événement

réglerTempérature arrive. Cet événement le fait encore entrer dans cet état mais en changeant la température.

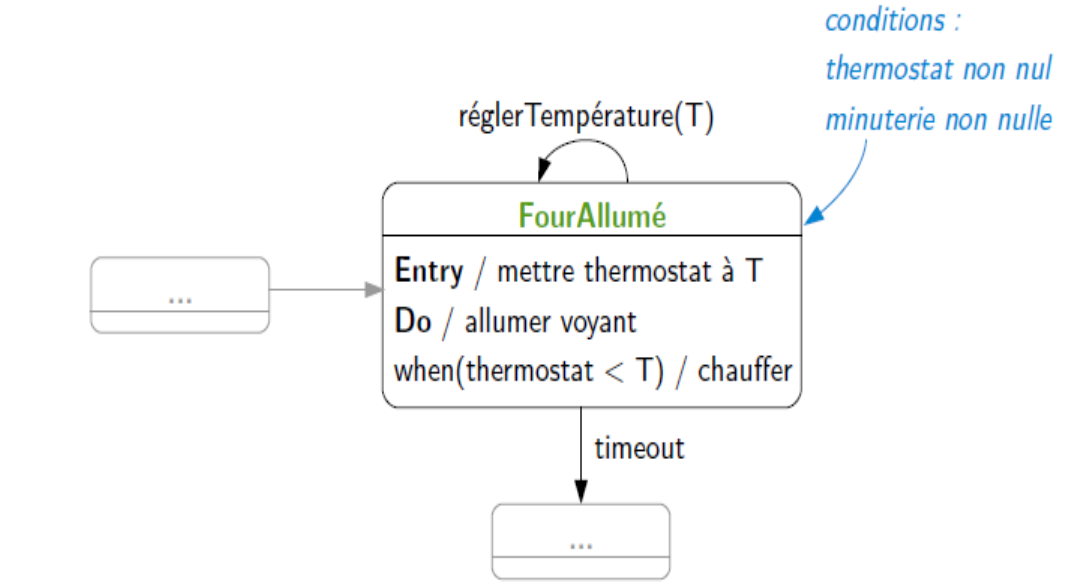


Figure 69 : exemple d'un état intermédiaire.

Sur la figure 70, un événement de timeout a été ajouté au modèle faisant sortir l'objet de cet état si un délai s'écoule. En sortant de cet état, le thermostat est remis à zéro.

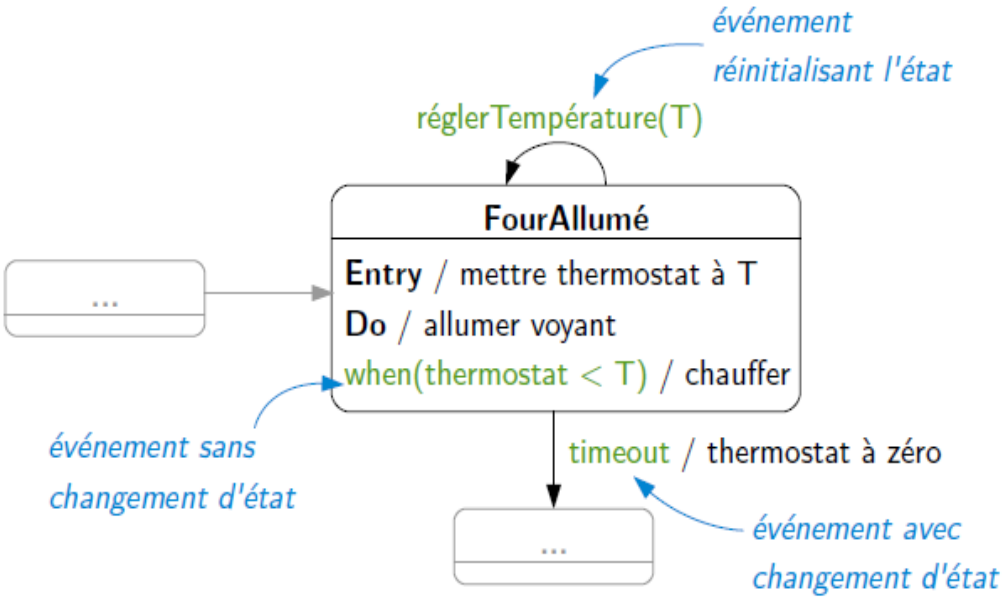


Figure 70 : exemple 2 d'un état intermédiaire.

Exemple complet

Sur la figure 71, on présente un exemple complet d'un diagramme état-transition ou state-char. Il décrit les états par lesquels passe distributeur de billet pour faire un retrait d'argent par une carte. Au début, le distributeur sera allumé et passe à l'état Inactif. De ce dernier, un événement est possible : InsérerCarte. Cet événement peut ne pas changer d'état si la carte et non valide, ou il peut faire transiter le système vers un autre état : CarteValidé. A l'entrée de ce dernier état, le système intialise le nbEssais à 0. Un événement internet et dit saisirCode augmente le nbEssais à chaque code erroné entré. Si ce nombre d'éssais dépasse 2, alors le système revient à l'état Inactif. Par contre, si le code saisi est valide alors le système transite vers un autre état : Code Validé, et à partir de ce nouveau état un événement externe : choisirMontant permet de passer à un état Transaction. choisirMontant exécute une action : demander autorisation. A l'entrée de l'état Transaction, le système commence par rendre la carte tout d'abord. Ensuite, le système se retrouve encore à l'état Inactif, suite à l'événement : autorisation. Cette autorisation peut être refusée, ou accordée. En cas où elle est accordée, une action délivrerBillets est exécuté avant de se remettre à l'état Inactif. Enfin, le système peut s'éteindre suite à un événement non conditionnelle : éteindre.

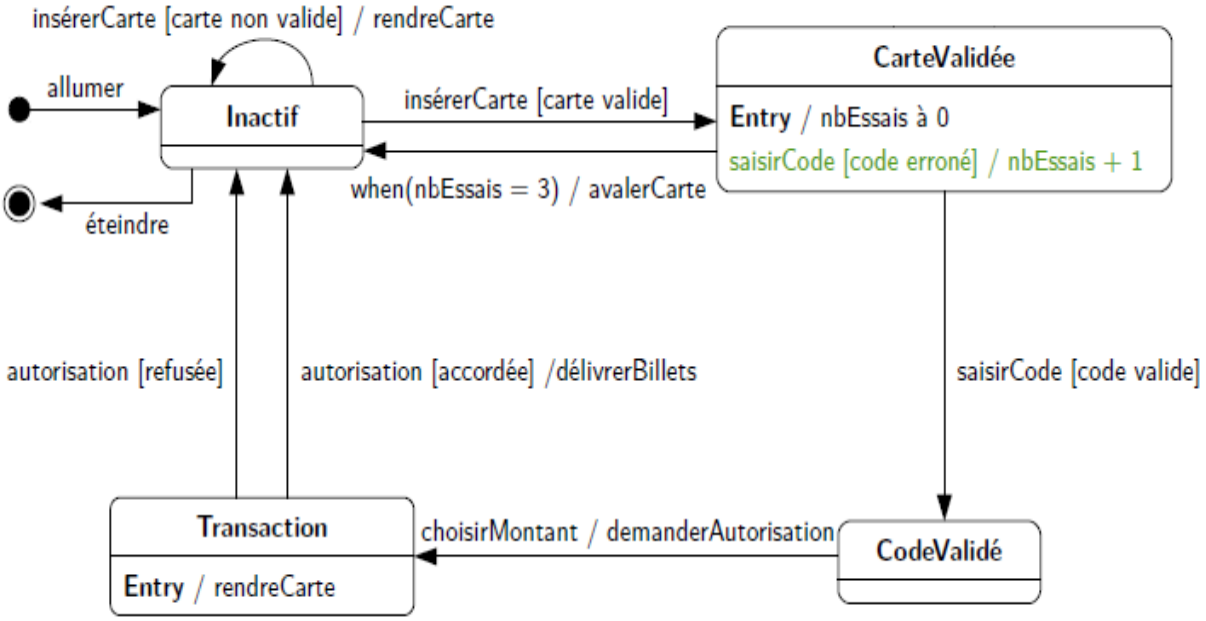


Figure 71 : exemple complet d'un SC.

Modèle état transition imbriqué

Il est possible d'avoir une vision hiérarchique dans un diagramme état transition. Il est possible d'avoir des états abstrait englobant tout un autre diagramme état

transition. Le diagramme interne sera déclenché suite que le système entre dans cet état abstrait. Voir la figure 72.

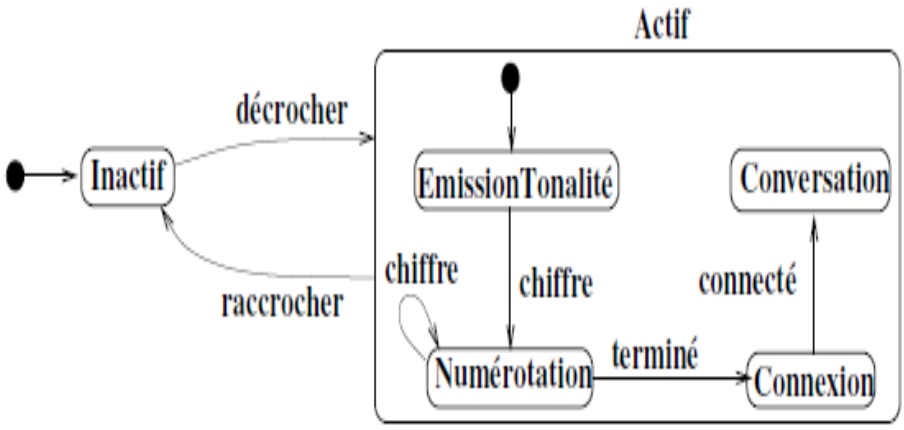


Figure 72 : SC imbriqué.

Dans la figure 72, on modélise le comportement d'un appareil téléphonique dans un processus d'appel. Une fois cet appareil décroché, le téléphone est dans l'état Actif. Ce dernier état est vu comme englobant tout un comportement modélisé sous forme d'un diagramme état-transition. Il est possible de sortir de cet état suite à l'événement raccrocher. Ceci veut dire, que le diagramme état transition interne peut être quitté à n'importe quel moment suite à l'événement raccrocher.

Sur la figure 73, des comportements concurrents sont englobé dans un état composite. A l'entrée de cet état, il est possible de faire l'un des deux comportements.

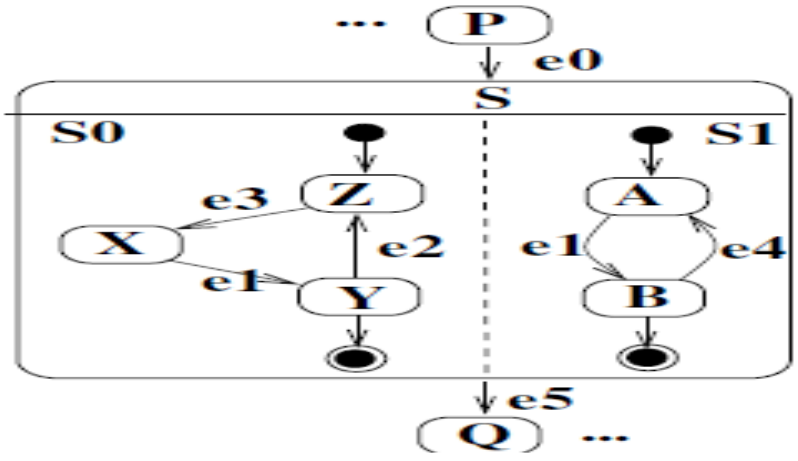


Figure 73 : SC avec concurrence.

8. Conclusion

Dans ce chapitre, nous avons présenté le langage UML (Unified modeling language). C'est un langage adopté depuis 1997, en unifiant plusieurs notations connues dans le monde de la modélisation orientée objet, auparavant. UML n'est donc

qu'un ensemble de diagrammes permettant de modéliser différents aspects d'un système orienté objet. Plusieurs diagrammes sont proposés dans ce langage, mais on peut les subdiviser en général en deux grande familles : des diagrammes dédiés à représenter l'aspect statique et structurel du système (exemple : diagramme de classe, objet, composant, etc) et ceux dédiés à modéliser tout ce qui est dynamique interne des objets (comme diagramme de state chart ou état transition) ou bien des interactions entre objet (cas des diagrammes de séquence).

Le langage UML peut être utilisé pour modéliser et décrire le système, ses besoins, ses services, ses acteurs et utilisateur que se soit dans la phase analyse (cas des diagrammes use cases), conception (cas des diagrammes de conception : objet, classes, séquence, etc) où même en explicitant le déploiement et l'installation. Ce langage est le langage le plus standard et les plus adopté dans la communauté de développement de logiciel et il est parrainé et maintenu par l'organisation OMG (Object Management Group) https://www.omg.org/.

Conclusion Générale

Conclusion générale

A travers ce support de cours, on a présenté l'ensemble de connaissance de base qu'un étudiant informaticien doit savoir sur le domaine de génie logiciel (GL). Ce document permet à son lecteur de comprendre tout d'abord les motivations et les raisons de l'apparition et de l'adoption du génie logiciel dans le monde de l'informatique. Cette adoption est due à la crise logicielle caractérisant la fin des années 60. Le GL a permis de changer la vision des informaticiens vis-à-vis la production de logiciels. Selon le GL, il s'agit d'un processus de manufacturing (ou processus logiciel) exigeant plusieurs compétences, plusieurs moyens financiers et aussi un calendrier rigoureux.

Le GL introduit des approches pour définir le processus logiciel et des modèles pour gérer ce processus. Plusieurs approches et modèles existent dans la littérature et qui sont mentionnés dans ce document. Ces approches définissent plusieurs activités qui le développeur doit appliquer pour obtenir son produit final : Analyse, conception, spécification, codage, teste, installation, validation, vérification, etc.

Afin, d'exprimer les besoins des clients en phase d'analyse, les représentations conceptuelles des services et fonctionnalités en phase de conception, et le déploiement ou l'installation en phase final, le développeur a besoin d'utiliser des langages de description, de spécification et de modélisation. Le langage UML (unified modeling language) est vu comme un standard assurant tout ça en GL. L'UML est donc traité dans le troisième chapitre de ce document avec suffisamment de détails permettant au lecteur d'apprendre ses principes et de découvrir quelques uns de ses diagrammes les plus importantes dans les phases d'analyse et de conception.

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

Annexe A :
Séries d'exercices

TD N°1 : Introduction à l'analyse des problèmes

Question de révision :

1) Proposer les raisons (au moins 2 raisons) motivant l'apparition du GL ?
2) Que voulons dire par Crise Logicielle ?
3) Proposez quelques échecs (2 au moins échecs) enregistrés dans l'histoire de l'informatique.
4) Expliquer le concept de fiabilité en GL?
5) Pourquoi le GL est considéré comme est une science multidisciplinaire ?

Exercice :

La dichotomie étagée : La dichotomie étagée est l'une des techniques de recherche dans les tableaux triés. Elle consiste à subdiviser le tableau en un ensemble d'étages (k étage, et on parle de dichotomie étagée k). Ensuite de localiser la valeur recherchée dans quelle zone se trouve-t-elle. On refait la même subdivision pour la zone trouvée (en k étages). Et de refaire la même chose pour la zone localisée. On arrête la subdivision quand la zone localisée est indivisible par k, où on fait une recherche séquentielle de la valeur.

Exemple : On veut chercher la valeur **3** dans le tableau T de 15 éléments :

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	2	2 ,5	2,75	3	8	12	23	45	78	90	111	130	145	178

- Pour un k=5. Donc on doit subdiviser le tableau : 15/5=3. Donc chaque étage doit contenir 3 éléments. Les cinq zones seront : [1,3], [4,6], [7,9], [10,12], [13,15] On doit comparer 3 avec les bornes de ces étages pour savoir elle est où : 3>T*1+ et 3 >T*3+, donc 3 n'est pas dans le premier étage. T*4+<3<T*5+, donc 3 peut exister dans ce deuxième étage … on arrête la comparaison avec les autre bornes.
- On essaye de subdiviser cet étage : on voit que cet étage contient trois éléments : T[4], t[5], T*6+ c'est indivisible par 5 donc on fait une recherche séquentielle pour trouver que 3=T[5].

Questions :
1. C'est quoi l'avantage de cette technique de recherche ?
2. Faites une analyse de ce problème:

a) Proposer une décomposition fonctionnelle de ce problème: les fonctions (tâches) que le programme doit assurer. Représenter ceci sous forme de diagrammes de boites (une fonction sous formes de boite, par exemple).

b) Y'a-t-il des choix à prendre par le programmeur dans la résolution de ce problème ? Donnez des exemples. Et faites vos choix.

3. Enrichir la décomposition faite dans la question (2.a), et proposer les algorithmes et les structure de donner convenables pour résoudre le problème de la dichotomie étagée.

4. Exécuter cette solution sur des exemples et voir si votre solution est correcte pour ces exemples.

5. Choisissez un langage de programmation (de préférence Java) et écrire le programme.

Bonne chance.

Corrigé type : TD N°1 : Introduction à l'analyse des problèmes

Question de révision : Cours : (L'objectif de ces questions est de faire une forme de rappel des notions déjà présentés dans le cours. Les étudiants doivent participer et donner des réponses. Vous pouvez même désigner des étudiants (avec leurs noms et prénom dans la liste de présence), et leurs demander de répondre. Ceci peut être noté pour préparer la note de l'interrogation. C'est à vous de choisir bien sur**).**

1) Proposer les raisons (au moins 2) pour laquelle le GL a apparu ?

   Plusieurs réponses : L'étudiant peut répondre : « Crise logicielle (tout court) », mais le mieux est de donner d'autres réponses comme : (1) assurer la fiabilité des logiciels, (2) contrôler les délias de livraison, (3) estimer et contrôler les coûts de développement de logiciel, (4) gérer le développement (gestion et organisation de l'équipe de développement, …) …

2) Que voulons dire par Crise Logicielle ?

   Réponses : Crise Logicielle c'est une période dans laquelle les utilisateurs et développeur ont senti que les logiciels ne sont plus des produits parfaits : non fiables, leurs calendrier et leur budgets rarement respectés.

3) Expliquer le concept de fiabilité en GL?

   Réponse : Fiabilité en GL= une qualité selon laquelle le logiciel ne doit tomber en pannes et doit répondre aux besoins de son utilisateur.

4) Proposez quelques échecs (2 au moins) qui ont été enregistré dans l'histoire de l'informatique.

   Réponse : Plusieurs exemples ont été présentés dans le cours. Des exemples : le système IBM 360 que IBM a voulu réaliser mais qui n'étais jamais comme voulu (problème de coût,

de mémoire, d'erreurs), le langage de programmation PL/1 que les développeurs ont voulu réaliser pour qu'il soit un langage unificateur mais qui n'était jamais achevé.

5) Pourquoi dit-on que le GL est une science multidisciplinaire ?
   Car elle n'exige pas uniquement la compétence de programmation, mais d'autre compétences : analyse (pour bien comprendre le problème à résoudre), gestion (pour gérer l'équipe de développement, le calendrier de réalisation, et aussi le budget offert), psychologie et sociologie (pour pouvoir interagir avec un client non informaticien, pour pouvoir coopérer avec des partenaires de travail, pour pouvoir gérer les problèmes de nature humaines qui peuvent se poser dans toute grande réalisation où des chantiers collaborent ensemble).

Exercice 1 : (Recherche par dichotomie étagée : ce n'est pas un exercice de POO mais un exercice d'analyse de problème, et d'algorithmique)

La dichotomie étagée ?

La dichotomie étagée est l'une des techniques de recherche dans les tableaux triés. Elle consiste à subdiviser le tableau en un ensemble d'étages (k étage, et on parle de dichotomie étagée k). Ensuite de localiser la valeur recherchée dans quelle zone se trouve-t-elle. On refait la même subdivision pour la zone trouvée (en k étages). Et de refaire la même chose pour la zone localisée. On arrête la subdivision quand la zone localisée est indivisible par k, où on fait une recherche séquentielle de la valeur.

Exemple : On veut chercher la valeur **3** dans le tableau suivant :

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	2	2 ,5	2,75	3	8	12	23	45	78	90	111	130	145	178

Pour un k=5.

Donc on doit subdiviser le tableau : 15/5=3. Donc chaque étage doit contenir 3 éléments.
Les cinq zones seront : [1,3], [4,6], [7,9], [10,12], [13,15]

On doit comparer 3 avec les bornes de ces étages pour savoir elle est où :

3>T[1] et 3 >T[3], donc 3 n'est pas dans le premier étage.
T[4]<3<T[5], donc 3 peut exister dans ce deuxième étage … on arrête la comparaison avec les autre bornes.

On essaye de subdiviser cet étage : on voit que cet étage contient trois éléments : T[4], t[5], T[6] c'est indivisible par 5 donc on fait une recherche séquentielle pour trouver que 3=T[5].

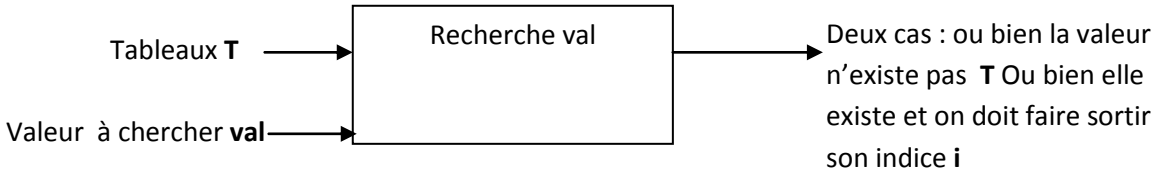
C'est quoi l'avantage de cette technique de recherche ?

C'est clair que c'est une technique qui doit permettre de minimiser le temps de recherche par rapport à la recherche séquentielle.

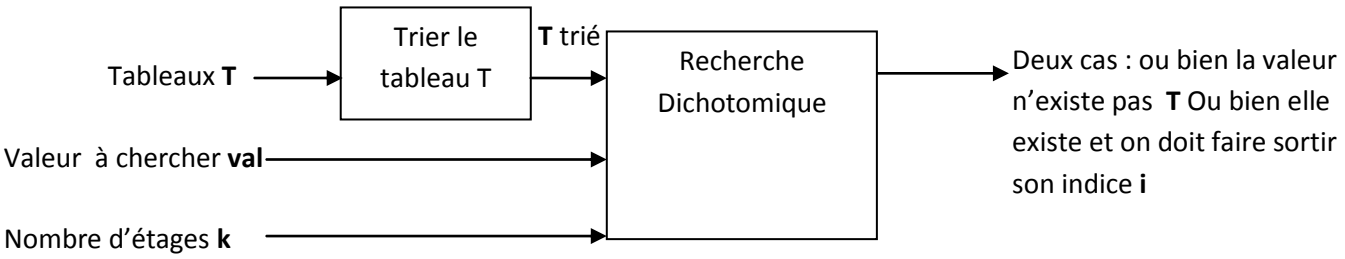
1) **Faites une analyse à ce problème:**

   a) **Proposer une décomposition fonctionnelle de ce problème: les fonctions (tâches) que le programme doit assurer. Représenter ceci sous forme de <u>diagrammes de boites</u> (une fonction sous formes de boite, par exemple).**

Tout d'abord, on peut considérer que le problème est une seule boite noire :



Cependant cette recherche exige que le tableau soit trié, donc on peut raffiner la solution vers:



On peut continuer ce raffinement par exemple, on proposant la décomposition de la tache trier tableau en :

1) Lire le tableau : donc définir la taille de ce tableau, définir le type de ses éléments.
2) Trier le tableau : dans ce cas : quel genre de tri sera choisi (question b)

Pour la recherche par dichotomie étagée :

1) On peut la subdiviser en trois parties aussi : lecture des données : le tableau trié (dans ce cas il peut être vu comme un paramètre ou dès le début trié),
2) Surtout aussi, on doit voir qu'il s'agit d'un processus qui se répète : subdivision : donc un choix est à faire : itératif ou récursif ???

   b) **Y'a-t-il des choix à prendre par le programmeur dans la résolution de ce problème ? Donnez des exemples. Et faites vos choix.**

Le problème nous impose de faire pas mal de choix, sur le plan **traitement** et aussi sur le plan de **données** et leur représentation:

- Les données : le tableau : on peut le représenter de manière contigu (statique), ou dynamique (sous forme d'une liste). Il est clair que travailler avec une structure statique est plus facile dans notre cas.
- Les traitements eux, par contre, nous offrent plusieurs choix :
  - le tri du tableau peut être effectué par plusieurs techniques : tri par sélection (pour un tableau T[1..N], chercher le minimum dans le tableaux, échanger les position de ce minimum avec l'élément T[1], ensuite refaire la même méthode pour le tableau T[2..N]), tri par insertion (on commence par trier T[1] et T[2], ensuite on insère l'élément T[3], donc les élément T[1],T[2], T[3] sont maintenant triés, on continue avec le reste des éléments), …
  - La recherche dichotomique aussi exige des subdivisions répétitives. Ces subdivisions peuvent se faire de manière itérative ou de manière récursive.

2) Enrichir la décomposition faite dans la question (2.a), et proposer les algorithmes et les structure de donner convenables pour résoudre le problème de la dichotomie étagée.

Dns cette phase, on commence à établir un raffinement de la solution. La dichotomie étagé k, peut être divisé en une suite d'étapes qui représente séquentiellement la solution cherchée :

Données enrichies :

- Entrée de la fonction de dichotomie : Tableau **T** de **1..N** éléments d'un type **TYPE**, et d'une valeur **VAL** ;
- Sortie : deux informations : Existence ou non de la valeur cherchée (on peut parlée d'une valeur booléenne **Existe**) dans le cas où cette valeur existe une autre information est importante comme sortie **: Position.**

Etape de la solution :

1- Voir si le nombre **N** d'élément dans le tableau < k, dans ce cas on ne peut que faire une recherche séquentielle dans le tableau … le problème de la dichotomie n'est pas posé.
2- Sinon : On doit commencer par définir les bornes des différents étages, pour faire les comparaisons :
   Chaque étage contient **N / K** éléments.
   Donc les étages auront les bornes suivantes :
   étage 1 = 1 .. (N/K)
   étage 2= (N/K)+1 .. 2 * (N/K)
   étage 2= 2 * (N/K)+1 .. 3* (N/K)
   …
   étage k= (k-1)*(N/K)+1 .. N
   (*Attention : on n'a pas besoin de faire les comparaisons avec toutes les bornes, il suffit de trouver deux bornes qui contiennent* **VAL**)
3- On commence à voir comment localiser **VAL** dans l'un des étages :

Si T[1] <=VAL<=T[N/K] alors
Relancer une recherche par dichotomie étagé depuis l'étape 1, mais avec un sous tableau T[1..N/K]

Sinon : refaire le même traitement depuis l'étape 3 avec le sous tableau suivant

Plusieurs raffinements peuvent être effectué maintenant, jusqu'à arriver à une solution. Je vous laisse le choix maintenant de choisir comment continuer le raffinement de la solution et comme se rapprocher de la solution finale.

Exemple d'une solution (on suppose que le tableau est trié):
==========

1- Nombre_éléments_tableau=borne_ finale -borne_initiale+1 ; // on fait comme ça car on doit le recalculer plusieurs fois, selon les étages, et les niveaux
2- **Si** nombre_éléments_tableau>k **alors**
// On peut faire la dichotomie étagée, donc on doit localiser l'étage
// On peut par exemple utiliser une boucle pour voir quel étage contient **VAL**
// trouver l'étage veut dire trouver deux nouvelles valeurs pour borne_intiale et //borne_finale
// si VAL n'existe en aucun étage, donc on va sortir de la boucle pour une autre raison //qui est : borne_intiale>=N

Nombre_éléments_par_étage= borne_finale /K ;
Etage_trouvée=faux
TANQUE (non Etage_trouvé **ET** borne_intiale< N **) FAIRE**

SI T[borne_intiale] <=VAL=<T[borne_finale /K] **ALORS**
Etage Trouvée ; **REPRENDRE** depuis 1 ;
Sinon
// voir un autre étage
borne_intiale= borne_finale+1 ;
borne_finale= borne_finale+Nombre_éléments_par_étage+1 ;

FINSI
FIN_TANQUE ;
Sinon

// pas de dichotomie étagée mais une recherche séquentielle de la borne initiale vers la //borne finale

---

Le mot **REPRENDRE** indique un autre traitement itératif. Deux solutions seront maintenant possibles : itérative ou récursive. La solution récursive est plus facile car c'est la plus naturelle.

Pour la solution itérative :

On doit définir donc une boucle qui itère en profondeur du tableau pour trouver des sous étage. Elle doit s'arrêter soit quand la subdivision est impossible ou que **VAL** est trouvée.

Exemple d'une autre solution plus raffinée (complètement itérative):

1- Nombre_éléments_tableau=borne_ finale -borne_initiale+1 ; // on fait comme ça car on doit le recalculer plusieurs fois, selon les étages, et les niveaux

TANQUE (NON Trouvé **ET** nombre_éléments_tableau>k**) FAIRE**
// On peut faire la dichotomie étagée, donc on doit localiser l'étage
// On peut par exemple utiliser une boucle pour voir quel étage contient **VAL**
// trouver l'étage veut dire trouver deux nouvelles valeurs pour borne_intiale et //borne_finale
// si VAL n'existe en aucun étage, donc on va sortir de la boucle pour une autre raison //qui est : borne_intiale>=N

Nombre_éléments_par_étage= borne_finale /K ;
Etage_trouvée=faux
TANQUE (non Etage_trouvé **ET** borne_intiale< =N **) FAIRE**

SI T[borne_intiale] <=VAL=<T[borne_finale /K] **ALORS**
Etage Trouvée ;
///////**REPRENDRE** depuis 1 ; remplacée par :
Nombre_éléments_tableau=borne_ finale -borne_initiale+1 ;
Sinon
// Voir un autre étage
borne_intiale= borne_finale+1 ;
borne_finale= borne_finale+Nombre_éléments_par_étage+1 ;

FINSI
FIN_TANQUE ;

FIN_TANQUE ; ////// ici on sort soit avec un étage localisé ou avec une décision de non ////existence de **VAL**

Si borne_initiale<N **ALORS /// la cause de la sortie de la boucle non pas la non existence de /// la valeur**

// pas de dichotomie étagée mais une recherche séquentielle de la borne initiale vers la //borne finale

---

3) **Exemples d'exécution**
   Au choix.
4) **Choisissez un langage de programmation (C, Pascal, ou Java) et écrire le programme.**

Ça sera au choix des étudiants pour cette fois, je pense qu'ils connaissent seulement le langage C.

Travaux Dirigé N°2 : GL + rappel sur la POO

Questions de Cours :

(I) GL (rappel)

1) Que signifie l'invisibilité du logiciel ? Pourquoi ceci est un grand problème pour sa production ? Comment peut-on vaincre ce défi ?
2) Expliquer la notion de flexibilité du logiciel. Donner un exemple. Pourquoi ceci risque de toucher à la fiabilité du logiciel ?
3) Expliquer le concept du processus logiciel. C'est quoi l'objectif du GL vis-à-vis ce processus ?
4) Expliquer le concept d'approches de développement ? Donner des exemples.

(II) POO (révision du S1 et mise au point des concepts)

1) Citer deux avantages de la programmation OO vis-à-vis celle procédurale ?
2) Quelle est la différence entre un objet et une classe ? donner des exemples.
3) Que veut dire une relation d'héritage entre classes ? Donner des exemples.
4) Citer les types d'héritages possibles entre classes ? Donner des exemples. Expliquer les problèmes (potentiels) de chacun de ces types.
5) Que signifie un **modificateur** dans le langage java? Donner quelques exemples de modificateurs.
6) Expliquer les concepts de **private**, **public**, et **static** en java. Donner des exemples.
7) Expliquer le concept de polymorphisme? Quels sont les types de polymorphisme ? Donner des exemples sur chacun de ces types. C'est quoi l'avantage d'avoir ce mécanisme dans la programmation ?

Exercices :

Exo 1.

Soit l'objet cercle dont les cordonnées du centre sont (x0, y0), et le rayon est r. On est intéressé à faire un ensemble d'opérations sur cet objet : création, changement de coordonnées et du rayon, calcul de la surface.

1) Proposer une solution orientée objet pour ce problème ?
2) Etendre cette solution pour prendre en considération les couleurs des cercles.
3) Pouvez-vous étendre cette solution pour traiter des sphères et non plus des cercles ? Proposer votre solution.

Exo 2.

On veut écrire un programme OO pour la résolution d'une équation de deuxième degré : $Ax^2+Bx+C=0$;

1) Proposer les classes nécessaires ? écrire le programme OO.
2) Y'a-t-il une idée pour exploiter l'héritage dans la résolution de ce problème ? Si oui, proposer une solution avec héritage.

Exo 3.

Soit le programme java suivant

```
class TableauParam {

static void listerTableau (float[] note, int[] coef) {
     System.out.println ("\nNote\tCoefficient");
     for (int i=0; i < note.length; i++) {
        System.out.println (note[i] + "\t\t" + coef[i]);
          }
}
static void modifCoeff (int[] coeff, int num, int valeur) {
```

```
        coeff [num] = valeur;
            }
public static void main (String[] args) {
            int[] coef = {1, 3, 2};
            float[] note = {10.75f, 12f, 8.25f};
            listerTableau (note, coef);
            modifCoeff (coef, 1, 4); // le coefficient 1 vaut 4
            listerTableau (note, coef);
           } // main
}
```

Questions :

1) Expliquer exactement l'objectif de ce programme.
2) Justifier les différentes occurrences du mot clé **static**.
3) Y'a-t-il un constructeur dans cette classe ? Comment peut-on créer des objets de cette classe ?

Exo 4. Soit le programme java suivant :

```
class Reel {
float valeur;
}
class SomMoyOK {

      static int somme (int[] tab, Reel moyenne) {
            int somTab = 0;
            int n = tab.length;
            for (int i=0; i < n; i++) somTab += tab[i];
            moyenne.valeur = somTab/n;
            return somTab;
      }

      public static void main (String[] args) {
            int somVect;
            Reel moy = new Reel();
            int[] vecteur = {10, 15, 20};
            somVect = somme (vecteur, moy);
            System.out.println ("Somme " + somVect +
            "\nMoyenne " + moy.valeur);
      }
      }
}
```

Questions :

1) C'est quoi l'objectif de ce programme?
2) Combien y'a-t-il de classes dans ce programme?
3) Quelle est la portée de l'attribut valeur de la classe Reel ?
4) Expliquer le rôle du texte souligné.

Exo 5.

On veut réaliser un programme orienté objet pour implémenter une structure de pile d'entiers. Les opérations possibles sont :

```
void initPile (int max);          // initialiser la pile
int pileVide ();                  // la pile est-elle vide ?
void empiler (int valeur);        // empiler une valeur
int depiler (int* valeur);        // dépiler à l'adresse de valeur
void viderPile ();                // vider la pile
void listerPile ();               // lister les éléments de la pile
```

1) Proposer une classe avec les attributs et les méthodes nécessaires à l'implémentation de cette classe.

Exo 6.

On veut simuler un écran graphique comportant un certain nombre de **lignes** et de **colonnes**. Chaque point de l'écran contient un caractère correspondant à un pixel. On dessine sur cet écran à l'aide d'un jeu de fonctions définies comme suit :

- **Ecran** (int nbLig, int nbCol): crée un écran de nbLig lignes sur nbCol colonnes.
- **effacerEcran** (): efface l'écran pour un nouveau dessin.
- **crayonEn** (int numLigCrayon, int numColCrayon): positionne le crayon sur une ligne et une colonne.
- **changerCouleurCrayon** (int couleurCrayon): change la couleur du crayon.
- **tracerCadre** (): trace un cadre autour de l'écran.
- **afficherEcra** (): affiche le contenu de l'écran (les caractères de l'écran).

Questions:

1) Proposer les attributs nécessaires.
2) Rédiger une implémentation possible de cette classe. Veuillez utiliser le mot clé **this**.

Exo 7.

On s'intéresse à réaliser la classe des nombres complexes. Un nombre complexe est composé d'une partie réelle, et d'une partie imaginaire. Les méthodes qu'on propose pour cette classe sont les suivantes :

- double **module** () : fournit le module du nombre complexe (le module de x+y*i est la racine de x^2+y^2).
- Complex **oppose** () : fournit l'opposé du nombre complexe (l'opposé de x+y*i est –x-y*i).
- Complex **conjugue** () : fournit le conjugué du nombre complexe (le conjugué de x+y*i est x-y*i)
- Complex **plus** (Complex z) : fournit l'addition du nombre complexe et de z.
- Complex **moins** (Complex z) : fournit la soustraction du nombre complexe et de z.
- Complex **multiplier** (Complex z) : fournit la multiplication du nombre complexe et de z.
- void **ecritC** () : écrit le nombre complexe sous la forme (pReel + pImag i).

Questions :

1) Proposer une réalisation en java de la classe complexe.
2) On veut avoir une méthode **plus** permettant d'additionner deux arguments complexes: z1 et z2. Utiliser le mécanisme de **surcharge** pour ajouter cette méthode à la classe complexe. Quelle est le modificateur qui doit être ajouté lors de la déclaration de cette méthode ? Justifier.
3) Proposer une utilisation de la classe complexe.
4) Ajouter un deuxième constructeur à la classe complexe permettant de créer des complexes à partir d'un **module** est d'un **argument**. Ce constructeur prend un troisième paramètre qui est le type de l'unité de mesure de l'argument (**degrés** ou **radians**).
5) Quel effet aura la ligne suivante dans la classe Complex: **this(0,0) ;** ?

```
class complex {
float x; float y;
complex(float x, float y){
this.x=x; this.y=y;
}
complex(float z){
this(0,0);
this.x=z;
}
```

Exo 8.

Une personne dispose d'un **nom**, d'un **prénom**, et des ses deux **parents** qui sont de leurs part des personnes.

Questions :

1) Proposer une classe **personne** permettant de créer des personnes. Cette classe doit permettre de créer des personnes sans parents. (essayez d'avoir deux constructeurs différents. L'usage de la **surcharge** et de l'opérateur **this** est obligatoire là où il est possible de les utiliser).
2) Utiliser la classe personne, déjà définie, pour créer la personne **Ahmed**, la personne **Lila** fille de **Ahmed**, la personne **Toufik** fils de **Mustapha** et de **Assia**, et la personne **Mohamed** fils de **Ahmed** et **Khadidja**.

Exo 9.

Soit le programme suivant :

```
class PPPersonne2 { //Programme Principal Personne 2
        public static void main (String[] args) {
                Personne jules = new Personne ("Durand", "Jules");
                Personne berthe = new Personne ("Dupond", "Berthe");
                Personne jacques = new Personne ("Durand", "Jacques",
                        jules, berthe);
                System.out.print (berthe.nom + jacques.nom + jacques.pere +
                                        jacques.mere);
        Personne p1 = jules ;
                Personne p2 = new Personne (jacques.nom, "Jacquot").
        }
}
```

Questions :

1) Expliquer ce programme. Montrer les résultats de son exécution.
2) C'est quoi la différence principale entre les deux variables p1 et p2.

Exo 10.

1) Proposer une classe date permettant de faire les services suivants :

- **Date** (int j, int m, int an) : constructeur créant un objet Date.
- String **toString** () : chaîne de caractères correspondant à la date de l'objet.
- boolean **bissex** () : fournit **true** si l'année est bissextile, false sinon.
- int **nbJoursEcoules** () : nombre de jours écoulés depuis le début de l'année.
- int **nbJoursRestants** () : nombre de jours restant dans l'année.

2) On veut avoir en plus des services précédents, les deux services suivants :

- boolean **bissex** (int annee) : fournit **true** si l'année est bissextile, false sinon.
- long **nbJoursEntre** (Date date1, Date date2) : fournit le nombre de jours écoulés entre les deux dates date1 et date2.

Quels modificateurs sont nécessaires pour ces deux méthodes ?

Exo 11. On veut réaliser un programme qui calcule la moyenne d'un **étudiant**. Proposer un programme Java permettant de faire ceci. On exige trois classes : **Module**, **Enseignant**, et enfin **Etudiant**.

Bonne chance.

Travaux Dirigé N°2 : Corrigé Type

Questions de Cours : (Réponses)

GL (rappel)

5) **Que signifie l'imprévisibilité du logiciel ?**
   On ne peut pas voir le logiciel durant son développement (seulement un code source incomplet et non compris par le client).
   Pourquoi ceci est un grand problème pour sa production ?
   Car on risque de découvrir tardivement des erreurs, et des besoins non satisfaits du client
   Comment peut-on vaincre ce défi ?
   Il y'a des techniques de prototypage, de maquettages, de modélisation pour représenter le logiciel avant sa terminaison, et donc d'avoir une prévision.
6) **Expliquer la notion de flexibilité du logiciel.**
   Souplesse, et où des petites modifications altèrent le logiciel
   Donner un exemple.
   Un changement d'un < vers un <= peut changer le fonctionnement
   Pourquoi ceci risque de toucher à la fiabilité du logiciel ?
   Car ces modifications sont tellement importantes et en même temps facile à se produire durant le développement (erreur de transcription, par exemple).
7) **Expliquer le concept du processus logiciel.**
   C'est la dynamique (la suite des tâches et d'activités) permettant de produire le logiciel
   C'est quoi l'objectif du GL vis-à-vis ce processus ?
   Définition de ce processus (des activités donc), et gestion de ce processus
8) **Expliquer le concept d'approches de développement ?**
   Approche= ensemble d'activités, méthodes, outils exploité selon une certaine politique pour construire un logiciel
   Donner des exemples.
   Cascade, programmation exploratoire, …

POO

1) **Citer deux avantages de la programmation OO vis-à-vis celle procédurale ?**
   - Plus naturelle (proche du monde réel, donc les objets du monde réel seront représentés directement comme des objets informatiques).
   - Réutilisabilité (grâce à l'héritage), donc gain du coût dans le développement.
2) **Quelle est la différence entre un objet et une classe ?**
   Classe= moule, type, mais un Objet= instance, réalisation
   Donner des exemples.

3) **Que veut dire une relation d'héritage entre classes ?**

   Donner des exemples.

4) **Citer les types d'héritages possibles entre classes ?**
   Simple, multiple
   Donner des exemples.

   Expliquer les problèmes (potentiels) de chacun de ces types

Simple : insuffisant quelques fois

Multiple : compliqué à implémenter, par exemple problème de confusion dans les appels des méthodes qui portent les mêmes noms dans différentes classes mères.

5) **polymorphisme**=poly+morphisme=plusieurs formes. En général c'est un mécanisme permettant à un même nom de méthode d'avoir plusieurs réalisations possibles ;
6) trois types :
   -polymorphisme par surcharge : surchager le même nom par plusieurs implémentation même dans la même classe :
   + : int x int → int
   + : réel x réel → réel
- polymorphisme par sous-typage : si la classe A définit une méthode M1 et la classe B hérite A, alors si a est un objet de A et b est un objet de b, alors : a.M1 et b.M2 représente un polymorphisme par sous typage. La méthode M1 est exécuter sur deux type différents.
- Polymorphisme par généricité : défini quand une méthode prend des arguments du genre générique. Exemple : un tableau T est générique si les ses éléments peuvent être des entiers, ou des réels… selon un certain paramètre. Donc une méthode M(T) est dite polymorphisme car T peut être un tableau d'entiers, de réel ou de caractère …

7) **Ça facilite la programmation.**

Exo 2.

On veut écrire un programme OO pour la résolution d'une équation de deuxième degré : $Ax^2+Bx+C=0$;

1) Proposer les classes nécessaires ? écrire le programme OO.

```
public class equation {

      float a;
```

```
	float b;
	float c;

	float delta(){
		return b*b-4*a*c;
	}

	void calcul(){
		float delta1=delta();
		if (delta1<0) System.out.print("pas de solution");
		else if (delta1>0) {
			float x1=(float)(-b-Math.sqrt(delta1));
				// on a fait un casting ver float, car le type de
				//-b-Math.sqrt(delta1) est double et non pas
float
				// on a utilisé la classe Math qui contien la
méthode sqrt
				// et comme vous remarquez, il n'y a pas de
procédure en java
				// mais uniquement des méthodes
			float x2=(float)(-b+Math.sqrt(delta1));
			System.out.println(x1);
			       // la méthode println ne peut pas avoir plus d'un
argument de type
				// float
			System.out.println(x2);
		}
		else {
			float x0=-b/(2*a);
			System.out.println(x0);

		}
	}
public static void main(){
	// d'ici commence l'exécution
	// on peut avoir plusieurs possibilité:
	//1) sans créer une classe, mais ici les attributs et les méthodes
doivent être static
	// doivent être static, sinon vous allez avoir des erreurs
syntaxiques
	a=3;b=4;c=5;
	calcul();

	// 2)avec création d'une instance
	equation eq=new equation();
	     // le constructeur par défaut créé implicitement par le système
	eq.a=3;
	eq.b=4;
	eq.c=5;
	eq.calcul();

}
}// fin de la classe
```

Exemple avec constructeur du programmeur

```
public class equation {

	float a;
```

```
	float b;
	float c;

	equation (float a0, float b0, float c0){
		// un constructeur initialisateur ajouté par le programmeut
		a=a0; b=b0; c=c0;
	}
	float delta(){
		return b*b-4*a*c;
	}

	void calcul(){
		float delta1=delta();
		if (delta1<0) System.out.print("pas de solution");
		else if (delta1>0) {
			float x1=(float)(-b-Math.sqrt(delta1));
				// on a fait un casting ver float, car le type de
				//-b-Math.sqrt(delta1) est double et non pas
float
				// on a utilisé la classe Math qui contien la
méthode sqrt
				// et comme vous remarquez, il n'y a pas de
procédure en java
				// mais uniquement des méthodes
			float x2=(float)(-b+Math.sqrt(delta1));
			System.out.println(x1);
				// la méthode println ne peut pas avoir plus d'un
argument de type
				 // float
			System.out.println(x2);
		}
		else {
			float x0=-b/(2*a);
			System.out.println(x0);

		}
	}
public static void main(){

	// avec création d'une instance
	equation eq=new equation(3, 4, 5);
		// le constructeur du programmeur

	eq.calcul();

}
}// fin de la classe
```

1) Y'a-t-il une idée pour exploiter l'héritage dans la résolution de ce problème ? Si oui, proposer une solution avec héritage.

```
class equation_1_degree{
	// supperclasse de la classe equation
	float b,c;
	equation_1_degree(float b0, float c0){
		// un constructeur
		b=b0; c=c0;
	}
	void calcul(){
```

```
            if (b==0)
                  System.out.print("erreur");
            else {
                  float x0=c/b;
                  System.out.println(x0);
            }

      }
}
public class equation extends equation_1_degree{

      float a;     // b et c sont hérités


      equation (float a0, float b0, float c0){
            // un constructeur initialisateur ajouté par le programmeut
            super(b0,c0);
            a=a0;
      }
      float delta(){
            return b*b-4*a*c;
      }

      void calcul(){
            if(a!=0){// il s'agit d'une equation de deuxième degrée
            float delta1=delta();
            if (delta1<0) System.out.print("pas de solution");
            else if (delta1>0) {
                  float x1=(float)(-b-Math.sqrt(delta1));
                        // on a fait un casting ver float, car le type de
                        //-b-Math.sqrt(delta1) est double et non pas
float
                        // on a utilisé la classe Math qui contien la
méthode sqrt
                        // et comme vous remarquez, il n'y a pas de
procédure en java
                        // mais uniquement des méthodes
                  float x2=(float)(-b+Math.sqrt(delta1));
                  System.out.println(x1);
                        // la méthode println ne peut pas avoir plus d'un
argument de type
                          // float
                  System.out.println(x2);
            }
            else {
                  float x0=-b/(2*a);
                  System.out.println(x0);

            }
            }
            else super.calcul();// il s'agit d'une équation de premier
degrée
      }
public static void main(){

      // avec création d'une instance
      equation eq=new equation(3, 4, 5);
           // le constructeur du programmeur

      eq.calcul();
```

```
}
      }// fin de la classe
```

Exo 5 :

```
// Pile.java gestion d'une pile d'entiers
public class Pile {
// sommet et element sont des attributs privés

private int sommet; // repère le dernier occupé (le sommet)

private int[] element; // tableau d'entiers alloué dynamiquement

// erreur est une méthode privée utilisable seulement
// dans la classe Pile

private void erreur (String mes) {
System.out.println ("***erreur : " + mes);
}

public Pile (int max) { // voir 2.2.1 Le constructeur d'un objet
sommet = -1;
element = new int [max]; // allocation de max entiers
}

public boolean pileVide () {
return sommet == -1;
}

public void empiler (int v) {
if (sommet < element.length - 1) {
sommet++;
element [sommet] = v;
} else {
erreur ("Pile saturee");
}
}

public int depiler () {
int v = 0; // v est une variable locale à depiler()
if (!pileVide()) {
v = element [sommet];
sommet--;
} else {
erreur ("Pile vide");
}
return v;
}
```

```
public void viderPile () {
sommet = -1;
}

public void listerPile () {
if (pileVide()) {
System.out.println ("Pile vide");
} else {
System.out.println ("Taille de la pile : " + element.length);
for (int i=0; i <= sommet; i++) {
System.out.print (element[i] + " ");
}
System.out.println(); // à la ligne
}
}

} // class Pile
```

Exo 6.

```
Ecran (int nbLig, int nbCol) { // nbLig désigne le paramètre
      this.nbLig = nbLig ; // this.nbLig désigne l'attribut
      this.nbCol = nbCol ;
      zEcran = new char [nbLig][nbCol]; // référence sur la zone écran
      changerCouleurCrayon (NOIR); // par défaut crayon noir
      crayonEn (nbLig/2, nbCol/2); // au milieu de l'écran
      effacerEcran();
}


// mettre l'écran à blanc
void effacerEcran () { // voir figure 2.5
      for (int i=0; i < nbLig; i++) {
            for (int j=0; j < nbCol; j++) zEcran [i][j] = ' ';
      }
}


// le crayon est mis en numLigCrayon, numColCrayon
void crayonEn (int numLigCrayon, int numColCrayon) {
      this.numLigCrayon = numLigCrayon ;
      this.numColCrayon = numColCrayon ;
}

// la couleur du crayon est couleurCrayon de 0 à 15
void changerCouleurCrayon (int couleurCrayon) {
      if ( (couleurCrayon >= 0) && (couleurCrayon <= 15) ) {
            this.couleurCrayon = couleurCrayon ;
      }
}

// tracer un cadre autour de l'écran
void tracerCadre () {
      for (int nc=0; nc < nbCol; nc++) {
            zEcran [0][nc] = '-'; // le haut de l'écran
            zEcran [nbLig-1][nc] = '-'; // le bas de l'écran
```

```
        }
        for (int nl=0; nl < nbLig; nl++) {
              zEcran [nl][0] = '|'; // le côté gauche
              zEcran [nl][nbCol-1] = '|'; // le côté droit
        }
}



void afficherEcran () {
        for (int i=0; i < nbLig; i++) {
              for (int j=0; j < nbCol; j++) {
                    System.out.print (zEcran [i][j]);
              }
              System.out.print ("\n");
        }
        System.out.print ("\n");
}
```

Exo 7.

public **Complex** (double module, double argument, int typ) {
if (typ == RADIANS) {
pReel = module * Math.cos (argument);
pImag = module * Math.sin (argument);
} else { // en degrés
pReel = module * Math.cos ((argument/180)*PI);
pImag = module * Math.sin ((argument/180)*PI);
}
}

```
class Personne {
        String nom; // les quatre attributs de Personne
        String prenom;
        Personne pere; // Personne contient deux références sur Personne
        Personne mere; // un attribut référence est initialisé à null

Personne (String nom, String prenom, Personne pere, Personne mere) {
        this.nom = nom;
        this.prenom = prenom;
        this.pere = pere;
        this.mere = mere;
        }

// les références de pere et mere sont à null par défaut

Personne (String nom, String prenom) {
// appel du constructeur défini ci-dessus avec 4 paramètres
        this (nom, prenom, null, null);
        }
}
```

Exo10 :

```
class Date {
private int jour;
private int mois;
private int an;
Date (int j, int m, int an) {
     jour = j;
     mois = m;
```

```
        this.an = an;
}
public String toString () {
return " " + jour + "/" + mois + "/" + an;
}
// fournit vrai si année bissextile, faux sinon
static boolean bissex (int annee) {
        if ( (annee % 400) == 0 ) return true;
        if ( (annee % 100) == 0 ) return false;
        if ( (annee % 4) == 0 ) return true;
        return false;
}
// idem ci-dessus pour une Date
boolean bissex () {
        return bissex (an);
}
//fournit le nombre de jours écoulés dans l'année
int nbJoursEcoules () {
        int lgMoisP [] = {0,31,59,90,120,151,181,212,243,273,304,334};
        return lgMoisP [mois-1] + jour + (Date.bissex (an) ? 1 : 0);
}
//fournit le nombre de jours restants dans l'année
int nbJoursRestants () {
        return 365 + (Date.bissex (an) ? 1 : 0) - nbJoursEcoules();
}
//nombre de jours écoulés entre 2 dates date1 < date2
static long nbJourEntre (Date date1, Date date2) {
        int nje1 = date1.nbJoursEcoules ();
        int njr1 = date1.nbJoursRestants ();
        int nje2 = date2.nbJoursEcoules ();
//Les deux dates de la même année
        if (date1.an == date2.an) return nje2-nje1;
//année de date2 supérieure à année de date1
        long nbj = njr1; // fin de la première année
        for (int i=date1.an+1; i < date2.an; i++ ) {
        nbj += 365 + (Date.bissex (i) ? 1 : 0); // les années entières
       }
        nbj += nje2; // fin de la dernière année
        return nbj;
        }
} // classe Date
```

1) Les deux méthodes doivent être **static**. Elles ne s'appliquent pas sur un objet.

Exo 11.

Une solution possible :

```
class professeur {
    String nom, prénom;
    public professeur(String nom, String prénom){
        this.nom=nom;
        this.prénom=prénom;
    }

}



class module {
```

```
    private String Intitulé;
    private float Note_examen;
    private float Note_TD;
    private float Myenne;
    private int Coef;
    private professeur Prof;
    // tout d'abord, on construit le constructeur de la classe
    public module(String Intitulé, float Note_examen, float Note_TD, int
Coef, professeur Prof){
        this.Intitulé=Intitulé;
        this.Note_examen=Note_examen;
        this.Note_TD=Note_TD;
        this.Coef=Coef;
        this.Prof=Prof;
    }
    // les méthodes de consultation
    public String get_Intitulé(){
    return Intitulé;
    }
    public float get_Note_examen(){
    return Note_examen;
    }
    public float get_Note_TD(){
    return Note_examen;
    }
    public int get_Coef(){
    return Coef;
    }
    public professeur get_prof(){
    return Prof;
    }
    // les méthodes de mise à jour
     public void set_intitulé(String Intitulé){
     this.Intitulé=Intitulé;
    }
    public void set_Note_examen(float Note_examen){
     this.Note_examen=Note_examen;
    }
    public void set_Note_TD(float Note_TD){
     this.Note_TD=Note_TD;
    }
    public int set_coef(int Coef){
    return this.Coef=Coef;
    }
    public float get_Moyenne(){
     return (this.Note_TD+this.Note_examen)/2;
    }
    public void set_prof(professeur Prof){
    this.Prof=Prof;
    }

}



class étudiant {
    String nom, prénom;
    module [] modules;// une table de tous les modules que l'étudiant doit
poursuivre
    int nb_modules;// nombre de modules que cet étudiant doit suivre
```

```
    float moyenne;

    public étudiant(String nom, String prénom){
        this.nom=nom;
        this.prénom=prénom;

        // ici tu dois faire entrer tous les modules: intitulé, note TD,
note exam, coéficient, nom du prof
        this.nb_modules=4;
        this.modules= new module [this.nb_modules];

        modules[0]=new module("math", 10f, 12f, 3, new
professeur("Ahmed","Mohamed"));
        modules[1]=new module("phys", 11f, 142f, 3, new
professeur("Leila","Mustapha"));
        modules[2]=new module("chim", 9f, 12f, 2, new
professeur("Ahmed","Mohamoud"));
        modules[3]=new module("algo", 10f, 15f, 4, new
professeur("Toufik","Mohamed"));



    }
    public float moyenne(){
        // ici tu dois calculer la moyenne de l'étudiant, en utilisant la
table modules déjàs définie dans le constructeur
        float somme_notes=0, somme_modules=0;
        for(int i=0; i<nb_modules; i++)
        {

somme_notes=somme_notes+modules[i].get_Moyenne()*modules[i].get_Coef();
            somme_modules=somme_modules+modules[i].get_Coef();
        }

        this.moyenne=somme_notes/somme_modules;
        return moyenne;
    }
    static public void main(String[]args){
        étudiant t=new étudiant("toufik","méchant");
        float m=t.moyenne();
        System.out.println("moyenne="+m);
    }

}
```

TD3 : UML

Questions de cours :

1. En programmation OO :
   - C'est quoi la différence entre une composition et une agrégation ? Donner un exemple.
   - Pourquoi on utilise des classes abstraites ?
2. Mettez « Vrai » ou « Faux » devant chaque phrase :
   - Le diagramme de classe est un diagramme dynamique.
   - Le diagramme de séquence est un diagramme statique.
3. Est-ce que un acteur **doit** avoir un accès direct à un UC qui étend un UC ou un UC inclus dans un autre UC ?

Exercice 1 :

Un système de gestion de commandes gère différents types de commandes. Parmi ces commandes, on distingue la gestion des commandes fournisseurs qui met en relation des acheteurs, des fournisseurs et un secrétaire qui doit gérer les commandes. La gestion de la commande exige le choix d'un fournisseur (un choix qui peut être urgent sous certaines conditions) et la commande du fournisseur.

Question: établir un diagramme de CU pour cette description.

Exercice 2 :

Dans une agence de voyage : Pour organiser un voyage, l'agent doit réserver au client une chambre d'hôtel, lui réserver un billet d'avion ou de train, lui réserver un taxi pour venir de l'aéroport ou de la gare et enfin, lui établir une facture. Certains clients peuvent demander une facture plus détaillée.

Question: Élaborez le diagramme des cas d'utilisation d'une **agence de voyage**.

Exercice 3 :

On veut modéliser avec un DCU les différentes UC, acteur et relation pour un distributeur automatique de billet. Le distributeur doit offrir par exemple deux services : consultation du solde et retrait d'argents, et aussi il a besoin de temps en temps d'être allumé/éteint et d'être revitaliser d'argents. Un retrait d'argent exige la vérification du stock d'argent dans le coffre du distributeur. Les deux opérations retrait d'argent et consultation du solde peuvent donner lieux à l'impression d'un ticket dans

le cas de la disponibilité des papier d'impression dans le distributeur. Ce distributeur peut être utilisé par plusieurs personnes dont deux sont : Client et technicien.

1) Déduire les cas d'utilisations possibles et les relations entre eux si elles existent.
2) Déduire les acteurs possibles et montre les relations entre eux si elles existent.
3) Tracer le DCU et délimite le système (comme package) de ses acteurs.

Exercice 4 :

On désire réaliser une application pour la gestion des Rapports Quotidiens de Vol (RQV) de véhicules dans les départements de police, via le web. On distingue initialement deux types d'utilisateurs pour ce système: **les victimes** et **les témoins**. Chacun de ces utilisateurs peut créer une déclaration de vol, en y indiquant son rôle (victime, témoin ou bien les deux), ses informations personnelles (son n°CIN, nom, prénom, adresse, tél), le type de la propriété volée (véhicule à moteur ou bien bicyclette) ainsi que les différentes informations disponibles qui l'identifient (couleur, marque, numéro de série pour les bicyclettes, matricule pour les véhicules à moteur, description générale), la date, l'heure et le lieu (avec tous les détails disponibles : n° de la rue, ville, code postal,…) du vol.

Le système attribue à chaque déclaration un identifiant, que l'utilisateur peut utiliser pour pouvoir éditer la déclaration (ajouter des informations, supprimer la déclaration), avant de sauvegarder la déclaration. Le système doit enregistrer, pour chaque déclaration, la date de sa dernière modification.

On distingue également un autre type d'utilisateurs : l'agent policier qui se charge de la création des Rapports Quotidiens de Vol. Un RQV est relatif à une date particulière, il contient toutes les déclarations de vols effectuées ou bien modifiées dans ce jour. Lorsqu'un véhicule déclaré est retrouvé, l'agent policier modifier l'état de la déclaration concernée. Evidemment, l'agent policier doit s'authentifier pour pouvoir accéder à cette application.

On désire déterminer pour chaque RQV la liste des nouvelles déclarations, la liste des déclarations mises à jour, ainsi que les déclarations qui ont été résolues.

- Décrire les différentes fonctionnalités de ce système en utilisant un diagramme de cas d'utilisation.

Exercice 5 :

Une entreprise souhaite modéliser avec UML le processus de formation de ses employés afin d'informatiser certaines tâches. Le processus de formation est initialisé quand le responsable formation reçoit une demande de formation d'un employé. Cet employé peut éventuellement consulter le catalogue des formations offertes par les

organismes agréés par l'entreprise. Cette demande est instruite par le responsable qui transmet son accord ou son refus à l'employé. En cas d'accord, le responsable cherche la formation adéquate dans les catalogues des formations agréées qu'il tient à jour. Il informe l'employé du contenu de la formation et lui soumet la liste des prochaines sessions prévues. Lorsque l'employé à fait son choix il inscrit l'employé à la session retenue auprès de l'organisme de formation concerné. En cas d'empêchement l'employé doit avertir au plus vite le responsable formation pour que celui-ci demande l'annulation de l'inscription. A la fin de la formation, l'employé transmet une appréciation sur le stage suivi et un document attestant sa présence. Le responsable de formation contrôle la facture envoyée par l'organisme de formation.

- Tracer le diagramme des cas d'utilisation.

TD4 : UML

Question de cours :

1) Dans quel diagramme on montre les scénarios des cas d'utilisation modélisé par un UCD ?
2) Comment peut-on décrire les différents modificateurs d'attributs ou de méthodes dans un diagramme de classe ?
3) Quelles est la relation entre le diagramme de classe et celui des objets ? Lequel sera construit le premier de préférence ? Justifier.

Exercice 1 :

Etablir un diagramme de classes pour l'énoncé suivant :

- Chaque étudiant du département INF suit un ensemble d'unités d'enseignement (UE).
- Chaque UE a un coefficient et est constituée de cours, de travaux dirigés (TD) et de travaux pratiques (TP).
- Chaque cours, TD ou TP a une date. Les cours sont faits en amphi, les TD en salle de classe et les TP en salle machine.
- Pour les TP et TD, les étudiants sont répartis dans des groupes. Pour chaque TP, chaque étudiant est en binôme avec un autre étudiant.
- Les cours et les TD sont assurés par un enseignant. Les TP sont assurés par deux enseignants.
- Pour chaque UE, l'étudiant a une note de devoir surveillé ; pour chaque TP, le binôme a une note de TP.

Exercice 2 :

Modéliser le fait qu'il y a des voitures bleues, des voitures rouges et des voitures vertes.

1) Proposer deux solutions : la première basée sur l'héritage et la deuxième basée sur la composition.
2) Modéliser en UML les deux solutions possible

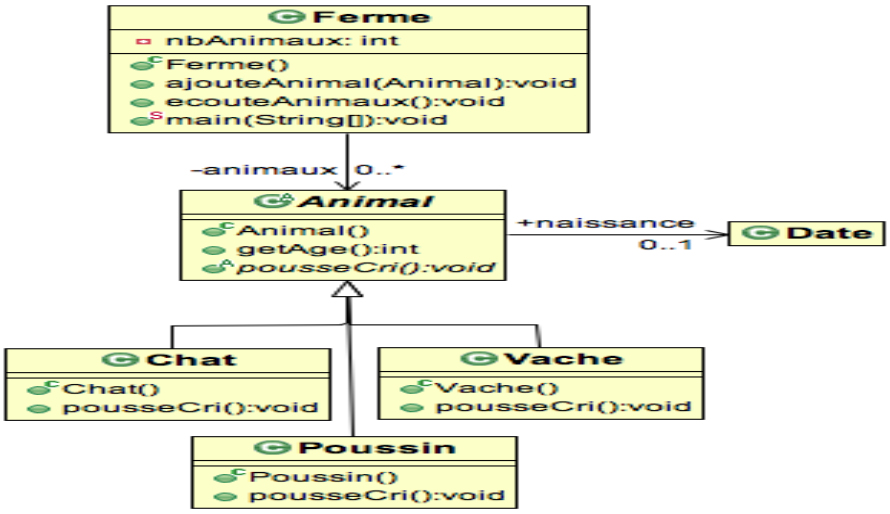
Exercice 3 :

On veut avoir une conception dans laquelle on peut appeler dans une classe B une opération op() d'une classe A ?

1) proposer deux solutions différentes : avec héritage, ou avec association
2) modéliser les deux cas avec UML

Exercice 4 :

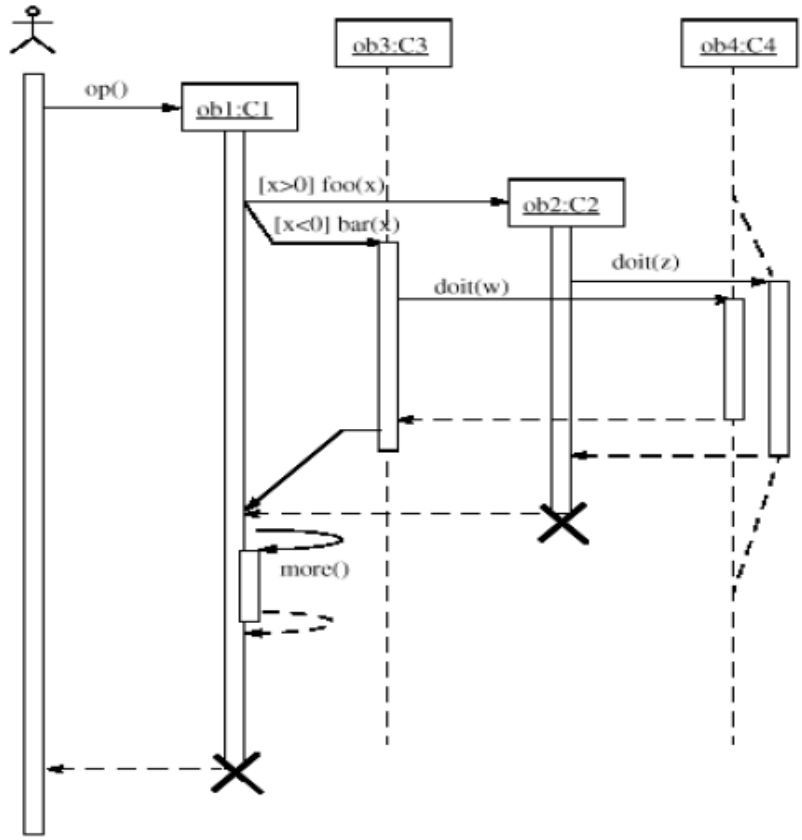
Soit la figure de DC ci-dessous :

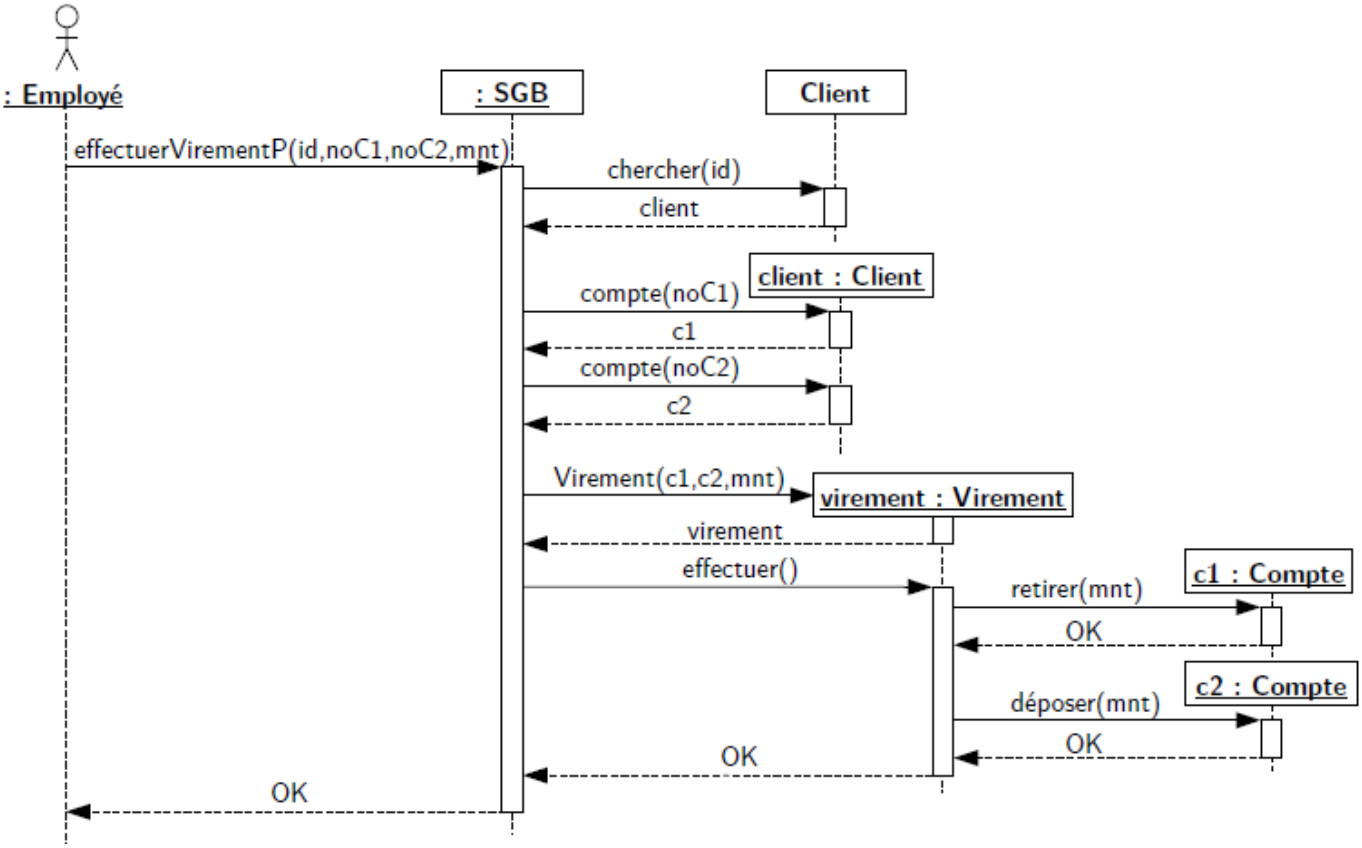


1) identifier les classes abstraites.
2) identifier les différents types d'associations, leurs contraintes aussi.
3) Proposer un code java pour le diagramme de classe de la figure ci-dessus.

Exercice 5 :

Déduire la signification des diagrammes de séquence ci-dessous :


ob3:C3
ob4:C4
op()
ob1:C1
[x>0] foo(x)
ob2:C2
[x<0] bar(x)
doit(z)
doit(w)
more()


: Employé
: SGB
Client
effectuerVirementP(id,noC1,noC2,mnt)
chercher(id)
client
client : Client
compte(noC1)
c1
compte(noC2)
c2
Virement(c1,c2,mnt)
virement : Virement
virement
effectuer()
c1 : Compte
retirer(mnt)
OK
c2 : Compte
déposer(mnt)
OK
OK
OK

Corrigé type (TD4):

Question de cours :

1) Dans quel diagramme on montre les scénarios des cas d'utilisation modélisé par un UCD ?
   Diagramme de séquence
2) Comment peut-on décrire les différents modificateurs d'attributs ou de méthodes dans un diagramme de classe ?
   public + , private -, protected #, static souligné
3) Quelles est la relation entre le diagramme de classe et celui des objets ? Lequel sera construit le premier de préférence ?
   DO est une instance du DC
   Le premier est à créer est le DO

Exo2 :

Solution 1 : Héritage
Créer une classe abstraite Voiture
Créer 3 classes VoitureBleue, VoitureRouge et VoitureVerte qui héritent de Voiture
Solution 2 : Composition
Créer une classe Voiture et une classe Couleur (énumération)
Créer une association entre Voiture et Couleur

exo3

Solution 1 : Héritage
- Faire hériter B de A
- op() peut être appelée depuis n'importe quelle instance de B

Solution 2 : Délégation
- Ajouter une association de B vers A :
  - Ajouter dans B un attribut a de type A
  - a.op() peut être appelée depuis n'importe quelle instance de B

exo4 :

1) les classes abstraites sont identifiées par un petit A juste à côté du nom de la classe

Annexe B : Examens

Examen 2015-2016

Questions de cours :

1. Citer trois objectifs du GL?

(1)……(2)…….....(3)
.....

2. Citer deux problèmes de la cascade?

(1)…….....

.....

(2)…….....

.....

3. Pourquoi UML n'est pas une méthode de développement?

…….....

.....

4. Proposer pour chacun des trois diagrammes UML : UCD, CD, SD la phase où il peut être le mieux exploité ?

…….....

…….....

Citer un diagramme UML utilisé pour décrire le comportement d'un seul objet et un diagramme UML utilisé pour décrire le comportement d'interaction entre objets :

……
….. …….

5. Quelle est la différence entre « extends » et « include » dans un diagramme de cas d'utilisation ?

…….....

Donner deux différence entre un diagramme de classe et un diagramme de séquence.

…….....**.....**

Exercice 1 :

Soir le diagramme de la figure 1.

1. C'est quoi le type de ce diagramme ?
2. Proposer un code Java correspondant aux éléments : Monstre, Marion, et Goomba.

Exercice 1 :

Soit le code source 1

1. Proposer un diagramme de classe pour ce code source.

2. Proposer un diagramme de séquence pour le programme principal.

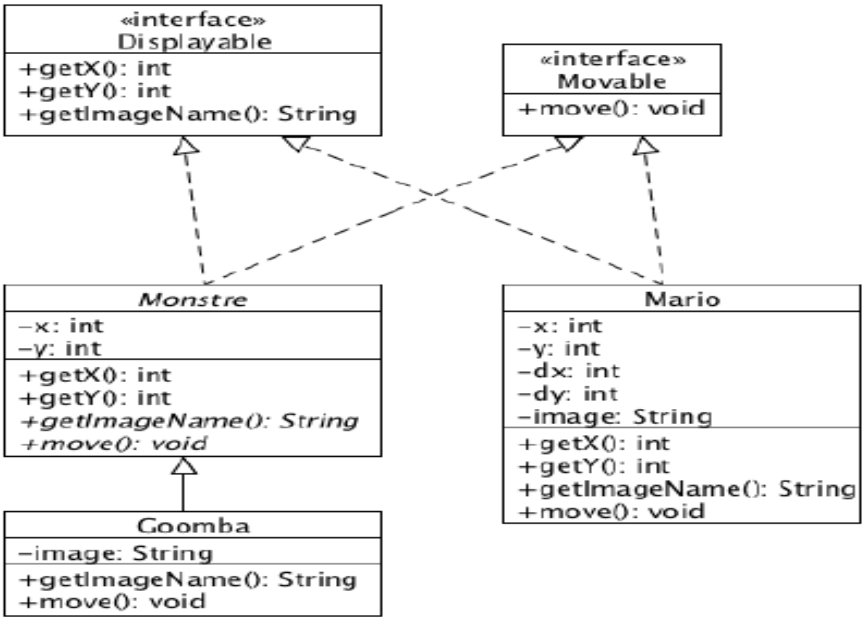


Figure 1

```
class Robot{
privée :
        BrasArticulé brasArticulé ;
publique :
        void chercherPièce() {
                brasArticulé.déplier() ;
                brasArticulé.replier() ;
        }
}
class BrasArticulé {
privée :
        Pince pince ;
publique :
        void déplier() {
                ...
                pince.fermer() ;
        }
        void replier() {
                ...
                pince.ouvrir() ;
        }
}
class Pince {
privée :
        ...
publique :
        void fermer() { ... }
        void ouvrir() { ... }
}
Début programme principal
        Robot robot ;
        robot.chercherPièce() ;
Fin programme principal
```

Examen 2016-2017

Questions de cours (1): cocher uniquement les énoncés corrects. (**Notation:** +0.5 énoncé correct et -0.5 énoncé incorrect)

1) Il existe une seule approche pour développer un logiciel
2) L'approche de la cascade souffre de problème de temps
3) Pour résoudre les problèmes de temps en cascade, on a proposé les transformations formelles
4) L'activité de test ne garantie pas l'absence d'erreurs dans un système
5) Le codage est l'activité la plus importante dans le cycle de développement
6) Le GL est proposé pour résoudre seulement la fiabilité
7) Le diagramme de classe ne doit pas nécessairement expliciter les noms des méthodes
8) Un diagramme de cas d'utilisation est utilisé pour l'analyse et la conception
9) UML est une méthode pour le développement de systèmes orienté objet
10) UML n'exige pas que l'implémentation soit en OO
11) La fiabilité garantie l'absence d'erreurs et le contrôle de temps de livraison
12) L'étude de faisabilité permet d'établir le cahier de définition des besoins
13) Le cahier de définition de besoin décrit les aspects fonctionnels et non fonctionnels du système
14) Un diagramme de classe donne lieu à plusieurs instances de diagramme d'objets
15) La programmation exploratoire peut être utilisée pour tout genre de système
16) Les diagrammes de séquence décrivent les classes et leurs relations
17) Les interactions entre objet sont données par le diagramme de séquence
18) UML est un langage de programmation orienté objet
19) Pour assurer la fiabilité, il faut adopter l'approche de prototypage
20) Un diagramme d'objet ne doit pas comporter des méthodes de classe

Questions de cours (2):

1) Donner un objectif de l'usage des transformations formelles:
   .....
2) Proposer une approche qui réduit le temps de développement:
   .....
3) Donner un objectif de l'usage de la programmation exploratoire:
   .....
4) Donne un objectif de l'usage des diagrammes de séquence:
   .....
5) Donner les deux phases où les diagrammes UC sont utilisés:
   .....

Exercice 1: établir un DCU pour la description suivante (Montrer les différentes relations possibles entre UC et entre acteurs si elles existent)

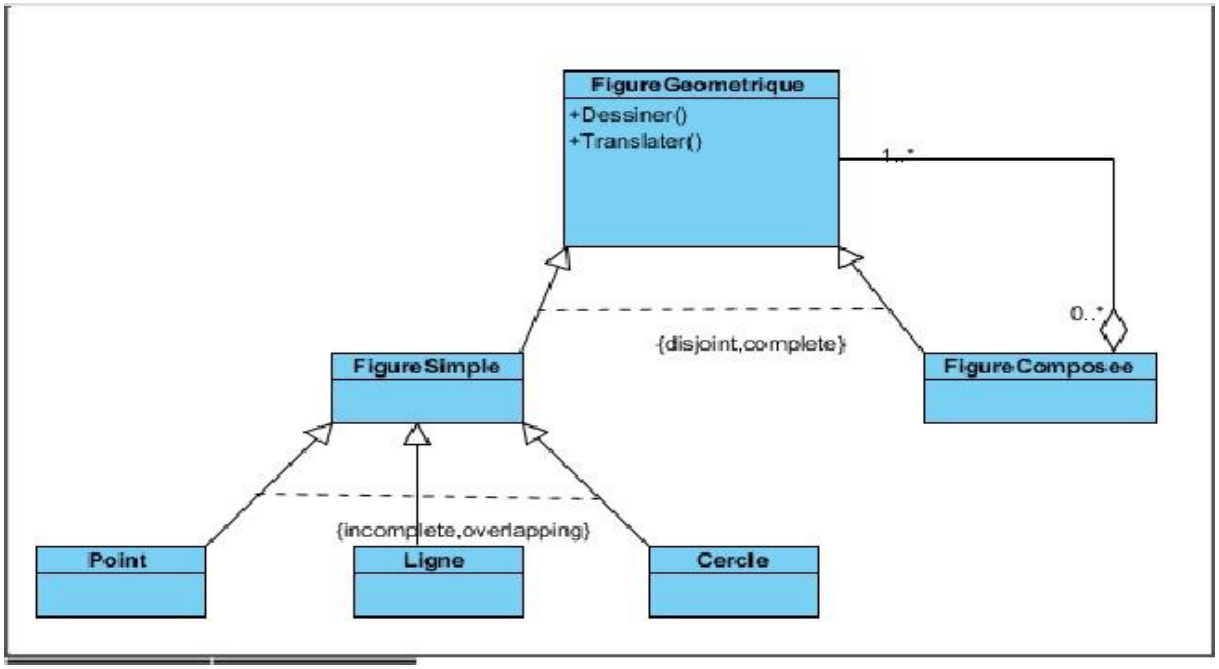
Dans une entreprise de construction de pièce, il existe plusieurs usines. Cette usine reçoit des commandes de ses clients. Une fois, le commande reçu, un agent de service commercial traitent ces commandes et aussi établit des factures de ces commandes. La fabrication des pièces est effectuée par une usine générale, et leur finalisation est effectuée par une usine particulière dit de montage. Enfin, en cas de besoin de matière brut un agent du service de stockage de l'entreprise assure l'approvisionnement des matières brutes. Cette tâche exige le choix d'un fournisseur et l'établissement de facture si le fournisseur l'exige.

Exercice 2: établir un DC pour la description suivante

Une figure géométrique est composée de figures simples ou composées. Une figure composée est constituée de plusieurs figures et une figure simple peut être un point, une ligne ou un cercle. Une figure peut être dessinée ou translatée.

Elément du corrigé de Examen 2016-2017

Exercice 2 :



Examen 2017-2018

Questions de cours (1) :

1) Que signifie une forte cohésion ?
   ……
   ……..
2) Que signifie un faible couplage ?
   ……
   …….
3) Quel type de relation existe entre le couplage et la cohésion ?
   ……
4) Une bonne conception sera caractérisée par quoi donc ?
   …….
5) C'est l'approche qui a pour objectif de garantir l'absence d'erreurs ?
   …….

Questions de cours (2): Mettez un X devant **uniquement** les énoncés correctes

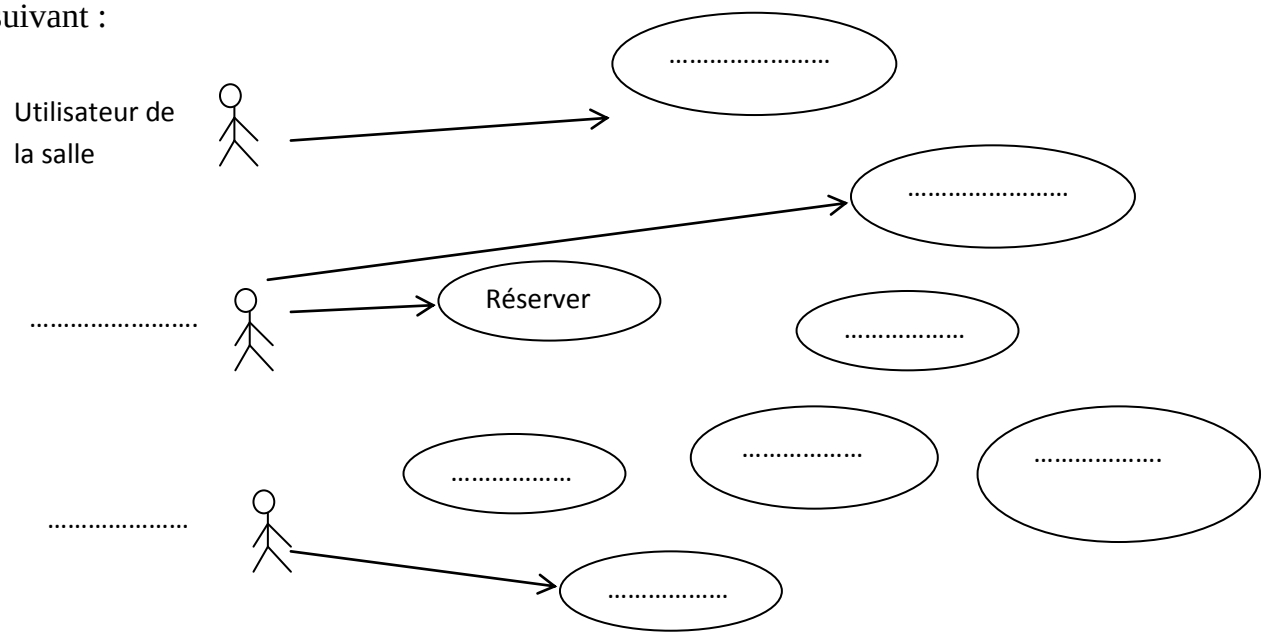
1) Un diagramme de séquence représente les interactions entre classes. Il permet de décrire plusieurs types de messages échangés entre ces classes.
2) Pour augmenter la fiabilité et garantir l'absence d'erreur, on fait appelle à des techniques utilisant la preuve de correction, c'est le cas de l'approche de prototypage par exemple.
3) La fiabilité est la qualité la plus exigée dans le logiciel. C'est une qualité qui ne peut jamais être garantie à 100%.
4) L'usage des transformations formelles rencontre plusieurs difficultés, surtout que cette approche n'a pas pour but l'absence d'erreurs.
5) Les activités destructrices seront guidées par celles utilisées dans les premières phases. Ces activités destructrices permettent l'analyse, la conception et le test du logiciel.
6) Il est toujours possible d'avoir plusieurs diagrammes OO à partir d'uns seul diagramme de classe.

Exercice 1 :

On s'intéresse à gérer la réservation de salles de cours et du matériel pédagogiques (portable, data-show). Seuls les enseignants peuvent faire la réservation (si la salle est disponible ou si le matériel est disponible). Les enseignants ainsi que les étudiants (les utilisateurs de salles) peuvent consulter les emplois du temps. Les enseignants sont les

seuls capables de consulter le récapitulatif horaire. Ce récapitulatif est réalisé par un enseignant responsable de la formation.

Proposer un diagramme de cas d'utilisation, en se limitant à compléter le schéma suivant :



Exercice 2 :

Un réseau de machines est composé de 4 machines clientes : C1, C2, C3, C4 (demandent des services) et de 2 machines serveurs : S1, S2. Ces machines se communiquent via un protocole (ensemble de règles définissant l'ordre des messages échangés entre machines). Tout d'abord, l'utilisateur lance les 6 machines. Après leur lancement, les clients commencent à envoyer un message d'authentification vers S1. Ces envois sont synchrones et se font en parallèle. Une fois, toutes ces machines authentifiées dans S1, les deux clients C1 et C2 envoient de ma même manière des messages d'authentification à S2. Après cette deuxième authentification, C3 et C2 ont le choix d'envoyer un premier message synchrone soit à C4 ou à S1. Si C4 a reçu un message de C2 alors C4 envoie n messages successif vers C1. Après que C4 envoie tous ces messages il arrête son exécution.

- Proposer un diagramme de séquence pour ce scénario.
- Proposer un diagramme de classe pour ce système

Bibliographie

Bibliographie :

1. Marie-Claude Gaudel, Bruno Marre, Françoise Schlienger,, Gilles Bernot (Auteur), « Précis de génie logiciel». Editeur : Dunod (1 mai 1996). ISBN-10: 2225851891. ISBN-13: 978-2225851896

2. Ian Sommerville . « Le Génie logiciel et ses applications ». Broché – 28 février 1988. ISBN-10: 2729601805. ISBN-13: 978-2729601805. Editeur : Dunod.

3. Grady Booch, James Rumbaugh, Ivar Jacobson, « UML 2.0: guide de référence ». Date de publication originale : 2004. CampusPress, 2004.

4. Le site du groupe OMG pour avoir toutes les informations officielles sur le langage UML : https://www.omg.org/