Prof. Dr. MUSANGU LUKA

Professeur des Universités

NOTES DE COURS DE CONCEPTION DE SYSTEMES D'INFORMATION : PROCESSUS UNIFIE

Edition EMMANUEL - 2016

TABLE DES MATIERES

PARTIE 1 : CADRAGE CONCEPTUEL SUR LA MODELISATION OBJET

CHAPITRE I : INTRODUCTION A LA MODELISATION OBJET

1. INTRODUCTION

L'approche objet est incontournable dans le cadre du développement de systèmes logiciels complexes, capables de suivre les évolutions incessantes des technologies et des besoins applicatifs. Cependant, la programmation objet est moins intuitive que la programmation fonctionnelle. En effet, il est plus naturel de décomposer les problèmes informatiques en termes de fonctions qu'en termes d'ensembles d'objets en interaction. De ce fait, l'approche objet requiert de modéliser avant de concevoir. La modélisation apporte une grande rigueur, offre une meilleure compréhension des logiciels, et facilite la comparaison des solutions de conception avant leur développement. Cette démarche se fonde sur des langages de modélisation, qui permettent de s'affranchir des contraintes des langages d'implémentation.

Le besoin d'une méthode de description et de développement de systèmes, prenant en compte à la fois les données et les traitements, a grandi en même temps que la taille des applications objet. Au milieu des années 90, plusieurs dizaines de méthodes objet sont disponibles, mais aucune ne prédomine. L'unification et la normalisation des trois méthodes dominantes, à savoir Booch, du nom de son auteur, OOSE (*Object Oriented Software Engineering*), d'Ivan Jacobson et OMT (*Object Modeling Technique*), de James Rumbaugh, sont à l'origine de la création du langage UML (*Unified Modeling Language*).

1. HISTORIQUE DU LANGAGE UML

À la fin des années 80, l'industrie commence à utiliser massivement les langages de programmation orientés objet, tels que C++, Objective C, Eiffel et Smalltalk. De l'industrialisation de ce type de programmation est né le besoin de « penser objet », indépendamment du langage d'implémentation. Plusieurs équipes proposent alors des méthodes (OMT, OOSE, Booch, Coad, Odell, CASE...) qui, pour la plupart, modélisent les mêmes concepts fondamentaux dans différents langages, avec une terminologie, des notations et des définitions différentes. Les différents protagonistes conviennent rapidement du besoin d'unifier ces langages en un standard unique.

Lors de la conférence OOPSLA d'octobre 1995, Booch et Rumbaugh présentent la version 0.8 de leur méthode unifiée (Unified Method 0.8). Ils sont rejoints la même année par Jacobson. Les trois auteurs améliorent la méthode unifiée et proposent en 1996 la version 0.9 du langage UML. Rational Software, qui emploie désormais le trio, publie en 1997 la documentation de

la version 1.0 d'UML et la propose à l'OMG en vue d'une standardisation. Des modifications sont apportées à la version proposée par Rational, puis l'OMG (Object Management Group) propose, la même année, la version UML 1.1, qui devient un standard.

L'OMG constitue ensuite un groupe de révision nommé RTF (Revision Task Force). Entre- temps, de très nombreux utilisateurs industriels adoptent UML et apportent quelques modifications, ce qui conduit à la proposition de la version 1.2 en 1999. La première révision significative du langage est la version 1.3, proposée en 1999, dont la spécification complète est publiée en mars 2000. En mars 2003, la version 1.5 voit le jour.

Concrètement, UML 1 est utilisé lors de la phase d'analyse et de conception préliminaire des systèmes. Il sert à spécifier les fonctionnalités attendues du système (diagrammes de cas d'utilisation et de séquence) et à décrire l'architecture (diagramme de classes). La description de la partie comportementale (diagrammes d'activités et d'états) est moins utilisée. Cela est dû essentiellement à l'insuffisance de la formalisation de la conception détaillée dans UML 1. La plupart du temps, en matière de spécification des algorithmes et des méthodes de traitement, le seul moyen est de donner une description textuelle informelle.

Certes, des outils comme les automates et les diagrammes d'activités sont disponibles, mais leur emploi est limité. Les utilisateurs restent sur le vieux paradigme centré sur le code : ils se contentent de recourir à UML lors des phases préliminaires et passent à un langage de programmation classique lors des phases de codage et de tests. L'un des objectifs de l'OMG est de proposer un paradigme guidé par des modèles décrivant à la fois le codage, la gestion de la qualité, les tests et vérifications, et la production de la documentation. Il s'agit de recentrer l'activité des informaticiens sur les fonctions que le système doit fournir, en conformité avec les exigences du client et les standards en vigueur.

2. POURQUOI MODELISER ?

Un modèle est une représentation simplifiée d'une réalité. Il permet de capturer des aspects pertinents pour répondre à un objectif défini a priori. Par exemple, un astronaute modélisera la Lune comme un corps céleste ayant une certaine masse et se trouvant à une certaine distance de la Terre, alors qu'un poète la modélisera comme une dame avec laquelle il peut avoir une conversation.

Le modèle s'exprime sous une forme simple et pratique pour le travail. Quand le modèle devient compliqué, il est souhaitable de le décomposer en plusieurs modèles simples et manipulables.

L'expression d'un modèle se fait dans un langage compatible avec le système modélisé et les objectifs attendus. Ainsi, le physicien qui modélise la lune utilisera les mathématiques comme langage de modélisation. Dans le cas du logiciel, l'un des langages utilisés pour la modélisation est le langage UML. Il possède une sémantique propre et une syntaxe composée de graphique et de texte et peut prendre plusieurs formes (diagrammes).

Les modèles ont différents usages :

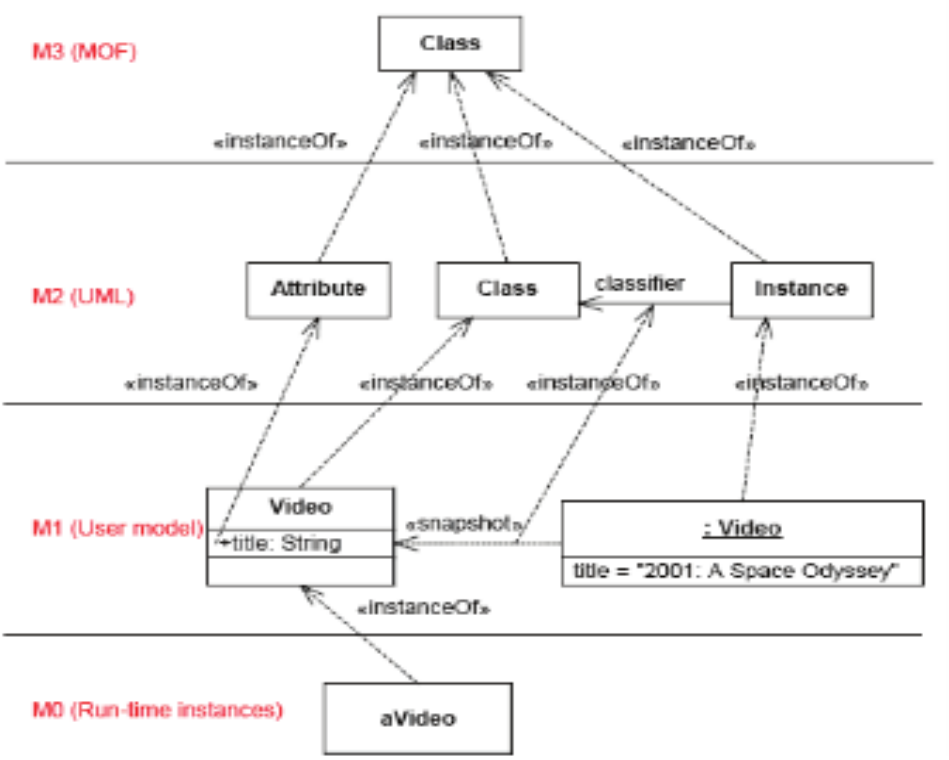
- Ils servent à circonscrire des systèmes complexes pour les dominer. Par exemple, il est inimaginable de construire une fusée sans passer par une modélisation permettant de tester les réacteurs, les procédures de sécurité, l'étanchéité de l'ensemble, etc.
- Ils optimisent l'organisation des systèmes. La modélisation de la structure d'une entreprise en divisions, départements, services, etc. permet d'avoir une vision simplifiée du système et par là même d'en assurer une meilleure gestion
- Ils permettent de se focaliser sur des aspects spécifiques d'un système sans s'embarrasser des données non pertinentes. Si l'on s'intéresse à la structure d'un système afin de factoriser ses composants, il est inutile de s'encombrer de ses aspects dynamiques. En utilisant, par exemple, le langage UML, on s'intéressera à la description statique (via le diagramme de classes) sans se soucier des autres vues.
- Ils permettent de décrire avec précision et complétude les besoins sans forcément connaître les détails du système.
- Ils facilitent la conception d'un système, avec notamment la réalisation de maquette approximative, à échelle réduite, etc.
- Ils permettent de tester une multitude de solutions à moindre coût et
- dans des délais réduits et de sélectionner celle qui résout les problèmes posés.

La modélisation objet produit des modèles discrets permettant de regrouper un ensemble de configurations possibles du système et pouvant être implémentés dans un langage de programmation objet. La modélisation objet présente de nombreux avantages à travers un ensemble de propriétés (classe, encapsulation, héritage et abstraction, paquetage, modularité, extensibilité, adaptabilité, réutilisation) qui lui confère toute sa puissance et son intérêt.

UML 2 apporte des évolutions majeures par rapport à UML 1.5, sans toutefois être révolutionnaire : les principales fonctionnalités de base se ressemblent. Au niveau du méta-modelé, qui concerne surtout les développements d'outils, UML 2 se rapproche davantage des standards de modélisation objet proposés par l'OMG. En particulier, l'unification du noyau UML et des parties conceptuelles de modélisation MOF (Meta-Object Facility) permet aux outils MOF de gérer les modèles UML ; l'ajout du principe de profils permet de mieux définir les extensions du domaine ; enfin, la réorganisation du méta modèle UML élimine les redondances présentes dans les versions précédentes.

Du point de vue de l'utilisateur, les changements concernent certaines notations. La notation des diagrammes de séquence se rapproche de la norme d'échanges de messages MSC (Message Sequence Chart) de l'IUT (Union Internationale des Télécommunications). Le concept de classeurs s'inspire des avancées de l'ingénierie temps réel du langage de description et de spécification SDL. De plus, UML 2 unifie la modélisation des activités et la modélisation des actions introduites dans UML 1.5 et utilise des notations de modélisation de processus métier. Des éléments de modélisation contextuels améliorent la souplesse et formalisent mieux le concept d'encapsulation des classes et des collaborations.

Afin de formaliser le modèle utilisateur du langage et de le rapprocher davantage des normes OMG, le langage UML est structuré en quatre couches (figure 0.1) ; seules les deux couches user model et run time instance sont destinées à l'utilisateur.



L'utilisateur n'a pas besoin de mettre en évidence cette organisation quand il utilise UML. Il doit se contenter de respecter la syntaxe et la sémantique du langage détaillées dans ce cours. Il doit également connaître les différents diagrammes mettant en valeur tantôt des aspects statiques, tantôt des aspects comportementaux du système. Une organisation conceptuelle des différents diagrammes UML permet d'avoir une vision plus claire des vues offertes par ce langage. Les trois auteurs à l'origine du langage UML proposent un découpage conceptuel en quatre domaines : structurel, dynamique, physique et gestion de modèles.

4. CONCEPTS IMPORTANTS DE L'APPROCHE OBJET

Il est important de savoir que l'approche objet rapproche les données et leurs traitements.

Mais cette approche ne fait pas que ça, d'autres concepts importants sont spécifiques à cette approche et participent à la qualité du logiciel.

4.1 Notion de classe

Une classe est un type de données abstrait, caractérisé par des propriétés (attributs et méthodes) communes à toute une famille d'objets et permettant de créer (instancier) des objets possédant ces propriétés. Les autres concepts

importants qu'il nous faut maintenant introduire sont l'encapsulation, l'héritage et l'agrégation.

4.2 Encapsulation

L'encapsulation consiste à masquer les détails d'implémentation d'un objet, en définissant une interface. L'interface est la vue externe d'un objet, elle définit les services accessibles (offerts) aux utilisateurs de l'objet. L'encapsulation facilite l'évolution d'une application car elle stabilise l'utilisation des objets : on peut modifier l'implémentation des attributs d'un objet sans modifier son interface, et donc la façon dont l'objet est utilisé.

L'encapsulation garantit l'intégrité des données, car elle permet d'interdire, ou de restreindre, l'accès direct aux attributs des objets.

4.3 Héritage, Spécialisation, Généralisation et polymorphisme

L'héritage est un mécanisme de transmission des propriétés d'une classe (ses attributs et méthodes) vers une sous-classe. Une classe peut être spécialisée en d'autres classes, afin d'y ajouter des caractéristiques spécifiques ou d'en adapter certaines. Plusieurs classes peuvent être généralisées en une classe qui les factorise, afin de regrouper les caractéristiques communes d'un ensemble de classes.
Ainsi, la spécialisation et la généralisation permettent de construire des hiérarchies de classes. L'héritage peut être simple ou multiple. L'héritage évite la duplication et encourage la réutilisation.
Le polymorphisme représente la faculté d'une méthode à pouvoir s'appliquer à des objets de classes différentes. Le polymorphisme augmente la généricité, et donc la qualité, du code.

4.4 Agrégation

Il s'agit d'une relation entre deux classes, spécifiant que les objets d'une classe sont des composants de l'autre classe. Une relation d'agrégation permet donc de définir des objets composés d'autres objets.
L'agrégation permet donc d'assembler des objets de base, afin de construire des objets plus complexes.

5. LANGAGE DE MEDELISATION UNIFIE (UML)

UML est une notation graphique conçue pour représenter, spécifier, construire et documenter les systèmes logiciels. Ses deux principaux objectifs sont la modélisation de systèmes utilisant les techniques orientées objet, depuis la conception jusqu'à la maintenance, et la création d'un langage abstrait compréhensible par l'homme et interprétable par les machines. UML s'adresse à toutes les personnes chargées de la production, du déploiement et du suivi de logiciels (analystes, développeurs, chefs de projets, architectes...), mais peut également servir à la communication avec les clients et les utilisateurs du logiciel. Il s'adapte à tous les domaines d'application et à tous les supports. Il permet de construire plusieurs modèles d'un système, chacun mettant en valeur des aspects différents : fonctionnels, statiques, dynamiques et organisationnels. UML est devenu un langage incontournable dans les projets de développement.

Une méthode de développement définit à la fois un langage de modélisation et la marche à suivre lors de la conception. Le langage UML propose uniquement une notation dont l'interprétation est définie par un standard, mais pas une méthodologie complète. Plusieurs processus de développement complets fondés sur UML existent, comme le Rational Unified Process (RUP), de Booch, Jacobson et Rumbaugh, ou l'approche MDA (Model Driven Architecture) proposée par l'OMG, mais ils ne font pas partie du standard UML.

Le processus RUP propose de bien maîtriser les étapes successives de la modélisation et du développement par une approche itérative. L'approche MDA propose une architecture du développement en deux couches : le PIM (Platform Indepent Model) représente une vision abstraite du système, indépendante de l'implémentation ; le PSM (Platform Specific Model) représente les programmes exécutables, qui doivent être obtenus en partie automatiquement à partir du PIM ; à cela s'ajoute le PDM (Platform Definition Model), en l'occurrence une description de l'architecture physique voulue (langages de programmation, architecture matérielle...).

UML intègre de nombreux concepts permettant la génération de programmes. C'est un langage de modélisation fondé sur des événements ou des messages. Il ne convient pas pour la modélisation de processus continus, comme la plupart des procédés en physique. Ce n'est pas un langage formel, ni un langage de programmation. Il ne peut pas être utilisé pour valider un système, ni pour générer un programme exécutable complet. Mais, il permet de produire des parties de code, comme le squelette des classes (attributs et

signatures de méthode). Même si la version 2 apporte des avancées significatives au niveau du formalisme, UML n'a pas encore atteint la rigueur syntaxique et sémantique des langages de programmation.

UML est le résultat d'un large consensus, continuellement enrichi par les avancées en matière de modélisation de systèmes et de développement de logiciels. C'est le résultat d'un travail d'experts reconnus, issus du terrain. Il couvre toutes les phases du cycle de vie de développement d'un système et se veut indépendant des langages d'implémentation et des domaines d'application.

UML dans sa version 2 propose treize diagrammes qui peuvent être utilisés dans la description d'un système. Ces diagrammes sont regroupés dans deux grands ensembles.

• **Les diagrammes structurels** – Ces diagrammes, au nombre de six, ont vocation à représenter l'aspect statique d'un système (classes, objets, composants...).

- ***Diagramme de classe*** – Ce diagramme représente la description statique du système en intégrant dans chaque classe la partie dédiée aux données et celle consacrée aux traitements. C'est le diagramme pivot de l'ensemble de la modélisation d'un système.
- ***Diagramme d'objet*** – Le diagramme d'objet permet la représentation d'instances des classes et des liens entre instances.
- ***Diagramme de composant*** (modifié dans UML 2) – Ce diagramme représente les différents constituants du logiciel au niveau de l'implémentation d'un système.
- ***Diagramme de déploiement*** (modifié dans UML 2) – Ce diagramme décrit l'architecture technique d'un système avec une vue centrée sur la répartition des composants dans la configuration d'exploitation.
- ***Diagramme de paquetage*** (nouveau dans UML 2) – Ce diagramme donne une vue d'ensemble du système structuré en paquetage. Chaque paquetage représente un ensemble homogène d'éléments du système (classes, composants...).
- ***Diagramme de structure composite*** (nouveau dans UML 2) – Ce diagramme permet de décrire la structure interne d'un ensemble complexe composé par exemple de classes ou d'objets et de composants techniques. Ce diagramme met aussi l'accent sur les liens entre les sous-ensembles qui collaborent.

▪ **Les diagrammes de comportement** – Ces diagrammes représentent la partie dynamique d'un système réagissant aux événements et permettant

de produire les résultats attendus par les utilisateurs. Sept diagrammes sont proposés par UML :

- *Diagramme des cas d'utilisation* – Ce diagramme est destiné à représenter les besoins des utilisateurs par rapport au système. Il constitue un des diagrammes les plus structurants dans l'analyse d'un système.
- *Diagramme d'état-transition* (machine d'état) – Ce diagramme montre les différents états des objets en réaction aux événements.
- *Diagramme d'activités* (modifié dans UML 2) – Ce diagramme donne une vision des enchaînements des activités propres à une opération ou à un cas d'utilisation. Il permet aussi de représenter les flots de contrôle et les flots de données.
- *Diagramme de séquence* (modifié dans UML 2) – Ce diagramme permet de décrire les scénarios de chaque cas d'utilisation en mettant l'accent sur la chronologie des opérations en interaction avec les objets.
- *Diagramme de communication* (anciennement appelé collaboration) – Ce diagramme est une autre représentation des scénarios des cas d'utilisation qui met plus l'accent sur les objets et les messages échangés.
- *Diagramme global d'interaction* (nouveau dans UML 2) – Ce diagramme fournit une vue générale des interactions décrites dans le diagramme de séquence et des flots de contrôle décrits dans le diagramme d'activités.
- *Diagramme de temps* (nouveau dans UML 2) – Ce diagramme permet de représenter les états et les interactions d'objets dans un contexte où le temps a une forte influence sur le comportement du système à gérer.

Aujourd'hui UML 2 décrit les concepts et le formalisme de ces treize diagrammes mais ne propose pas de démarche de construction couvrant l'analyse et la conception d'un système. Ce qui a pour conséquence par exemple de ne pas disposer d'une vision des interactions entre les diagrammes.

5.1 Différentes vues et diagrammes D'UML

Toutes les vues proposées par UML sont complémentaires les unes des autres, elles permettent de mettre en évidence différents aspects d'un logiciel à réaliser. On peut organiser une présentation d'UML autour d'un découpage en vues, ou bien en différents diagrammes, selon qu'on sépare plutôt les aspects fonctionnels des aspects architecturaux, ou les aspects statiques des aspects dynamiques. Nous adopterons plutôt dans la suite un découpage en diagrammes, mais nous commençons par présenter les différentes vues, qui sont les suivantes :

- *La vue fonctionnelle, interactive*, qui est représentée à l'aide de diagrammes de cas et de diagrammes des séquences. Elles cherchent à appréhender les interactions entre les différents acteurs/utilisateurs et le système, sous forme d'objectif à atteindre d'un côté et sous forme chronologique de scénarios d'interaction typiques de l'autre.
- *La vue structurelle ou statique,* réunit les diagrammes de classes et les diagrammes de packages. Les premiers favorisent la structuration des données et tentent d'identifier les objets/composants constituant le programme, leurs attributs, opérations et méthodes, ainsi que les liens ou associations qui les unissent. Les seconds s'attachent à regrouper les classes fortement liées entre elles en des composants les plus autonomes possibles. A l'intérieur de chaque package, on trouve un diagramme de classes.
- *La vue dynamique*, qui est exprimée par les diagrammes d'états. Cette vue est plus algorithmique et orientée « traitement », elle vise à décrire l'évolution (la dynamique) des objets complexes du programme tout au long de leur cycle de vie. De leur naissance à leur mort, les objets voient leurs changements d'états guidés par les interactions avec les autres objets. Le diagramme d'activité est une sorte d'organigramme correspondant à une version simplifiée du diagramme d'états. Il permet de modéliser des activités qui se déroulent en parallèle les unes des autres, quand ce parallélisme peut poser problème. En général, les diagrammes d'états à eux seuls ne permettent pas de faire apparaître les problèmes spécifiques posés par la synchronisation des processus en concurrence, pour assurer la cohérence du comportement et l'absence d'inter-blocage. Etablir un diagramme d'activité peut aider à mettre au point un diagramme d'états.

CHAPITRE II : LES METHODES AGILES

Une méthode agile est une méthode de développement informatique permettant de concevoir des logiciels en impliquant au maximum l'utilisateur, avec une grande réactivité à ses demandes. Ces méthodes se veulent plus pragmatiques que leurs homologues classiques et commencent à être appliquées dans tous environnements.

Pour une entreprise ou une organisation, l'agilité peut être définie comme l'aptitude à faire évoluer de façon continue ses structures et ses ressources ; en ce sens, l'agilité est une préoccupation aussi ancienne que l'entreprise elle-même. Toutefois, cet objectif stratégique a surtout été poursuivi par le moyen de la qualité, qui vise à améliorer les processus pour innover ou améliorer ses produits et services.

Dans le domaine de l'informatique, l'approche par les processus est rapidement devenue synonyme de planification lourde du développement ; or l'accélération de la course à la rentabilité et la banalisation progressive des technologies de l'information créent, aujourd'hui, un contexte peu propice à ces méthodes traditionnelles.

C'est ainsi qu'est apparu à la fin des années quatre-vingt-dix, un mouvement de remise en cause des méthodes informatiques traditionnelles, perçues comme "bureaucratiques", soutenu par un management de plus en plus pressé.

1. LES NOUVELLES METHODES AGILES

Ainsi les pionniers du mouvement Agile Programming – principalement aux États-Unis – ont développé, de façon pragmatique, des nouvelles méthodes, agiles ; citons notamment :

- XP - 1999 (Kent Beck, Ward Cunningham, Ron Jeffries), qui s'appuie sur la remise en cause permanente – refactoring – d'un système à chaque itération ; les itérations durent entre une et trois semaines ; XP requiert beaucoup de discipline dans son exécution ;
- SCRUM - 2002 (Ken Schwaber, Mike Beedle), issu du développement orienté objet qui se concentre sur le processus itératif de suivi et de planification, avec des itérations d'un mois ;
- La famille Crystal/ASD (Alistair Cockburn, Jim Highsmith), qui veut être la moins disciplinée possible ; l'accent est mis sur la revue à la fin de chaque itération, encourageant ainsi le processus à évoluer de lui-même.

- Le processus unifié est une autre variante des méthodes agiles utilisant les diagrammes UML tout étant guidé par les cas d'utilisation.

Une synthèse et la rédaction des grands principes de l'agilité ont été élaborées en commun par ces ténors du développement agile à l'issu du séminaire de Snowbird (Utah, février 2001) sous l'intitulé de "Manifeste Agile".

Pour le Manifeste agile de logiciels

2. LES APPORTS DE L'AGILITE EN INFORMATIQUE

Trop souvent, les méthodes agiles sont définies en réaction aux méthodes traditionnelles ; cette définition de nature "défensive", cache leurs apports réels, notamment :

- L'utilisation d'un processus adaptatif, à la fois itératif et incrémental ;
- Le contact rapproché avec l'expertise métier, par le travail en binôme (métier, informaticien) et la présence simultanée des membres d'équipes de développeurs sur le plateau, pour favoriser la communication et l'interaction, deux notions érigées en valeurs essentielles des méthodes agiles ;
- L'explicitation des exigences en continu et la présence active de l'utilisateur, qui minimise les risques liés à la gestion des évolutions ;
- L'orientation vers l'humain avec un modèle de gestion avec délégation, où les acteurs décident eux-mêmes comment travailler ;

- La remise au premier plan de la qualité technique, par le travail permanent du code, permet d'obtenir de façon éprouvée, un logiciel maintenable ;
- L'accent mis sur les tests : on écrit les cas de test en même temps (ou avant) d'écrire le code (c'est-à-dire l'ensemble des instructions exécutables par la machine).

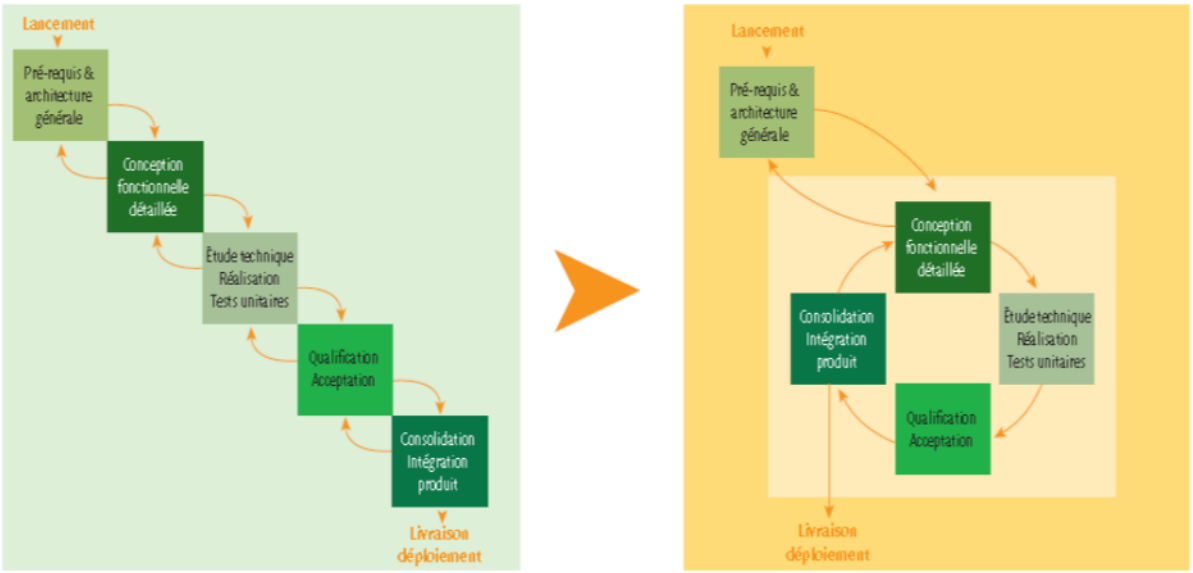
Un autre aspect important de l'agilité est le processus lui-même, qui change au fur et à mesure que l'équipe découvre ce qui fonctionne pour elle et modifie le processus en conséquence : de fait dans ce type de démarche, l'accent est mis sur la prise en charge du changement qui reste bienvenu, même tardivement, dans le processus de développement.

Pour un établissement qui ne souhaiterait pas adopter une méthode agile toute faite, il y a une troisième voie : conjuguer les apports de l'approche par les processus et des méthodes agiles pour rendre plus souples les processus logiciel existants.

3. CONJUGUER L'APPROCHE PAR LES PROCESSUS ET L'AGILITE

Pour rendre agile un processus basé sur la planification des tâches, il faut repenser son ordonnancement pour produire dans un cycle rapide (l'itération) par petit paquet (l'incrément) des unités d'œuvre du produit logiciel. Le but est de livrer une version du logiciel "qui tourne", mais n'ayant qu'une partie des fonctionnalités demandées. La figure ci-dessous montre la transformation d'un processus traditionnel en cascade, en un processus adaptatif.

Rendre agile un processus traditionnel (en cascade)



Un cas d'application des méthodes agiles dans les banques se fait jour : le développement d'applications bancaires à vocation internationale. En effet, avec la mondialisation de l'économie, apparaissent des projets informatiques visant à développer des systèmes dédiés à une ligne d'activité à l'international (les moyens de paiements, les crédits...) pour toutes les filiales et succursales mondiales d'un même établissement.

De tels projets posent des problèmes difficiles à traiter de façon planifiée :

- La mondialisation impose la création de systèmes multidimensionnels (pays, devises, langues, sociétés...) ;
- Les besoins sont particulièrement flous, car une grande partie de l'activité bancaire reste dépendante des réglementations nationales, peu connues ;
- Le niveau d'automatisation est variable, compte tenu des volumes envisagés ;
- L'accessibilité de partout, sans installation lourde, impose des technologies Internet, qui évoluent de façon rapide.

Le projet bénéficie alors du retour d'interaction rapide (le feed-back) entre informaticiens et utilisateurs, facteur clé de succès des méthodes agiles ; cela permet de démarrer vite, au gré des opportunités d'affaire dans les pays et d'améliorer au fur et à mesure le produit et le processus du projet.

4. AVANTAGES ET LIMITES

Les méthodes agiles sont aujourd'hui une réalité dans l'informatique, en particulier lorsque les besoins sont peu précis au départ, mais que les utilisateurs sont prêts à s'investir tout au long du projet. Elles sont bien adaptées si le projet est suffisamment limité pour tenir sur un seul site physique et lorsque les contraintes de fiabilité et de sécurité sont modérées.

Ces méthodes ont remis en lumière l'importance du contact permanent entre fonctionnels, concepteurs, développeurs et testeurs, afin d'informer tous les acteurs des changements et de créer la relation de confiance essentielle pour entretenir l'esprit d'équipe.

Toutefois, l'importance du facteur humain montre que les vraies limites sont l'agilité des esprits, ce qui exclut leur application à des équipes importantes.

CHAPITRE III : LE PROCESSUS UNIFIE

1. PRESENTATION D'UP

UML n'est qu'un langage de modélisation. Nous n'avons pas aujourd'hui la norme, de démarche unifiée pour construire les modèles et conduire un projet mettant en œuvre UML. Cependant les auteurs d'UML, ont décrit, dans un ouvrage [Jacobson2000a] le processus unifié (UP, Unified Process) qui doit être associé à UML. Nous n'allons pas, dans le cadre de ce chapitre, donner une présentation détaillée d'UP. Cependant il nous a paru intéressant de dégager les idées fondatrices d'UP dans le cadre d'une présentation générale. Nous allons tout d'abord expliciter les principes de la méthode UP. Nous compléterons ensuite cette présentation générale en décrivant l'architecture à deux dimensions d'UP et ses principaux concepts, nous passerons aussi en revue les différentes phases d'UP, et pour finir nous détaillerons les activités d'UP.

2. LES PRINCIPES D'UP

Le processus de développement UP, associé à UML, met en œuvre les principes suivants :

- Processus guidé par les cas d'utilisation,
- Processus itératif et incrémental,
- Processus centré sur l'architecture,
- Processus orienté par la réduction des risques.

Ces principes sont à la base du processus unifié décrit par les auteurs d'UML.

2.1 Processus guidé par les cas d'utilisation

L'orientation forte donnée ici par UP est de montrer que le système à construire se définit d'abord avec les utilisateurs. Les cas d'utilisation permettent d'exprimer les interactions du système avec les utilisateurs, donc de capturer les besoins.

Une seconde orientation est de montrer comment les cas d'utilisation constituent un vecteur structurant pour le développement et les tests du système. Ainsi le développement peut se décomposer par cas

d'utilisation et la réception du logiciel sera elle aussi articulée par cas d'utilisation.

2.2 Processus itératif et incrémental

Ce type de démarche étant relativement connu dans l'approche objet, il paraît naturel qu'UP préconise l'utilisation du principe de développement par itérations successives. Concrètement, la réalisation de maquette et prototype constitue la réponse pratique à ce principe. Le développement progressif, par incrément, est aussi recommandé en s'appuyant sur la décomposition du système en cas d'utilisation.

Les avantages du développement itératif se caractérisent comme suit :

- Les avancées sont évaluées au fur et à mesure des itérations,
- Les premières itérations permettent d'avoir un feed-back des utilisateurs,
- Les tests et l'intégration se font de manière continue,
- Les avancées sont évaluées au fur et à mesure de L'implémentation.

2.3 Processus centré sur l'architecture

Les auteurs d'UP mettent en avant la préoccupation de l'architecture du système dès le début des travaux d'analyse et de conception. Il est important de définir le plus tôt possible, même à grandes mailles, l'architecture type qui sera retenue pour le développement, l'implémentation et ensuite le déploiement du système. Le vecteur des cas d'utilisation peut aussi être utilisé pour la description de l'architecture.

2.4 Processus orienté par la réduction des risques

L'analyse des risques doit être présente à tous les stades de développement d'un système. Il est important de bien évaluer les risques des développements afin d'aider à la bonne prise de décision. Du fait de l'application du processus itératif, UP contribue à la diminution des risques au fur et à mesure du déroulement des itérations successives.

3. LES CONCEPTS ET LES DEUX DIMENSIONS DU PROCESSUS UP

3.1 Définition des principaux concepts et schéma d'ensemble

Le processus unifié décrit qui fait quoi, comment et quand les travaux sont réalisés tout au long du cycle de vie du projet. Quatre concepts d'UP répondent à ces questions :

- Rôle (qui ?)
- Activité (comment ?)
- Artefact (quoi ?)
- Workflow (quand ?)

3.1.1 Rôle

Un rôle définit le comportement et les responsabilités d'une ressource ou d'un groupe de ressources travaillant en équipe. Le rôle doit être considéré en termes de « casquette » qu'une ressource peut revêtir sur le projet. Une ressource peut jouer plusieurs rôles sur le projet.

Par exemple sur un projet, Paul peut être à la fois chef de projet et architecte. Il représente deux rôles au sens d'UP (fig. 4.1).

3.1.2 Activité

Les rôles ont des activités qui définissent le travail qu'ils effectuent. Une activité est une unité de travail qu'une ressource, dans un rôle bien précis, peut effectuer et qui produit un résultat dans le cadre du projet. L'activité a un but clairement établi, généralement exprimée en termes de création ou de mise à jour d'artefacts, comme un modèle, une classe ou un planning. Les ressources sont affectées aux activités selon leurs compétences et leur disponibilité.

Par exemple, les activités « planifier une itération » et « anticiper les risques » sont attribuées au rôle de chef de projet.

Le schéma de la figure 4.1 présente sur un exemple le positionnement et les liens entre ressources, rôles et activités.

3.1.3 Artefacts

Un artefact est un ensemble d'informations qui est produit, modifié ou utilisé par le processus. Les artefacts sont les produits effectifs du projet. Les artefacts sont utilisés comme input par les ressources pour effectuer une activité et sont le résultat d'output d'activités du processus.

Un exemple d'artefacts rencontrés au cours du projet est un planning d'une itération ou un diagramme produit dans une activité.

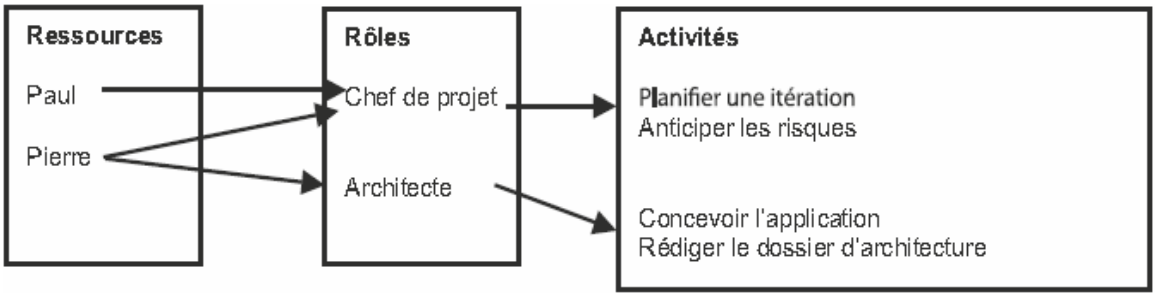


Figure 4.1 — Schéma de positionnement des ressources, rôles et activités

3.1.4 Workflows

Une énumération de tous les rôles, activités et artefacts ne constitue pas un processus. En effet, il est nécessaire d'avoir une façon de décrire des séquences d'activités mesurables qui produisent un résultat de qualité et montre l'interaction entre les ressources. Le workflow est une séquence d'activités qui produit un résultat mesurable. En UML, il peut être exprimé par un diagramme de séquence, un diagramme de communication ou un diagramme d'activité.

3.1.5 Schéma d'ensemble

UP peut être décrit suivant deux dimensions traduites en deux axes comme le montre la figure 4.2 :

- Un axe horizontal représentant le temps et montrant l'aspect dynamique du processus. Sur cet axe, le processus est organisé en phases et itérations.
- Un axe vertical représentant l'aspect statique du processus. Sur cet axe, le processus est organisé en activités et workflows.

3.2 Phases et itérations du processus (aspect dynamique)

Le processus unifié, organisé en fonction du temps, est divisé en quatre phases successives.

- Inception (Lancement).
- Elaboration.
- Construction.
- Transition.

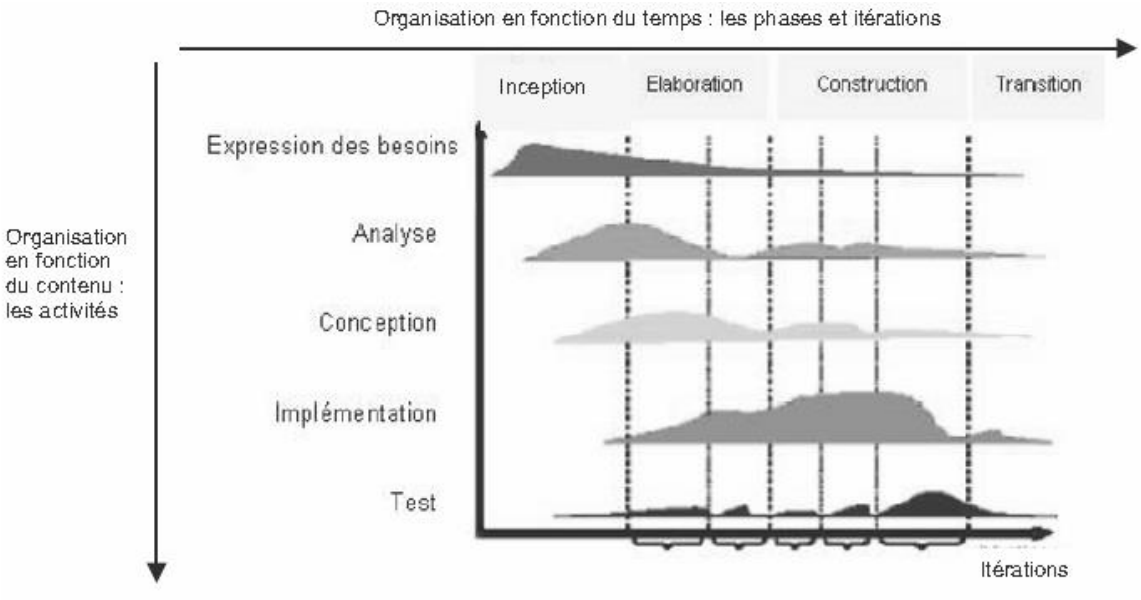


Figure 4.2 — Schéma d'ensemble d'UP

3.2.1 Capture de besoin (Inception)

Cette phase correspond à l'initialisation du projet où l'on mène une étude d'opportunité et de faisabilité du système à construire. Une évaluation des risques est aussi réalisée dès cette phase.

En outre, une identification des principaux cas d'utilisation accompagnée d'une description générale est modélisée dans un diagramme de cas d'utilisation afin de définir le périmètre du projet. Il est possible, à ce stade, de faire réaliser des maquettes sur un sous-ensemble des cas d'utilisation identifiés.

Ce n'est qu'à l'issue de cette première phase que l'on peut considérer le projet véritablement lancé.

3.2.2 Élaboration

Cette phase reprend les résultats de la phase d'inception et élargit l'appréciation de la faisabilité sur la quasi-totalité des cas d'utilisation. Ces cas d'utilisation se retrouvent dans le diagramme des cas d'utilisation qui est ainsi complété.

Cette phase a aussi pour but d'analyser le domaine technique du système à développer afin d'aboutir à une architecture stable. Ainsi, toutes les exigences non recensées dans les cas d'utilisation, comme par exemple les exigences de performances du système, seront prises en compte dans la conception et l'élaboration de l'architecture.

L'évaluation des risques et l'étude de la rentabilité du projet sont aussi précisées. Un planning est réalisé pour les phases suivantes du projet en

indiquant le nombre d'itérations à réaliser pour les phases de construction.

3.2.3 Construction

Cette phase correspond à la production d'une première version du produit. Elle est donc fortement centrée sur les activités de conception, d'implémentation et de test. En effet, les composants et fonctionnalités non implémentés dans la phase précédente le sont ici.

Au cours de cette phase, la gestion et le contrôle des ressources ainsi que l'optimisation des coûts représentent les activités essentielles pour aboutir à la réalisation du produit. En parallèle est rédigé le manuel utilisateur de l'application.

3.2.4 Transition

Après les opérations de test menées dans la phase précédente, il s'agit dans cette phase de livrer le produit pour une exploitation réelle. C'est ainsi que toutes les actions liées au déploiement sont traitées dans cette phase.

De plus, des « bêta tests » sont effectués pour valider le nouveau système auprès des utilisateurs.

- *Itérations*

Une phase peut être divisée en itérations. Une itération est un circuit complet de développement aboutissant à une livraison (interne ou externe) d'un produit exécutable.

Ce produit est un sous-ensemble du produit final en cours de développement, qui croît incrémentalement d'itération en itération pour devenir le système final.

Chaque itération au sein d'une phase aboutit à une livraison exécutable du système.

3.3 Activités du processus (aspect statique)

Les activités menées à l'intérieur des quatre phases sont plus classiques, car déjà bien documentées dans les méthodes existantes par ailleurs. Nous nous limiterons donc à ne donner qu'une brève explication de chaque activité.

3.3.1 Expression des besoins

UP propose d'appréhender l'expression des besoins en se fondant sur une bonne compréhension du domaine concerné pour le système à développer et une modélisation des procédures du système existant.

Ainsi, UP distingue deux types de besoins :

- Les besoins fonctionnels qui conduisent à l'élaboration des cas d'utilisation,
- Les besoins non fonctionnels (techniques) qui aboutissent à la rédaction d'une matrice des exigences.

3.3.2 Analyse

L'analyse permet une formalisation du système à développer en réponse à l'expression des besoins formulée par les utilisateurs. L'analyse se concrétise par l'élaboration de tous les diagrammes donnant une représentation du système tant statique (diagramme de classe principalement), que dynamique (diagramme des cas d'utilisation, de séquence, d'activité, d'état-transition...).

3.3.3 Conception

La conception prend en compte les choix d'architecture technique retenus pour le développement et l'exploitation du système. La conception permet d'étendre la représentation des diagrammes effectuée au niveau de l'analyse en y intégrant les aspects techniques plus proches des préoccupations physiques.

3.3.4 Implémentation

Cette phase correspond à la production du logiciel sous forme de composants, de bibliothèques ou de fichiers. Cette phase reste, comme dans toutes les autres méthodes, la plus lourde en charge par rapport à l'ensemble des autres phases (au moins 40 %).

3.3.5 Test

Les tests permettent de vérifier :

- La bonne implémentation de toutes les exigences (fonctionnelles et techniques),
- Le fonctionnement correct des interactions entre les objets,
- La bonne intégration de tous les composants dans le logiciel.

Classiquement, différents niveaux de tests sont réalisés dans cette activité : test unitaire, test d'intégration, test de réception, test de performance et test de non-régression.

4. MODELES MIS EN PLACE

Pour mener efficacement le cycle, les développeurs ont besoin de construire toutes les représentations du produit logiciel :

Modèle des cas d'utilisation	Expose les cas d'utilisation et leurs relations avec les utilisateurs
Modèle d'analyse	Détaille les cas d'utilisation et procède à une première répartition du comportement du système entre divers objets
Modèle de conception	Définit la structure statique du système sous forme de sous système, classes et interfaces ; Définit les cas d'utilisation réalisés sous forme de collaborations entre les sous systèmes les classes et les interfaces
Modèle d'implémentation	Intègre les composants (code source) et la correspondance entre les classes et les composants
Modèle de déploiement	Définit les nœuds physiques des ordinateurs et l'affectation de ces composants sur ces nœuds.
Modèle de test	Décrit les cas de test vérifiant les cas d'utilisation
Représentation de l'architecture	Description de l'architecture

PARTIE 2 : LE PROCESSUS DE DEVELOPPEMENT DE LOGICIEL EN UP

Dans cette deuxième partie, il sera question d'utiliser les différents diagrammes UML dans le processus de développement d'une application en respectant les phases de processus unifié.

CHAPITRE 4 : ETUDES DE FAISABILITES ET CAPTURE DE BESOINS

Ce chapitre va s'occuper de l'initialisation du projet où l'on va mener une étude de faisabilité du système à construire et capturer les besoins des utilisateurs.

1. ETUDES DE FAISABILITES

Les études de faisabilités sont considérées comme le point de départ pour la réalisation de tous projet d'informatisation. À défaut d'un avant-projet complet, les études de faisabilités décrivent les processus, préparent la gouvernance du projet et permettent de comprendre les attentes de maitre d'ouvrage. Ces études s'imposent d'autant plus dans le contexte incertain et hautement concurrentiel du moment où les maîtres mots sont « optimisation », « mutualisation », « réactivité » et « retour sur investissement ».

L'Étude de faisabilité dans la gestion de projets est une étude qui s'attache à vérifier que le projet soit techniquement faisable et économiquement viable. Dans une optique plus large, on distingue les volets suivants dans une étude de faisabilité : opérationnel, financier et technique.

Cette étude se base sur une consultation des maîtres d'œuvres potentiels, la comparaison des propositions techniques et des scénarios financiers possibles, ainsi que sur l'analyse des environnements d'affaire et l'historique des projets similaires. Au bout du compte, l'étude de faisabilité doit justifier le projet en termes d'objectifs chiffrés, réalistes, mesurables, atteignables et temporel définis, dans un contexte donné tout en présentant les moyens pour les réaliser.

1.1 Faisabilité opérationnelle

Dans l'étude de faisabilité opérationnelle, il sera question de définir la manière dont le projet sera développé en tenant compte des contraintes de temps et financières. Pour ce faire un recours aux notions de recherches opérationnelles nous ai indispensable.

Un projet est un ensemble d'opérations présentant une certaine unicité qui doivent permettre à atteindre un objectif clairement exprimable.

Le projet constitue un ensemble destiné à répondre à des objectifs précis dans un temps limité en rassemblant les ressources financières, matérielles, humaines nécessaires suivant un processus cohérant.

Le projet est caractérisé par le triplet : Projet= [Objectifs, Moyens, Délais].

1.1.1 Choix de la méthode d'ordonnancement

L'ordonnancement permet d'organiser un planning optimal des taches, mais également d'indiquer celles des taches qui peuvent souffrir de retard sans compromettre la durée totale du projet. Il s'agit ordonner dans le temps un ensemble d'opérations contribuant à la réalisation d'un projet, et ces opérations ou taches doivent respecter certaines contraintes qui peuvent être :

- Soit des contraintes d'antériorité : une tache B ne peut commencer que lorsque la tache A est terminée (contrainte de succession) ;
- Soit des contraintes de date : une tache ne peut commencer avant une certaine date. L'objet étant ici de minimiser la durée totale de réalisation de chacune des opérations et des contraintes qu'elles doivent respecter.

Il existe plusieurs méthodes d'ordonnancement qui peuvent être classifiées en deux grands groupes selon les principes de bases qu'elles utilisent :

- Les méthodes à diagramme (la méthode GANTT)
- Les méthodes à chemin critique (les méthodes CPM, MPM et PERT).

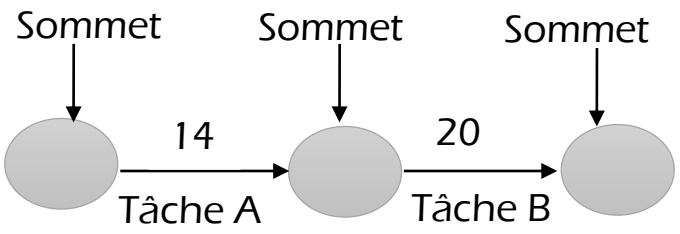
1.1.2 Présentation de la méthode PERT

La méthode PERT (Program Evaluation and Reseach Task) fut créée par les américains à la fin des années 50 dans l'objectif de gagner du temps dans la réalisation de missiles à ogives nucléaires. Cette méthode d'ordonnancement a progressivement conquis l'Amérique avant de s'installer en Europe.

Cette méthode consiste à créer un réseau qui prend en compte la chronologie des taches et leurs dépendances afin de parvenir à l'étape finale, c'est-à-dire au produit fini. Ce réseau permet ainsi de déterminer le chemin critique : les taches pour lesquelles le moindre retard entraine un retard sur l'ensemble du projet.

Dans la méthode PERT, les taches sont symbolisées par les arcs, chaque arc étant défini par son début, sa fin et sa durée. Les arcs étant orientés, les sommets entre les arcs définissant les relations d'antériorité.

Prenons un exemple, une tâche A qui dure 14 minutes suivit de la tâche B qui a une durée de 20 minutes la représentation de graphe est la suivante :



La méthode PERT se décompose selon les étapes suivantes :

- Lister les tâches d'une manière exhaustive ;
- Estimer la durée de chaque tâche ;
- Tracer le réseau PERT ;
- Déterminer les dates au plus tôt ;
- Déterminer les dates au plus tard ;
- En déduire le chemin critique ;
- Calculer les marges libres ;
- Calculer les marges totales.
- Diagramme de GANTT

1.2 Faisabilité financière

La faisabilité financière consiste à déterminer tous les couts liés à la réalisation du projet. La détermination de ces couts passe par la maitrise du calendrier prévisionnel établit lors de la faisabilité opérationnelle. L'étude de la faisabilité financière se termine par l'élaboration d'un tableau budgétaire.

2. CAPTURE DES BESOINS

La capture des besoins se subdivise en deux parties, on parle des besoins fonctionnels et des besoins techniques. Elle consiste à identifier les fonctionnalités à développer dans l'application et à décrire les spécifications techniques à respecter lors de ce développement.

2.1 Capture de besoins fonctionnels

La capture des besoins fonctionnels va permettre à l'informaticien (maitre d'œuvre) de maitriser le domaine métier après phase de spécification des besoins des utilisateurs (maitre d'ouvrage). L'informaticien charge de développement de l'application doit s'atteler à identifier les différents acteurs intervenant dans le processus manuel et le rôle qu'ils jouent dans ce système.

La capture des besoins fonctionnels recours aux diagrammes UML ci-après :

- Diagramme de cas d'utilisation ; et
- Diagramme des séquences.

2.1.1 Diagrammes de cas d'utilisation

Bien souvent, le maître d'ouvrage et les utilisateurs ne sont pas des informaticiens. Il leur faut donc un moyen simple d'exprimer leurs besoins. C'est précisément le rôle des diagrammes de cas d'utilisation qui permettent de recueillir, d'analyser et d'organiser les besoins, et de recenser les grandes fonctionnalités d'un système. Il s'agit donc de la première étape UML d'analyse d'un système.

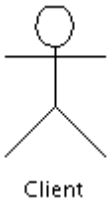
Un diagramme de cas d'utilisation capture le comportement d'un système, d'un sous-système, d'une classe ou d'un composant tel qu'un utilisateur extérieur le voit. Il scinde la fonctionnalité du système en unités cohérentes, les cas d'utilisation, ayant un sens pour les acteurs. Les cas d'utilisation permettent d'exprimer le besoin des utilisateurs d'un système, ils sont donc une vision orientée utilisateur de ce besoin au contraire d'une vision informatique.

Il ne faut pas négliger cette première étape pour produire un logiciel conforme aux attentes des utilisateurs. Pour élaborer les cas d'utilisation, il faut se fonder sur des entretiens avec les utilisateurs.

2.1.1.1 Éléments des diagrammes de cas d'utilisation

A. Acteur

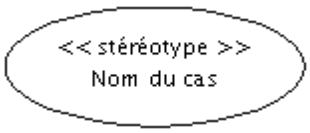
Un acteur est l'idéalisation d'un rôle joué par une personne externe, un processus ou une chose qui Interagit avec un système.
Il se représente par un petit bonhomme avec son nom (son rôle) inscrit dessous.



Il est également possible de représenter un acteur sous la forme d'un classeur stéréotypé « actor ».

B. Cas d'utilisation

Un cas d'utilisation est une unité cohérente d'une fonctionnalité visible de l'extérieur. Il réalise un service de bout en bout, avec un déclenchement, un déroulement et une fin, pour l'acteur qui l'initie. Un cas d'utilisation modélise donc un service rendu par le système, sans imposer le mode de réalisation de ce service.
Un cas d'utilisation se représente par une ellipse contenant le nom du cas (un verbe à l'infinitif), et optionnellement, au-dessus du nom, un stéréotype.



2.1.1.2 Représentation d'un diagramme de cas d'utilisation

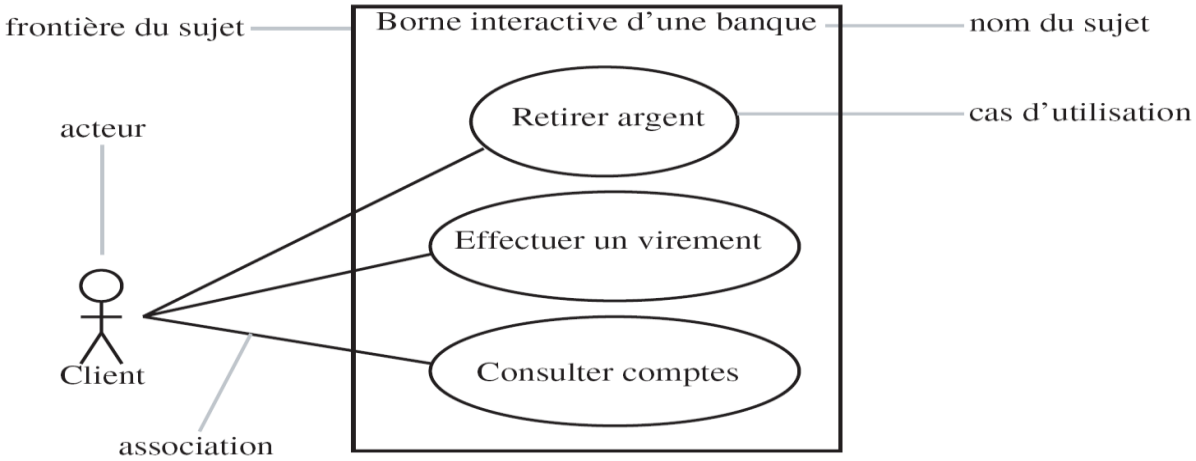
Comme le montre la figure suivante, la frontière du système est représentée par un cadre. Le nom du système figure à l'intérieur du cadre, en haut. Les acteurs sont à l'extérieur et les cas d'utilisation à l'intérieur.

2.1.1.3 Relations dans les diagrammes de cas d'utilisation

A. Relations entre acteurs et cas d'utilisation

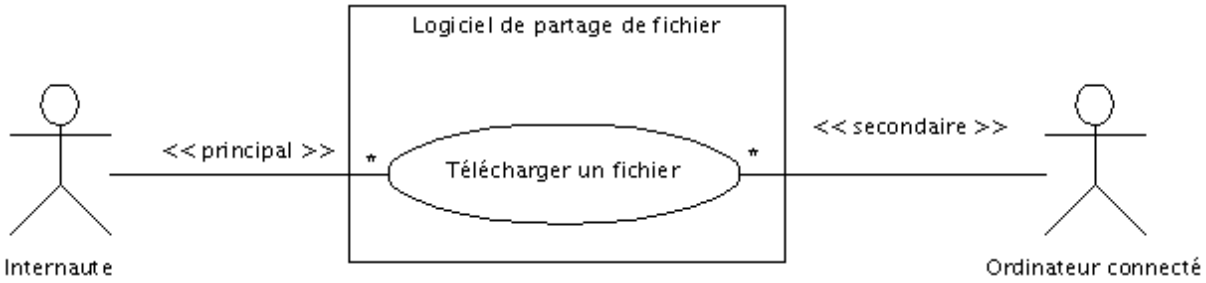
Relation d'association

Une relation d'association est un chemin de communication entre un acteur et un cas d'utilisation, elle est représentée par un trait continu.



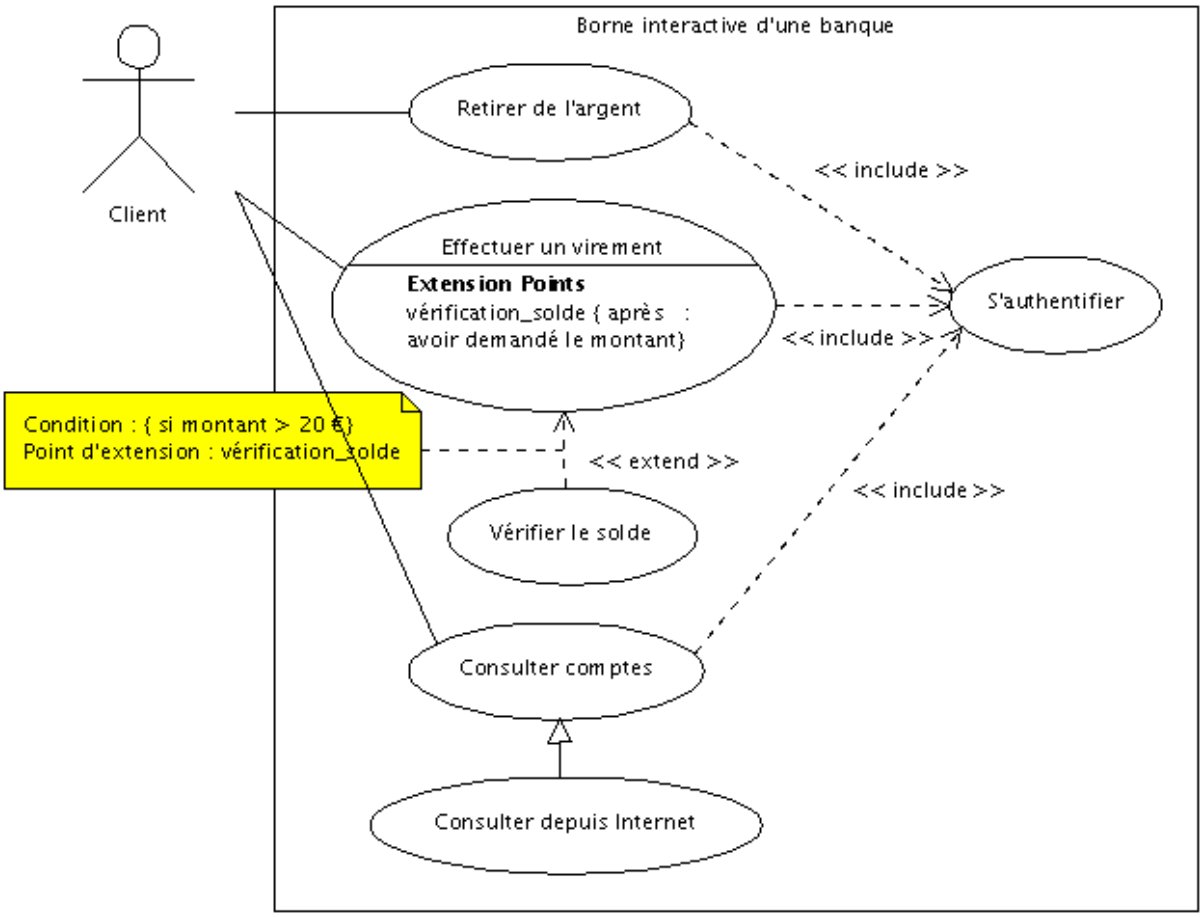
Acteurs principaux et secondaires

Un acteur est qualifié de principal pour un cas d'utilisation lorsque ce cas rend service à cet acteur. Les autres acteurs sont alors qualifiés de secondaires. Un cas d'utilisation a au plus un acteur principal. Un acteur principal obtient un résultat observable du système tandis qu'un acteur secondaire est sollicité pour des informations complémentaires. En général, l'acteur principal initie le cas d'utilisation par ses sollicitations. Le stéréotype « primary » vient orner l'association reliant un cas d'utilisation à son acteur principal, le stéréotype « secondary » est utilisé pour les acteurs secondaires.



Cas d'utilisation interne

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



Types et représentations

Il existe principalement deux types de relations :

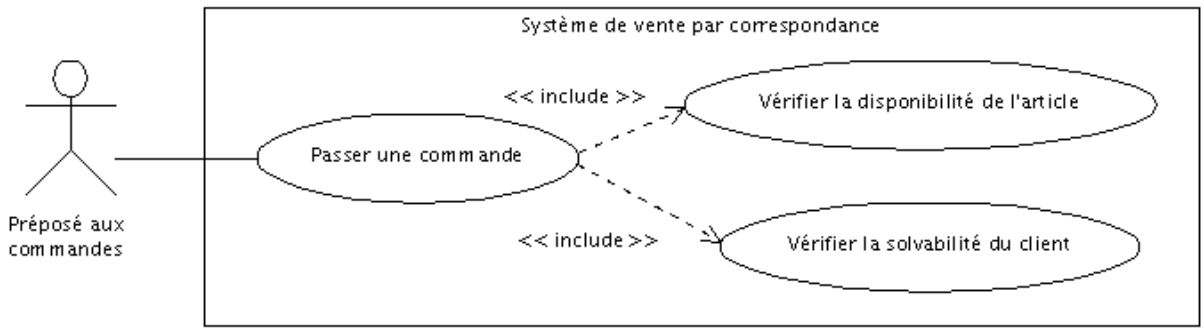
- *Les dépendances stéréotypées, qui sont explicitées par un stéréotype (les plus utilisés sont l'inclusion et l'extension) ; et*
- *La généralisation/spécialisation.*

Une dépendance se représente par une flèche avec un trait pointillé (figure ci-dessus). Si le cas A inclut ou étend le cas B, la flèche est dirigée de A vers B.

Le symbole utilisé pour la généralisation est une flèche avec un trait plein dont la pointe est un triangle fermé désignant le cas le plus général (figure ci-dessus).

Relation d'inclusion

Un cas A inclut un cas B si le comportement décrit par le cas A inclut le comportement du cas B : le cas A dépend de B. Lorsque A est sollicité, B l'est obligatoirement, comme une partie de A. Cette dépendance est symbolisée par le stéréotype « include ». Par exemple, l'accès aux informations d'un compte bancaire inclut nécessairement une phase d'authentification avec un identifiant et un mot de passe.
Les inclusions permettent essentiellement de factoriser une partie de la description d'un cas d'utilisation qui serait commune à d'autres cas d'utilisation.
Les inclusions permettent également de décomposer un cas complexe en sous-cas plus simples. Cependant, il ne faut surtout pas abuser de ce type de décomposition : il faut éviter de réaliser du découpage fonctionnel d'un cas d'utilisation en plusieurs sous-cas d'utilisation pour ne pas retomber dans le travers de la décomposition fonctionnelle.
Attention également au fait que, les cas d'utilisation ne s'enchaînent pas, car il n'y a aucune représentation temporelle dans un diagramme de cas d'utilisation.



Relation d'extension

La relation d'extension est probablement la plus utile car elle a une sémantique qui a un sens du point de vue métier au contraire des deux autres qui sont plus des artifices d'informaticiens.
On dit qu'un cas d'utilisation A étend un cas d'utilisation B lorsque le cas d'utilisation A peut être appelé au cours de l'exécution du cas d'utilisation B. Exécuter B peut éventuellement entraîner l'exécution de A : contrairement à l'inclusion, l'extension est optionnelle. Cette dépendance est symbolisée par le stéréotype « extend ».

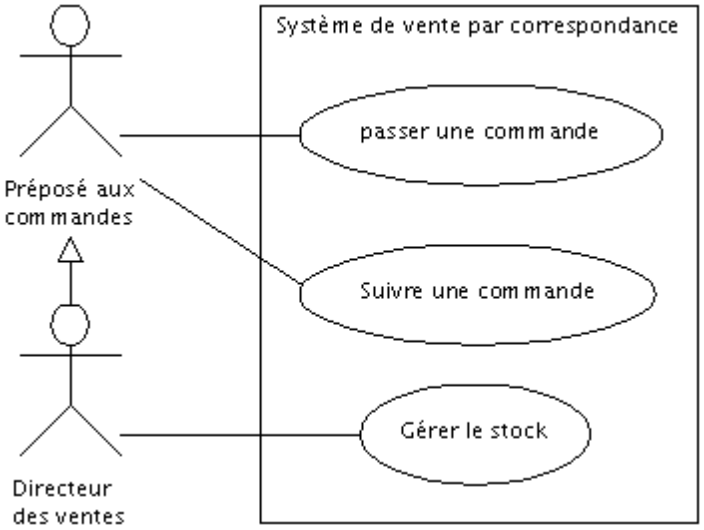
L'extension peut intervenir à un point précis du cas étendu. Ce point s'appelle le point d'extension.
Il porte un nom, qui figure dans un compartiment du cas étendu sous la rubrique point d'extension, et est éventuellement associé à une contrainte indiquant le moment où l'extension intervient. Une extension est souvent soumise à condition. Graphiquement, la condition est exprimée sous la forme d'une note.
La figure précédente présente l'exemple d'une banque où la vérification du solde du compte n'intervient que si la demande de retrait dépasse 20 euros.

Relation de généralisation

Un cas A est une généralisation d'un cas B si B est un cas particulier de A. Dans la figure précédente présentant la situation de la banque, la consultation d'un compte via Internet est un cas particulier de la consultation. Cette relation de généralisation/spécialisation est présente dans la plupart des diagrammes UML et se traduit par le concept d'héritage dans les langages orientés objet.

2.1.1.4 Relations entre acteurs

La seule relation possible entre deux acteurs est la généralisation : un acteur A est une généralisation d'un acteur B si l'acteur A peut-être substitué par l'acteur B. Dans ce cas, tous les cas d'utilisation accessibles à A le sont aussi à B, mais l'inverse n'est pas vrai.
Le symbole utilisé pour la généralisation entre acteurs est une flèche avec un trait plein dont la pointe est un triangle fermé désignant l'acteur le plus général (comme nous l'avons déjà vu pour la relation de généralisation entre cas d'utilisation).
Par exemple, la figure suivante montre que le directeur des ventes est un préposé aux commandes avec un pouvoir supplémentaire : en plus de pouvoir passer et suivre une commande, il peut gérer le stock. Par contre, le préposé aux commandes ne peut pas gérer le stock.



2.1.1.5. Décrire les cas d'utilisation

Un cas d'utilisation représente un ensemble de séquences d'interactions entre le système et ses acteurs. Pour décrire la dynamique du cas d'utilisation, le plus naturel consiste à recenser toutes les interactions de façon textuelle. Le cas d'utilisation doit par ailleurs avoir un début et une fin clairement identifiés.

Il doit préciser quand ont lieu les interactions entre acteurs et système, et quels sont les messages échangés. Il faut également préciser les variantes possibles, telles que les différents cas nominaux, les cas alternatifs, les cas d'erreurs, tout en essayant d'ordonner séquentiellement les descriptions, afin d'améliorer leur lisibilité. Chaque unité de description de séquences d'actions est appelée enchaînement. Un scénario représente une succession particulière d'enchaînements, qui s'exécute du début à la fin du cas d'utilisation.

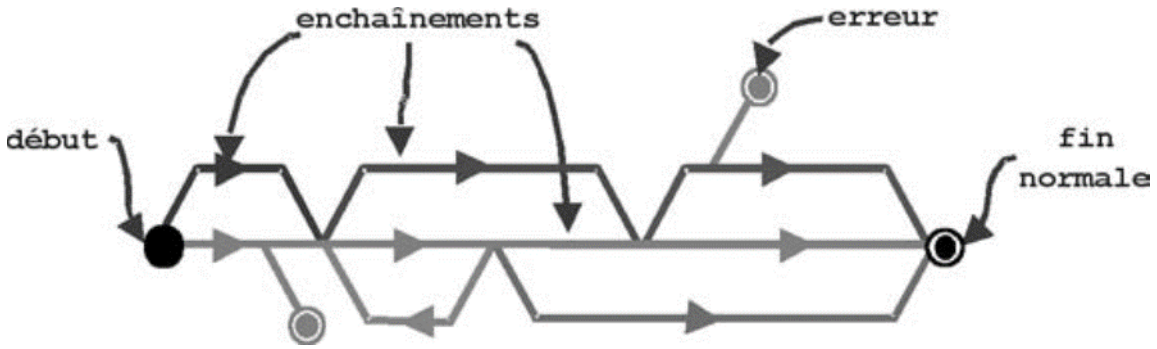


Figure 14 : Représentation des variantes d'un cas d'utilisation

Les scénarios sont aux cas d'utilisation ce que les objets sont aux classes : on peut considérer qu'un scénario est une instance particulière d'un cas

d'utilisation. L'acteur principal d'un cas d'utilisation dispose donc de l'ensemble des enchaînements pour réaliser une certaine tâche métier. Les exceptions décrivent les interruptions possibles d'exécution empêchant l'acteur d'obtenir sa plus-value métier.

2.1.1.6. Complétez les descriptions textuelles avec des diagrammes dynamiques simples

Pour documenter les cas d'utilisation, la description textuelle est indispensable, car elle seule permet de communiquer facilement et précisément avec les utilisateurs.

Elle est également l'occasion de s'entendre sur la terminologie employée, ainsi que d'identifier le contexte d'exécution de l'un ou de l'autre des enchaînements. En revanche, le texte présente des désavantages puisqu'il est difficile de montrer comment les enchaînements se succèdent ; en outre la maintenance des évolutions s'avère souvent périlleuse.

Il est donc recommandé de compléter la description textuelle par un ou plusieurs diagrammes dynamiques, qui apporteront un niveau supérieur de formalisation.

À vous de décider en fonction de votre contexte si vous montrez ces diagrammes au futur utilisateur, ou si vous les utilisez uniquement comme support d'analyse pour lui poser des questions supplémentaires, et ainsi mieux valider votre texte.

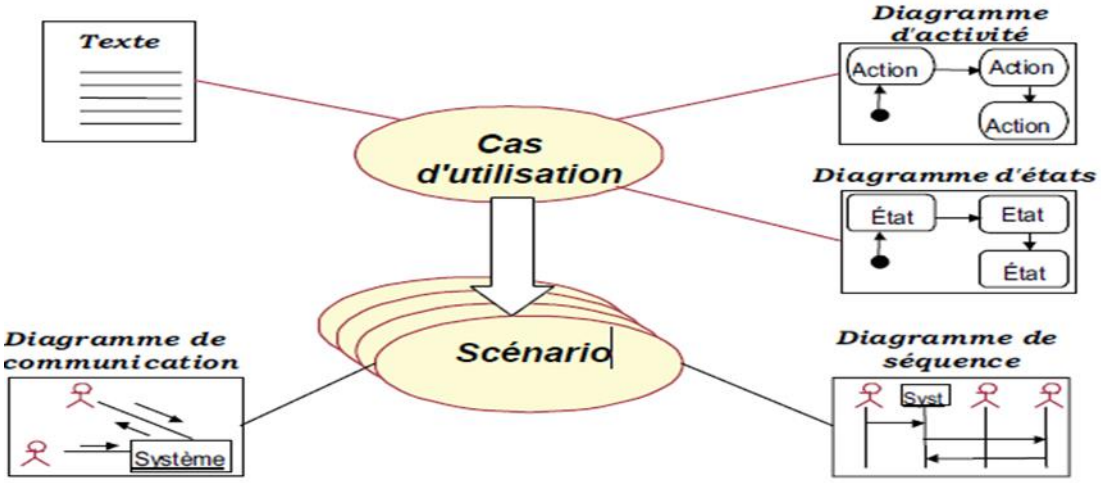


Figure 15 : Types de diagrammes dynamiques utilisables pour documenter les cas d'utilisation

Les critères de choix entre les différents types de diagrammes dynamiques utilisables sont énumérés de façon plus détaillée ci-après :

- ***Pour documenter les cas d'utilisation :***
  - Le diagramme d'activité est celui que nous recommandons particulièrement, car il permet de consolider les enchaînements de la fiche textuelle. Ce diagramme est également très utile en cas d'actions parallèles. De plus, les utilisateurs le comprennent aisément, car il ressemble à un organigramme traditionnel. Il permet enfin d'identifier d'un seul coup d'œil la famille des scénarios d'un cas d'utilisation qui décrivent toutes les réactions du système. Il suffit en effet de dessiner les différents chemins du diagramme d'activité qui passent par toutes les transitions entre actions.
  - Le diagramme d'états se prête mieux à la modélisation d'un déroulement événementiel. Il est néanmoins plus complexe à comprendre pour les utilisateurs du monde de la gestion.
- ***Pour illustrer des scénarios particuliers :***
  - Le diagramme de séquence est une bonne illustration. Il est facilement compris par les utilisateurs.
  - Le diagramme de communication est une autre illustration possible. Il est cependant ici moins utile que le précédent pour les utilisateurs, car il rend la séquence moins claire, sans apporter de véritable plus-value au diagramme de séquence.

2.1.1.7 Validation et consolidation de cas d'utilisation

La révision des cas d'utilisation doit absolument inclure une phase de présentation aux futurs utilisateurs et poser les questions clés ci-après :

- Les frontières du système sont-elles bien définies ?
- Les acteurs sont-ils tous pris en compte (au moins une fois) ?
- Chaque cas d'utilisation a-t-il un processus de déclenchement (par un acteur) ?
- Le niveau d'abstraction des cas d'utilisation est-il homogène ?
- Toutes les fonctionnalités du système sont-elles traitées ?

Il y a nécessité d'assurer une traçabilité des besoins des utilisateurs avec les cas d'utilisation.

Concrètement, on réalisera une matrice de traçabilité entre cas d'utilisation et éléments de cahier des charges. Cette matrice pourra être détaillée pour faire apparaître les différents enchaînements de chaque cas d'utilisation.

		Exigence 1	Exigence 2	Exigence 3	Exigence 4	Exigence 5
Use case A	scénario 1					
	scénario 2		X		X	X
	scénario 3		X		X	X
Use case B	scénario 1					
	scénario 2		X		X	
	scénario 3		X		X	
	scénario 4					
Use case C	scénario 1					
	scénario 2		X		X	

Cela permettra, dans la suite de l'analyse, d'établir le suivi des classes avec le cahier des charges, par le biais des cas d'utilisation ; il sera même possible de retrouver les opérations impliquées. L'aide d'un outil CASE est alors précieuse. Cette traçabilité entre besoins de haut niveau et opérations permet d'améliorer notablement la capacité de maintenance et d'évolution du code, tout au long de la vie de l'application.

2.1.1.8 Définition des itérations et incrémentations

Dans le cadre d'un développement itératif et incrémental, il est très utile de recourir au découpage en cas d'utilisation pour définir les itérations. À cet effet, il convient en premier lieu d'identifier les cas d'utilisation les plus critiques en termes de gestion des risques. Ces cas d'utilisation devront être traités prioritairement afin de lever au plus tôt les risques majeurs. Il sera également demandé au client d'affecter une priorité

fonctionnelle à chaque cas d'utilisation, afin de livrer d'abord les cas d'utilisation les plus demandés.

Ces deux critères pouvant être contradictoires, la décision du découpage en itérations incombe au chef de projet, qui doit le faire valider par le client.

Il faut aussi prendre en compte les éventuelles relations entre cas d'utilisation :

- Développer plutôt les cas factorisés (<<include>>) avant ceux qui les utilisent ;
- Développer plutôt les cas qui étendent (<<extend>>) après les cas de base.

Exemple d'un diagramme de cas d'utilisation global

Lors de développement d'une application de gestion de cours en ligne pour le compte d'une institution d'enseignement supérieur de la place le diagramme de cas d'utilisation ci-après a été obtenu :


enseignant
gestion technique de la plate forme
gestion des étudiants
gestion des examens et TP
gestion des cours
inscription
gestion de formation
accès aux cours
Administrateur
authentification
Etudiant
gestion des comptes utilisateurs
faire les examens et TP
consultation des résultats
modification profil
<<include>>

Description textuelle des cas d'utilisation

Afin de décrire les interactions entre les cas d'utilisation, nous présentons ces derniers de façon textuelle.

Il s'agit donc d'associer à chaque cas d'utilisation un nom, un objectif, les acteurs qui y participent, les préconditions et des scénarii. Cependant il existe trois types de scénarii : les scénarii nominaux ; les scénarii d'exceptions et les scénarii alternatifs. Dans cette description textuelle, nous présentons seulement les scénarii nominaux et alternatifs.

CU1: Inscription au site
Résumé: Ce CU permet à l'acteur s'inscrire.
Acteurs: Enseignant, Etudiant
Post-Condition: le cas démarre après le point 02 de l'enchainement nominal, l'utilisateur s'inscrit au site
Scénario nominal
DESCRIPTION DU SCENARIO NOMINAL « DEBUT» 01 : le système affiche un formulaire d'inscription à l'acteur 02 : l'acteur saisit ses informations. 03 : le système vérifie la validité des informations saisies. 04 : le système enregistre ces informations dans la base de données. 05 : le système notifie l'acteur du bon déroulement de l'inscription « FIN»
Scenario alternative
Les informations sont manquantes ou incorrectes : ce scénario commence au point 03 du scénario nominal. 01 : Le système informe l'acteur que les données saisies sont erronées, garde les informations saisies avant et le scénario reprend au point 02 du scénario nominal.

CU2: Authentification
Résumé: Ce CU permet à un utilisateur de se connectant au système ; et lui présenter l'interface, les fonctionnalités relatives à son profil.
Acteurs: Enseignant, Etudiant, Administrateur
Pré conditions : Introduire login et mot de passe
Post-Condition: le cas démarre après le point 02 de l'enchainement nominal, l'utilisateur s'authentifie
Scénario nominal
DESCRIPTION DU SCENARIO NOMINAL « DEBUT» 01 : Le système invite l'acteur à entrer son login et son mot de passe. 02 : L'acteur saisit le login et le mot de passe et choisit son profil.

03 : Le système vérifie les paramètres. 04 : Le système ouvre l'espace de travail correspondant au profil. « FIN»
Scenario alternative
DESCRIPTION DU SCENARIO ALTERNATI F Le login ou le mot de passe est incorrect : ce scénario commence au point 03 du scénario nominal. 01 : Le système informe l'acteur que les données saisies sont erronées et le scénario reprend au point 01 du scénario nominal.

Exercice sur le diagramme de cas d'utilisation

La création d'un site web passe par l'élaboration de deux parties, la première partie concerne la création et la mise en place des pages accessibles par tous les Internautes, la seconde partie c'est la configuration et la mise en place de l'espace d'administration.

Les pages accessibles par l'Internaute peuvent être :

- Pages simples.
- Pages NEWS (actualités)
- Page contact pour l'envoie de message.
- Pages produits.

A traves l'espace d'administration, l'administrateur de site web peut :

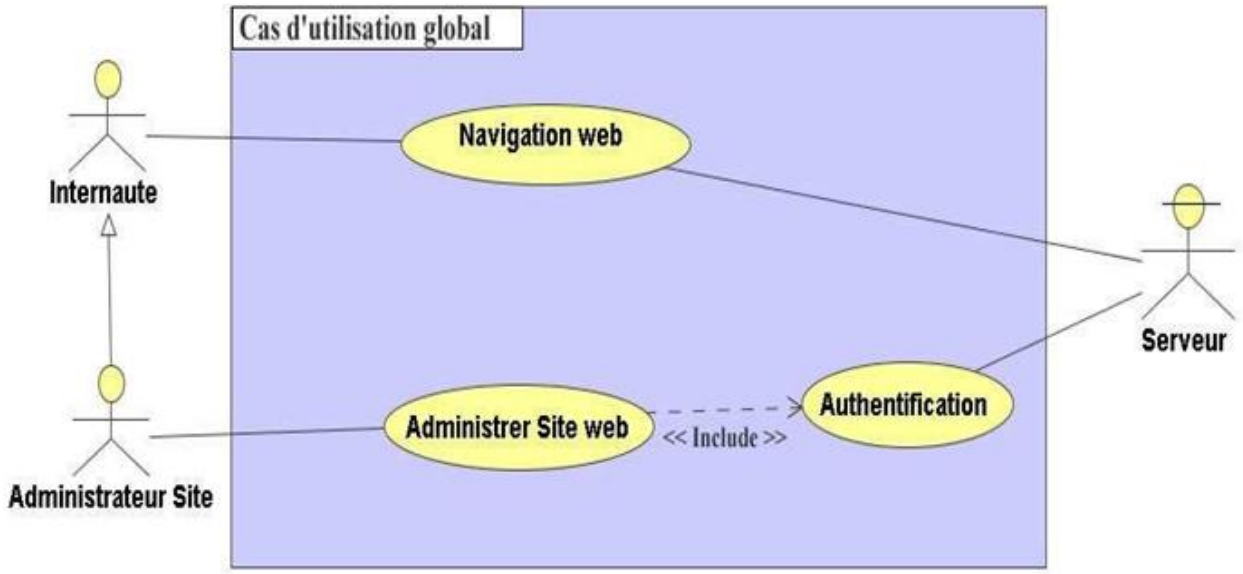
- Modifier le contenu des pages simple
- Mettre à jour la liste des produits
- Mettre à jour de la liste des familles de produits
- Modifier l'adresse e-mail pour l'envoie de message à travers la page contact.
- Charger les images pour les utiliser dans les pages simples ou les pages NEWS.
- Modifier le mot de passe d'accès à l'espace d'administration.

Travail à faire :

- Présenter les DCU et la description textuelle

Résolution

- DCU Global



2.1.2 Diagramme de séquence

Les principales informations contenues dans un diagramme de séquence sont les messages échangés entre les lignes de vie, présentés dans un ordre chronologique. Ainsi, contrairement au diagramme de communication, le temps y est représenté explicitement par une dimension (la dimension verticale) et s'écoule de haut en bas.

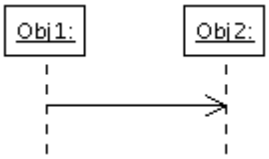
2.1.2.1 Représentation des messages

A. Messages synchrones et asynchrones, création et destruction d'instance

Un message définit une communication particulière entre des lignes de vie. Plusieurs types de messages existent, les plus communs sont :

- L'envoi d'un signal ;
- L'invocation d'une opération ;
- La création ou la destruction d'une instance.

Une interruption ou un évènement sont de bons exemples de signaux. Ils n'attendent pas de réponse et ne bloquent pas l'émetteur qui ne sait pas si le message arrivera à destination, le cas échéant quand il arrivera et s'il sera traité par le destinataire. Un signal est, par définition, un message asynchrone.



Représentation d'un message asynchrone.

Graphiquement, un message asynchrone se représente par une flèche en traits pleins et à l'extrémité ouverte partant de la ligne de vie d'un objet expéditeur et allant vers celle de l'objet cible.
L'invocation d'une opération est le type de message le plus utilisé en programmation objet. L'invocation peut être asynchrone ou synchrone. Dans la pratique, la plupart des invocations sont synchrones, l'émetteur reste alors bloqué le temps que dure l'invocation de l'opération.

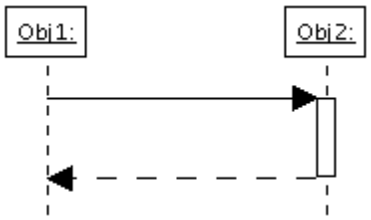
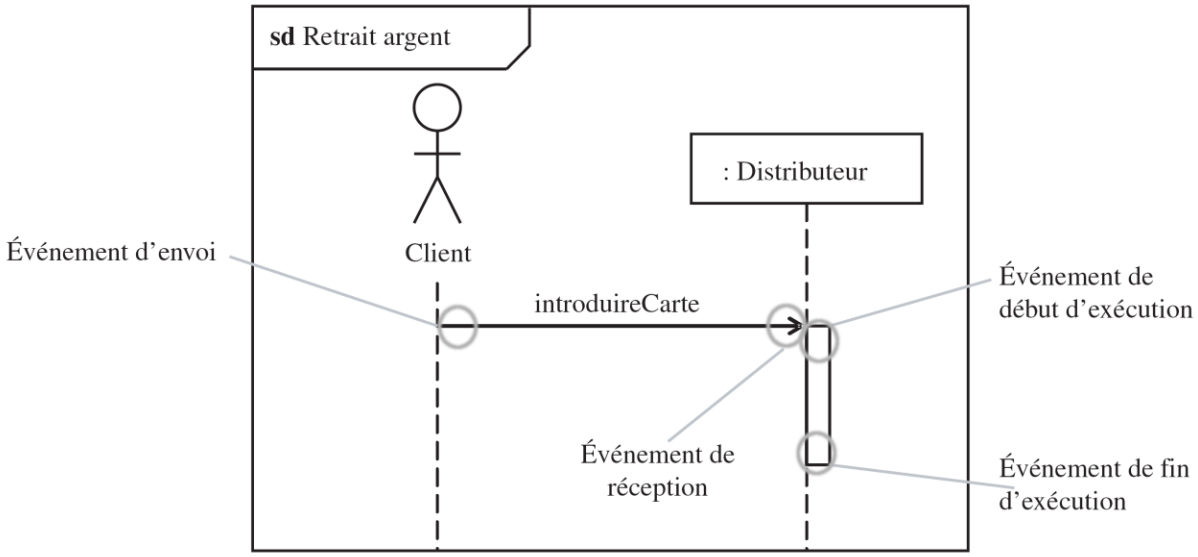


Fig. Représentation d'un message synchrone

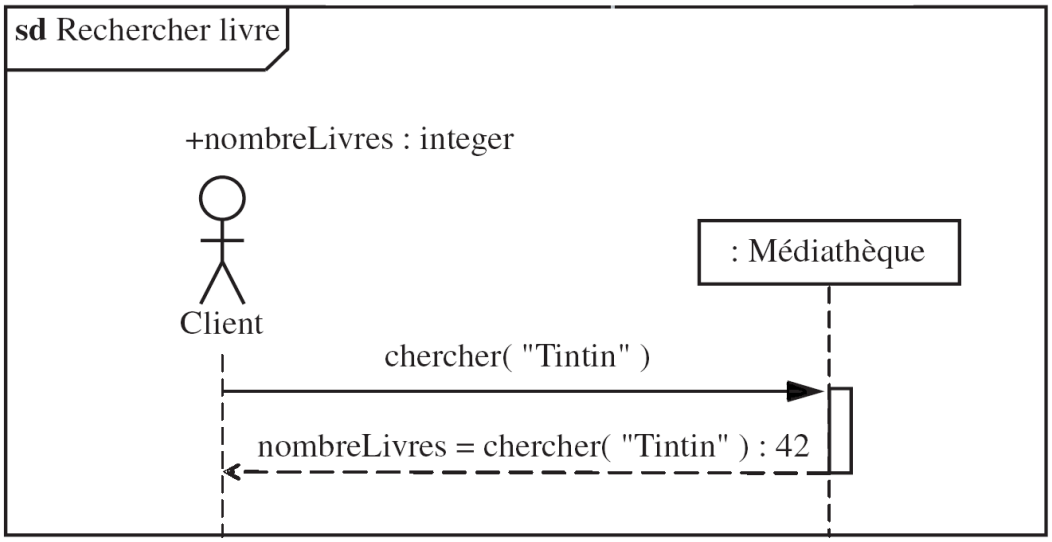
Graphiquement, un message synchrone se représente par une flèche en traits pleins et à l'extrémité pleine partant de la ligne de vie d'un objet expéditeur et allant vers celle de l'objet cible. Ce message peut être suivi d'une réponse qui se représente par une flèche en pointillé.
La création d'un objet est matérialisée par une flèche qui pointe sur le sommet d'une ligne de vie. La destruction d'un objet est matérialisée par une croix qui marque la fin de la ligne de vie de l'objet.



Les différents évènements correspondant à un message asynchrone.

UML permet de séparer clairement l'envoi du message, sa réception, ainsi que le début de l'exécution de la réaction et sa fin.

B. Syntaxe des messages et des réponses



Syntaxe des messages et des réponses.

Dans la plupart des cas, la réception d'un message est suivie de l'exécution d'une méthode d'une classe. Cette méthode peut recevoir des arguments et la syntaxe des messages permet de transmettre ces arguments.

C. Autre utilisation du diagramme de séquence

Le diagramme de séquence peut être aussi utilisé pour documenter un cas d'utilisation. Les interactions entre objets représentent, dans ce cas, des flux d'informations échangés et non pas de véritables messages entre les

opérations des objets. Un exemple de cette utilisation du diagramme de séquence est donné à la figure 3.54.

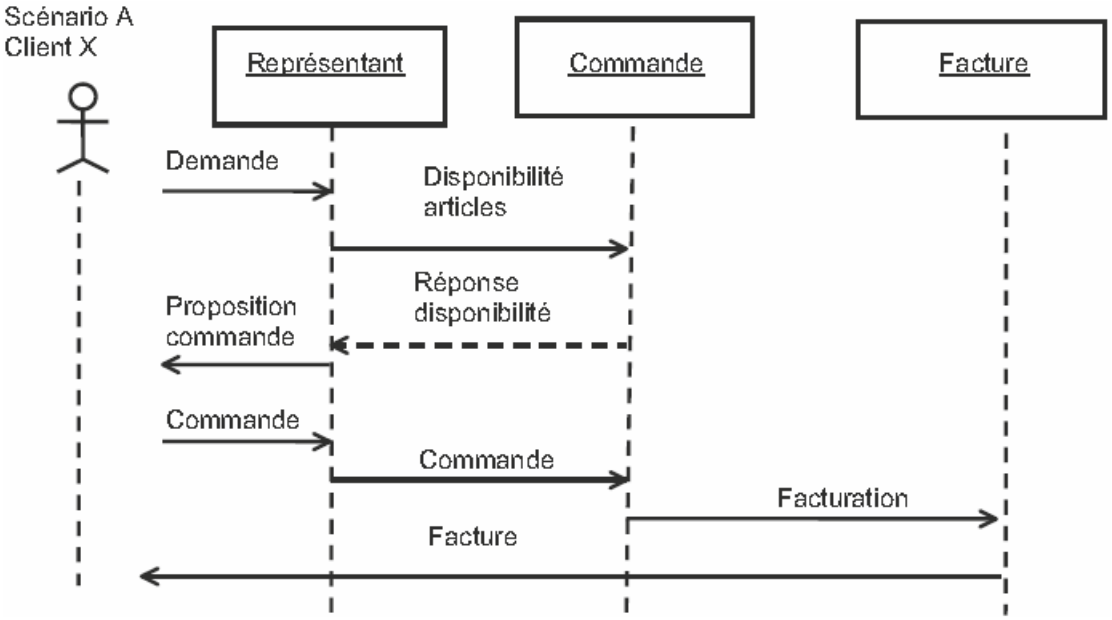
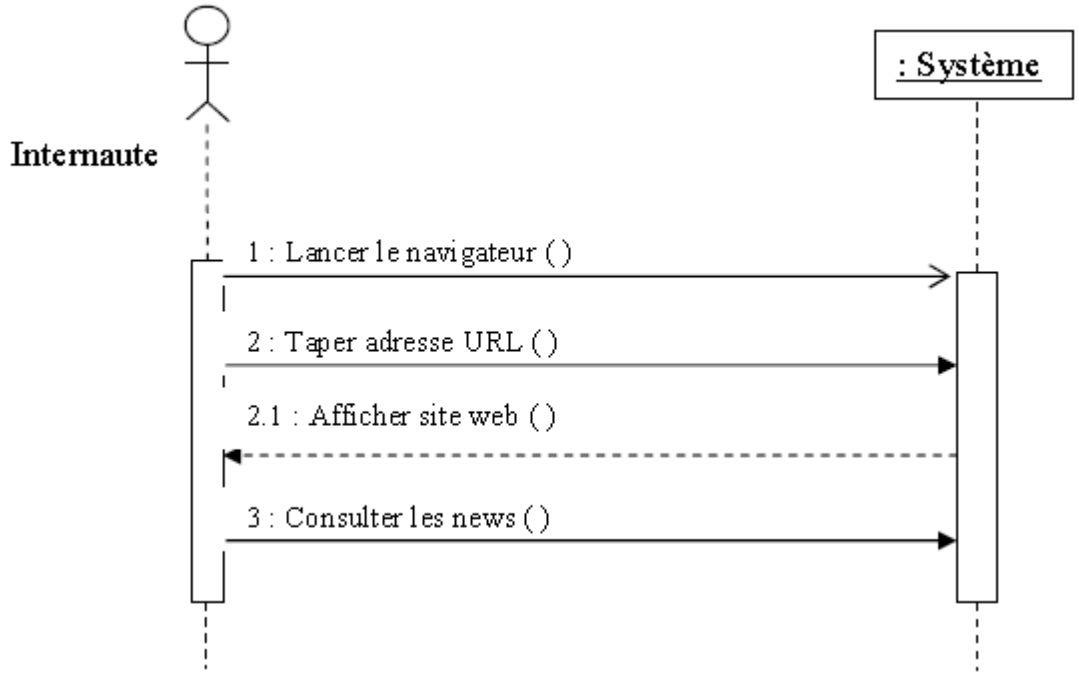


Figure — Exemple de diagramme de séquence associé à un cas d'utilisation

Exercice sur le diagramme de séquence

Se référant à l'exercice sur le diagramme de cas d'utilisation de la page 37. Le diagramme de séquence de scenario consulter page Web se présente comme suit :



2.2 Capture de besoins techniques

La capture des besoins techniques couvre, par complémentarité avec celle des besoins fonctionnels, toutes les contraintes qui ne traitent ni de la description du métier des utilisateurs, ni de la description applicative. Le modèle de spécification logicielle concerne donc les contraintes techniques. La spécification technique est une activité primordiale pour la conception d'architecture.

Cette étape a lieu lorsque les architectes ont obtenu suffisamment d'informations sur les prérequis techniques. Ils doivent a priori connaître au moins le matériel, à savoir les machines et réseaux, les progiciels à intégrer, et les outils retenus pour le développement. En cours d'élaboration, viendront s'ajouter les contraintes non fonctionnelles identifiées dans les cas d'utilisation. Le niveau d'abstraction à atteindre est l'analyse technique. Le modèle s'y exprime suivant les deux points de vue que sont la spécification logicielle et la structure du matériel à exploiter. Cette étape se termine lorsque le niveau de description des cas d'utilisation techniques a permis l'identification des problèmes à résoudre. À ce moment-là pourra débuter l'étape de conception générique, qui consiste à construire une solution d'architecture technique.

2.2.1 Spécification technique du point de vue matériel

Ces choix impliquent des contraintes relatives à la configuration du réseau matériel. Elles sont de nature géographique, organisationnelle, et technique. Elles concernent les performances d'accès aux données, la sécurité du système, l'interopérabilité, l'intégration des applications, la volumétrie et le mode d'utilisation du système.

2.2.2 Style d'architecture en niveaux

Le style d'architecture en niveaux spécifie le nombre de niveaux géographiques et organisationnels où vont se situer les environnements d'exécution du système.

- L'architecture à deux niveaux met en œuvre un environnement de travail de niveau départemental et local. Un tel système répond généralement à la demande d'un métier particulier dans l'entreprise. Par exemple, le département des ressources humaines dispose d'un système informatique indépendant et localisé au sein de la société.
- L'architecture à trois niveaux met en œuvre le système informatique d'une entreprise. Nous y trouvons les niveaux suivants : central, départemental et local. Une telle architecture couvre les différents métiers de l'entreprise.
- La contrainte géographique conditionne également l'architecture en niveaux. Le système de ressources humaines, réparti sur plusieurs agences ou départements, devient de fait un système à trois niveaux. La conjonction des contraintes géographique et organisationnelle conduit donc à des systèmes complexes dotés d'une architecture multiniveau.

Les contraintes techniques amènent également à diversifier le nombre et le type des machines :

- soit pour des raisons de performances
- soit pour des raisons de sécurité
- soit pour des raisons d'interopérabilité
- soit pour des raisons de disponibilité

2.2.3 Structuration des spécifications d'exploitation technique autour du modèle de configuration matérielle

Les spécifications qui concernent l'exploitation technique d'un réseau ont toutes une relation directe soit avec une connexion, soit avec une machine particulière du modèle de configuration matérielle. Du fait de leur existence, les machines imposent des contraintes de performances ou d'intégration matérielle. La nature des connexions permet également de spécifier des contraintes liées au besoin de communication et de bande passante. L'intégration de l'application dans le système d'information existant impose de nouvelles contraintes liées aux machines dédiées à une fonction particulière du système informatique.

2.2.4 Spécification d'architecture et influence sur le modèle de déploiement

L'expression des prérequis techniques implique également le choix d'un style d'architecture client/serveur. Ce choix conditionne la façon dont seront organisés et déployés les composants d'exploitation du système.

2.2.5 Style d'architecture en tiers

Le style d'architecture en tiers (*tier* signifie « partie » en anglais) spécifie l'organisation des composants d'exploitation mis en œuvre pour réaliser le système. Chaque partie indique une responsabilité technique à laquelle souscrivent les différents composants d'exploitation d'un système.

On distingue donc plusieurs types de composants en fonction de la responsabilité technique qu'ils jouent dans le système. Un système client/serveur fait référence à au moins deux types de composants, qui sont les systèmes de base de données en serveur, et les applications qui en exploitent les données en client.

Le style d'architecture 2-tiers correspond à la configuration la plus simple d'un système client/serveur. Dans ce cas, il incombe aux clients de gérer

l'interface utilisateur et les processus d'exploitation. Les serveurs ont pour responsabilité de traiter le stockage des données. Ce type d'architecture est parfaitement bien adapté aux systèmes locaux, dans la mesure où les concepts et les processus manipulés n'existent qu'une seule fois au sein d'un département de l'entreprise.

Dans le cadre des architectures d'entreprise, certains concepts et processus sont communs à plusieurs domaines d'activité. Cette caractéristique implique une synchronisation souvent complexe des données entre différents départements de l'entreprise. Le concept d'objet métier consiste à centraliser cette gestion afin d'en maîtriser la complexité. L'objet métier est à la fois un modèle d'analyse qui colle à la réalité du problème de l'entreprise, mais également un modèle de composant d'exploitation qui s'insère dans le déploiement du système d'entreprise. L'intégration des objets métier sous la forme de composants métiers fait passer l'architecture client/serveur du 2-tiers au 3-tiers, car elle implique un nouveau type de composants d'exploitation qui s'insère entre les clients et les serveurs de données.

2.2.6 Phases de réalisation en capture des besoins techniques

La capture des besoins techniques est une étape de prise en compte des contraintes techniques et logicielles. Elle doit être suffisamment détaillée pour permettre de passer à la phase d'élaboration.

Le processus de construction mis en œuvre dans l'étape est le suivant :

1. Capture des spécifications techniques liées à la configuration matérielle :
    - identifier les contraintes techniques liées aux machines, aux connexions et aux déploiements existants ;
    - produire le diagramme de configuration matérielle ;
    - identifier les contraintes d'organisation spécifiées par les

choix d'architecture.

2. Capture initiale des spécifications logicielles :
    - identifier les besoins logiciels du point de vue des exploitants ;
    - élaborer la description sommaire des cas d'utilisation techniques.
3. Spécification logicielle détaillée :
    - identifier un découpage en couches logicielles ;
    - identifier les cas d'utilisation techniques pour chaque couche ;
    - élaborer la description détaillée des cas d'utilisation techniques.

CHAPITRE 5 : ELABORATION DU SYSTEME

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

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

Ainsi, toutes les exigences non recensées dans les cas d'utilisation seront prises en compte dans la conception et l'élaboration de l'architecture.

L'évaluation des risques et l'étude de la rentabilité du projet sont aussi précisées. Un planning est réalisé pour les phases suivantes du projet en indiquant le nombre d'itérations à réaliser pour les phases de construction.

1. REPARTITION DE DCU GLOBAL EN PACKAGE

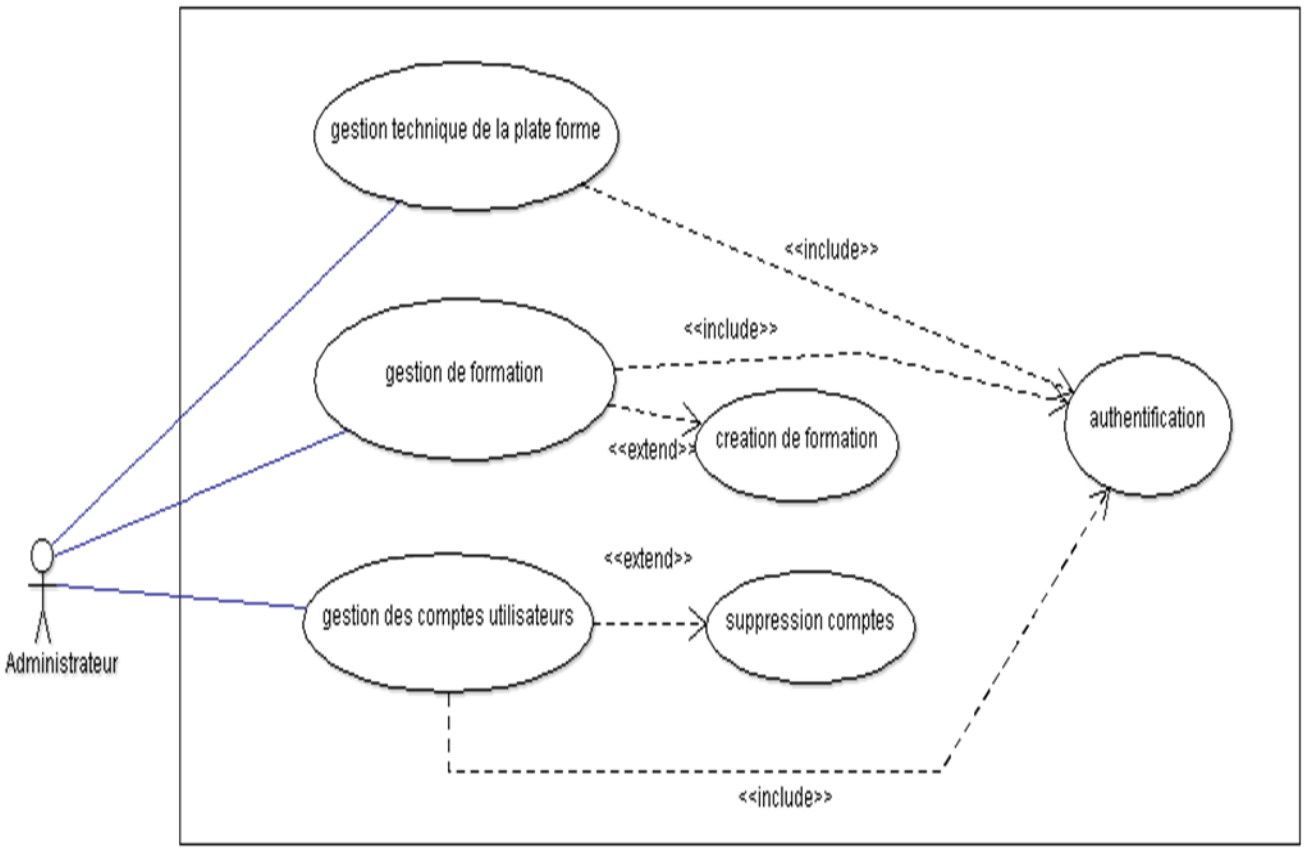
A ce stade il est question d'affiner le diagramme de cas d'utilisation global en package afin de bien maitriser les spécifications de maitre d'ouvrage.

Pour faciliter la maitrise des fonctionnalités le DCU global présenté lors de la capture de besoins sera découpé en sous-systèmes. Un sous-système (appelé package) doit avoir un nom et regrouper une famille de fonctionnalités clairement identifiable

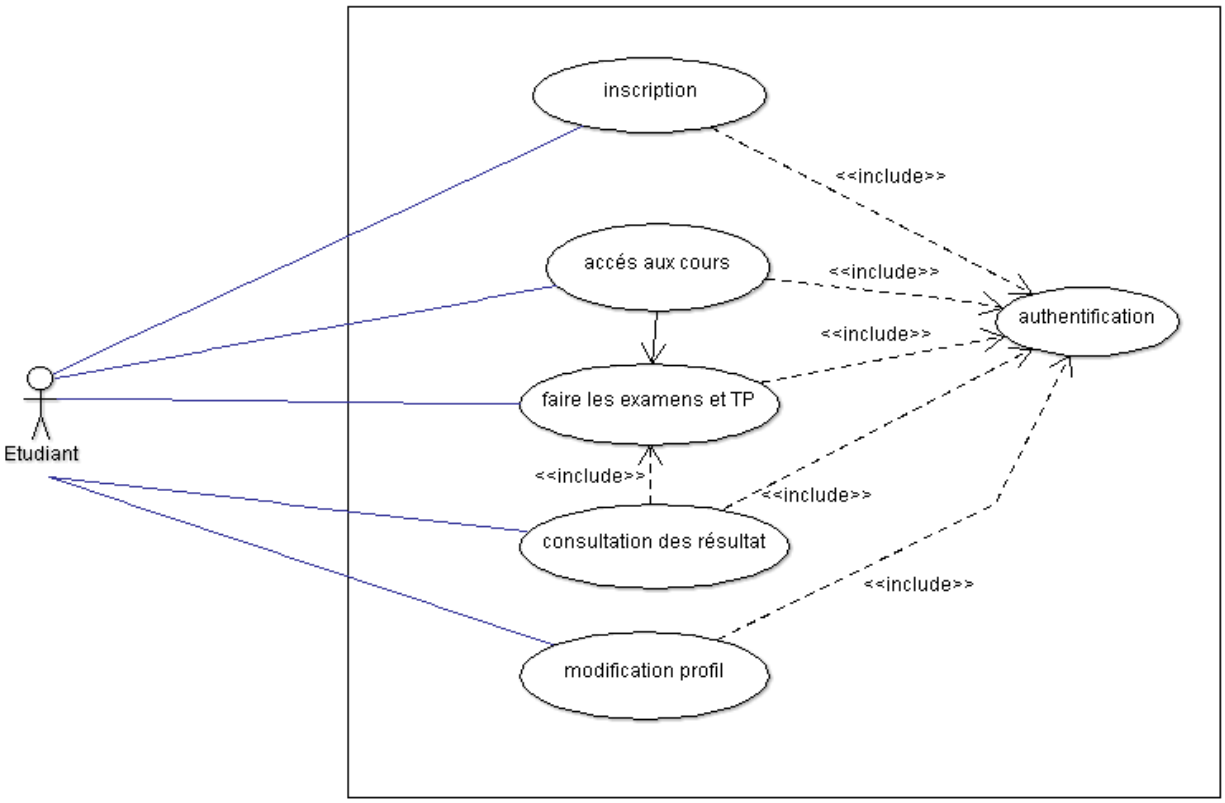
En se basant sur l'exemple de développement d'une application de gestion de cours en ligne pour le compte d'une institution d'enseignement supérieur, le DCU Global sera donne dans le tableau présentant les acteurs et les cas d'utilisation, en affectant chaque cas d'utilisation à un package. Nous obtenons le tableau ci-dessous :

Cas d'utilisation	Acteurs	package
Gestion technique de la plate forme	administrateur	Gestion de l'administration
Gestion de formation		
Gestion des comptes utilisateurs		
Gestion des étudiants	enseignant	Gestion des enseignants
Gestion des examens et TP		
Gestion des cours		
inscription	Etudiant	Gestion des étudiants
Accès aux cours		
Faire les examens et TP		
Consultation des résultats		
Modification profil		

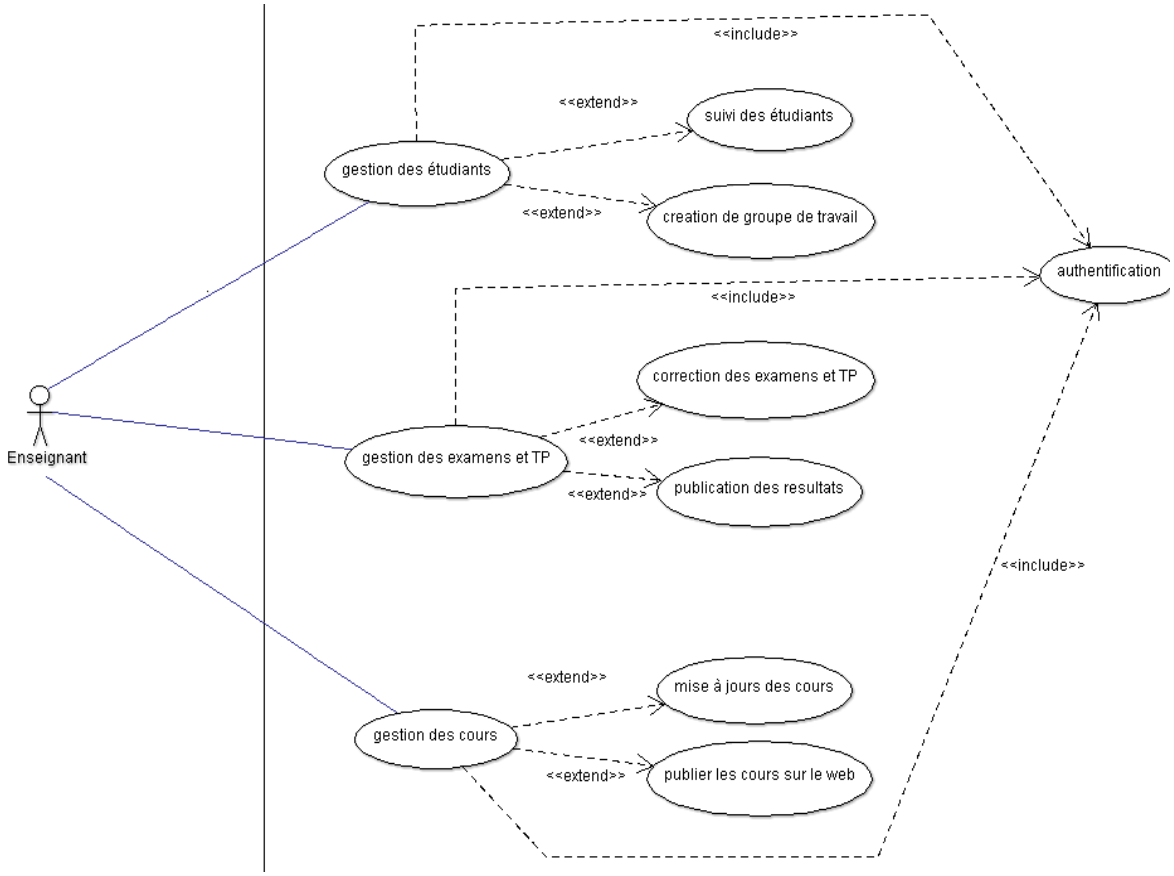
a) Package «gestion de l'administration»



b) Package «gestion des étudiants »

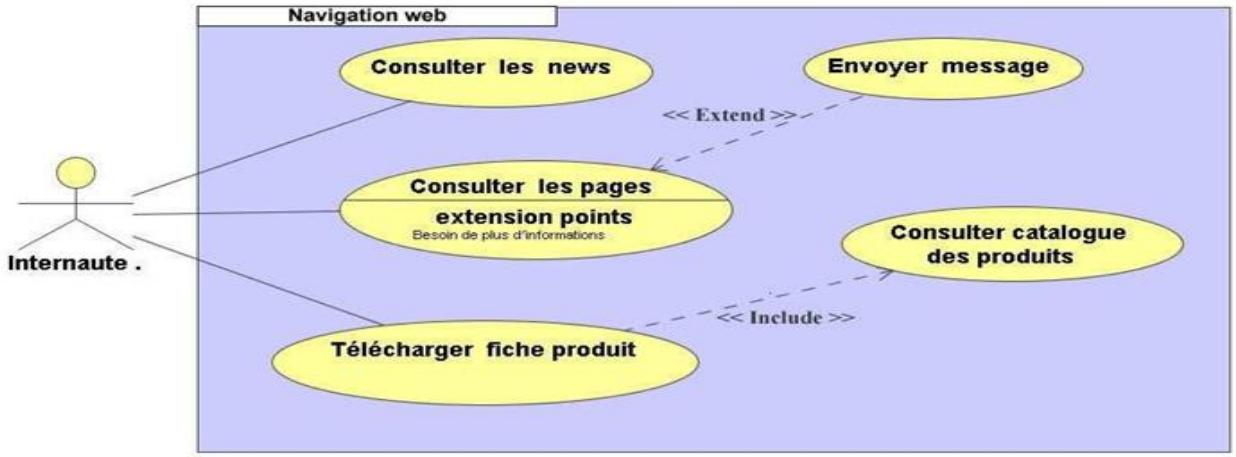


c) Package «gestion des enseignants »



Exercices sur la création d'un site WEB, le package Internaute se présentera comme suit :

- DCU Navigation web :



Description textuelle : cas d'utilisation Consulter NEWS

FICHE DE DESCRIPTION TEXTUELLE
Cas d'utilisation : Consulter News Nom de l'application : Création d'un site Web pour la gestion de news Nom du développeur : Prof. Dr. Ir. MUSANGU Luka
Objectif : Ce cas d'utilisation vise à décrire toutes les étapes relatives à la consultation de news par l'internaute. **Acteur principal :** Internaute. **Acteur secondaire :** **Date :** 2016 **Précondition :** ▪ Disponibilité d'accès au réseau Internet. ▪ Serveur accessible 24 h/24h et 7j/7j. ▪ Navigateur en bon état de fonctionnement. ▪ Existence de la page news. ▪ Le système en état de fonctionnement. **Scénario nominal :** 1- l'internaute lance le navigateur. 2- l'internaute tape l'adresse URL de site Web. 3- Le système affiche le site Web. 4- l'internaute demande au système la page des news. 5- Le système affiche la page des news. **Scénario alternatif :** E1 : Cas où il n'y a pas de news. Le système affiche un message de l'inexistence de news. **Scénario d'exception :** E1 : Erreur dans d'adresse URL. Aller à l'opération qui suit 2. 3- le navigateur affiche un message d'erreur **Poste condition:** ▪ news consulté

2. CHOIX DE L'ARCHITECTURE DU SYSTEME

A ce stade, il sera question de choisir l'architecture la plus appropriée pour le système à mettre en place. Ce choix sera basé sur les argumentaires développés lors de la capture de besoins techniques

Le réseau informatique est une technique qui consiste à relier un certain nombre de matériel informatique (ordinateur et périphérique) dans l'objectif de partager les ressources de ce derniers (ordinateur et périphérique).

Pour permettre à notre logiciel de bien fonctionner en tenant compte de l'emplacement géographique de différent bureau de l'entreprise, nous avons opté pour un réseau local avec une architecture Client/serveur 2 tiers. Cette architecture nous permettra de partager les ressources du serveur d'application pour être utilisé dans le différent poste utilisateurs et d'assurer le service d'impression en réseau.

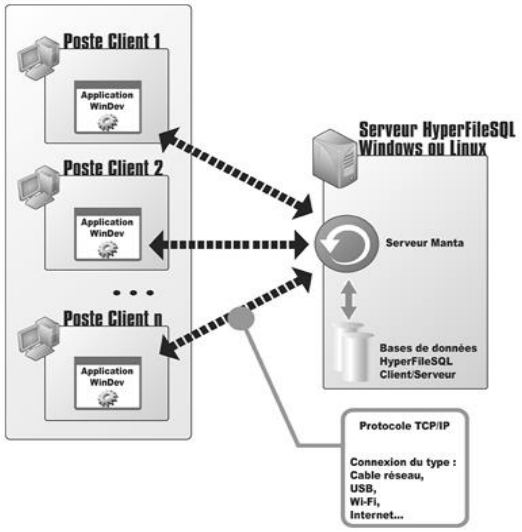


Figure 4.6 : L'architecture Client/serveur 2 tiers

Comme le montre la figure, notre système est équipé de :

- ✓ **Un serveur de gestion de base de données** comporte une importante capacité de stockage, doit être disponible afin qu'on puisse y accéder à tout moment, et doit avoir une puissante capacité de traitement dans le cas où plusieurs clients y accèdent en même temps.
- ✓ **Postes Clients :** sont des ordinateurs de bureau ou toutes sortes de machine ayant une possibilité de communiquer avec le serveur, et représentent les environnements d'exécution des applications.

2.1. Avantages de l'architecture client-serveur

- Toutes les données sont centralisées sur un seul serveur, ce qui simplifie les contrôles de sécurité, l'administration, la mise à jour des données et des logiciels.
- Les technologies supportant l'architecture client-serveur sont plus matures que les autres.
- La complexité du traitement et la puissance de calculs sont à la charge du ou des serveurs, les utilisateurs utilisant simplement un client léger sur un ordinateur terminal qui peut être simplifié au maximum.
- Recherche d'information : les serveurs étant centralisés, cette architecture est particulièrement adaptée pour retrouver et comparer de vaste quantité d'informations (moteur de recherche sur le Web), ce qui semble être rédhibitoire pour le P2P beaucoup plus lent, à l'image de Freenet.

2.2. Inconvénients de l'architecture client-serveur

- Si trop de clients veulent communiquer avec le serveur au même moment, ce dernier risque de ne pas supporter la charge (alors que les réseaux pair-à-pair fonctionnent mieux en ajoutant de nouveaux participants).
- Si le serveur n'est plus disponible, plus aucun des clients ne fonctionne (le réseau pair-à-pair continue à fonctionner, même si plusieurs participants quittent le réseau).
- Les coûts de mise en place et de maintenance peuvent être élevés.
- En aucun cas les clients ne peuvent communiquer entre eux, entrainant une asymétrie de l'information au profit des serveurs.

Dans une architecture deux tiers, encore appelée client/serveur de première génération ou client/serveur de données, le poste client se contente de déléguer la gestion des données à un service spécialisé. Le cas typique de cette architecture est une application de gestion fonctionnant sous Windows ou Linux et exploitant un SGBD centralisé.

Ce type d'application permet de tirer parti de la puissance des ordinateurs déployés en réseau pour fournir à l'utilisateur une interface riche, tout en garantissant la cohérence des données, qui restent gérées de façon centralisée. La gestion des données est prise en charge par un SGBD centralisé, s'exécutant le plus souvent sur un serveur dédié.

Ce dernier est interrogé en utilisant un langage de requête qui, le plus souvent, est SQL. Le dialogue entre client et serveur se résume donc à

l'envoi de requêtes et au retour des données correspondant aux requêtes.

3. IDENTIFICATION DES RISQUES ET PLANIFICATION DES ITERATIONS

Niveau recommande une relecture de l'étude de faisabilité opérationnelle pour identifier les taches à haut risque ou susceptible de retarder le bon déroulement de processus. Ici le maitre d'œuvre doit faire appel à son expertise dans la détection des taches a problème.

En tenant compte des différentes packages, il sera important que le maitre d'œuvre planifie l'ordre de développement des itérations en accord avec le maitre d'ouvrage.

CHAPITRE 6 : CONSTRUCTION DU SYSTEME

La construction du nouveau système d'information est une tâche complexe. Sa réalisation prend beaucoup de temps pour modéliser, concevoir, programmer et tester le système. Elle nécessite une analyse et une conception des données qui constituent le point de passage de toute application mettant en œuvre une base de données relationnelle.

La phase de construction du nouveau système prend en contact les aspects ci-après :

- La modélisation statique ;
- La modélisation dynamique ;
- Présentation d'environnement de développement et du SGBD ;
- Codification ;
- Présentation des interfaces et des structures des tables
- Test.

1. MODELISATION STATIQUE

La modélisation statique du système est une activité itérative, fortement couplée avec la modélisation dynamique. Pour les besoins de compréhension, nous avons présenté ces deux activités de façon séquentielle, mais dans la réalité elles sont effectuées quasiment en parallèle.

Dans cette partie, il sera question de s'occuper de la partie structurelle de système à mettre en place. L'accent est beaucoup plus porté sur les données.

1.1 DIAGRAMME DE CLASSES & DIAGRAMME D'OBJETS

Le diagramme de classes est considéré comme le plus important de la modélisation orientée objet, il est le seul obligatoire lors d'une telle modélisation.

Alors que le diagramme de cas d'utilisation montre un système du point de vue des acteurs, le diagramme de classes en montre la structure interne. Il permet de fournir une représentation abstraite des objets du système qui vont interagir ensemble pour réaliser les cas d'utilisation. Il est important de noter qu'un même objet peut très bien intervenir dans la réalisation de plusieurs cas d'utilisation. Les cas d'utilisation ne réalisent donc pas une partition des classes du diagramme de classes. Un diagramme de classes n'est donc pas adapté

(sauf cas particulier) pour détailler, décomposer, ou illustrer la réalisation d'un cas d'utilisation particulier.
Il s'agit d'une vue statique car on ne tient pas compte du facteur temporel dans le comportement du système. Le diagramme de classes modélise les concepts du domaine d'application ainsi que les concepts internes créés de toutes pièces dans le cadre de l'implémentation d'une application. Chaque langage de Programmation Orienté Objets donne un moyen spécifique d'implémenter le paradigme objet (pointeurs ou pas, héritage multiple ou pas, etc.), mais le diagramme de classes permet de modéliser les classes du système et leurs relations indépendamment d'un langage de programmation particulier.
La description du diagramme de classe est fondée sur :

- le concept d'objet,
- le concept de classe comprenant les attributs et les opérations,
- les différents types d'association entre classes.

1.1.1 Objet

Nous allons donner une première définition du concept d'objet avant de traiter le concept de classe. La description d'un objet sera complétée simultanément à la présentation du concept de classe.
Un objet est un concept, une abstraction ou une chose qui a un sens dans le contexte du système à modéliser. Chaque objet a une identité et peut être distingué des autres sans considérer a priori les valeurs de ses propriétés.

Exemple

La figure 2.1 montre des exemples d'objets physiques (une chaise, une voiture, une personne, un vélo) et d'objets de gestion (la Commande n° 12, le Client Durand).

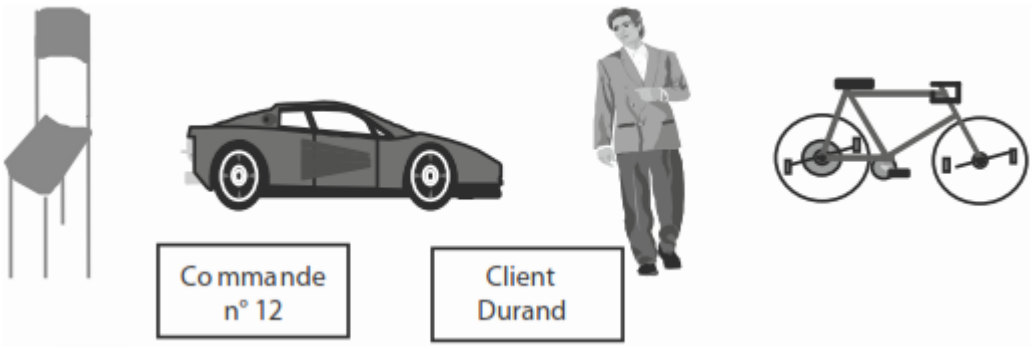


Figure 2.1 — Exemples d'objets physiques et d'objets de gestion

Autres caractéristiques

Un objet est caractérisé par les valeurs de ses propriétés qui lui confèrent des états significatifs suivant les instants considérés. Le formalisme de représentation d'un objet est donné après celui d'une classe.

1.1.2 Classe, attribut et opération

Classe

Une classe décrit un groupe d'objets ayant les mêmes propriétés (attributs), un même comportement (opérations), et une sémantique commune (domaine de définition).

Un objet est une instance d'une classe. La classe représente l'abstraction de ses objets. Au niveau de l'implémentation, c'est-à-dire au cours de l'exécution d'un programme, l'identificateur d'un objet correspond une adresse mémoire.

- *Formalisme général et exemple*

Une classe se représente à l'aide d'un rectangle comportant plusieurs compartiments.

Les trois compartiments de base sont :

- la désignation de la classe,
- la description des attributs,
- la description des opérations.

Deux autres compartiments peuvent être aussi indiqués :

- la description des responsabilités de la classe,
- la description des exceptions traitées par la classe.

Il est possible de manipuler les classes en limitant le niveau de description à un nombre réduit de compartiments selon les objectifs poursuivis par le modélisateur. Ainsi les situations suivantes sont possibles pour la manipulation d'une description restreinte de classe :

- description uniquement du nom et des caractéristiques générales de la classe,
- description du nom de la classe et de la liste d'attributs.

La figure 2.2 montre le formalisme général des compartiments d'une classe et des premiers exemples.

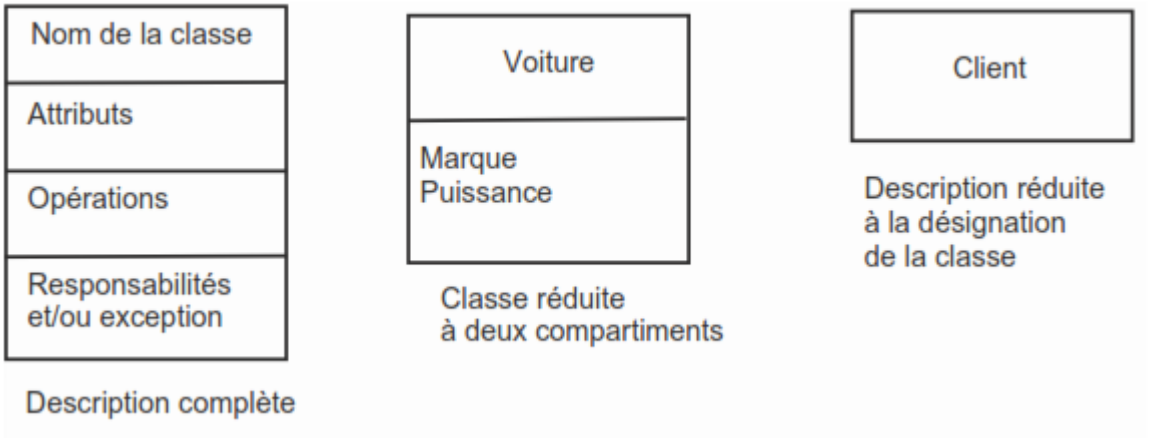


Figure 2.2 — Formalisme général d'une classe et exemples

- **Attribut**

Un attribut est une propriété élémentaire d'une classe. Pour chaque objet d'une classe, l'attribut prend une valeur (sauf cas d'attributs multivalués).

- ***Formalisme et exemple***

La figure 2.3 montre le formalisme et un exemple de représentation des attributs de classe.

Figure 2.3 — Formalisme d'attributs de classe et exemple

- ***Caractéristiques***

Le nom de la classe peut être qualifié par un « stéréotype ». La description complète des attributs d'une classe comporte un certain nombre de caractéristiques qui doivent respecter le formalisme suivant :

- *Visibilité/Nom attribut : type [= valeur initiale {propriétés}]*
    - *Visibilité* : se reporter aux explications données plus loin sur ce point.
    - *Nom d'attribut* : nom unique dans sa classe.
    - *Type* : type primitif (entier, chaîne de caractères...) dépendant des types disponibles dans le langage d'implémentation ou type classe matérialisant un lien avec une autre classe.
    - *Valeur initiale* : valeur facultative donnée à l'initialisation d'un objet de la classe.
    - *{propriétés}* : valeurs marquées facultatives (ex. : « interdit » pour mise à jour interdite).

Un attribut peut avoir des valeurs multiples. Dans ce cas, cette caractéristique est indiquée après le nom de l'attribut (ex. : prénom [3] pour une personne qui peut avoir trois prénoms).
Un attribut dont la valeur peut être calculée à partir d'autres attributs de la classe est un attribut dérivé qui se note « /nom de l'attribut dérivé ». Un exemple d'attribut dérivé est donné à la figure 2.5.

- *Opération*

Une opération est une fonction applicable aux objets d'une classe. Une opération permet de décrire le comportement d'un objet. Une méthode est l'implémentation d'une opération.

- *Formalisme et exemple*

Chaque opération est désignée soit seulement par son nom soit par son nom, sa liste de paramètres et son type de résultat. La signature d'une méthode correspond au nom de la méthode et la liste des paramètres en entrée. La figure 2.4 montre le formalisme et un exemple de représentation d'opérations de classe.

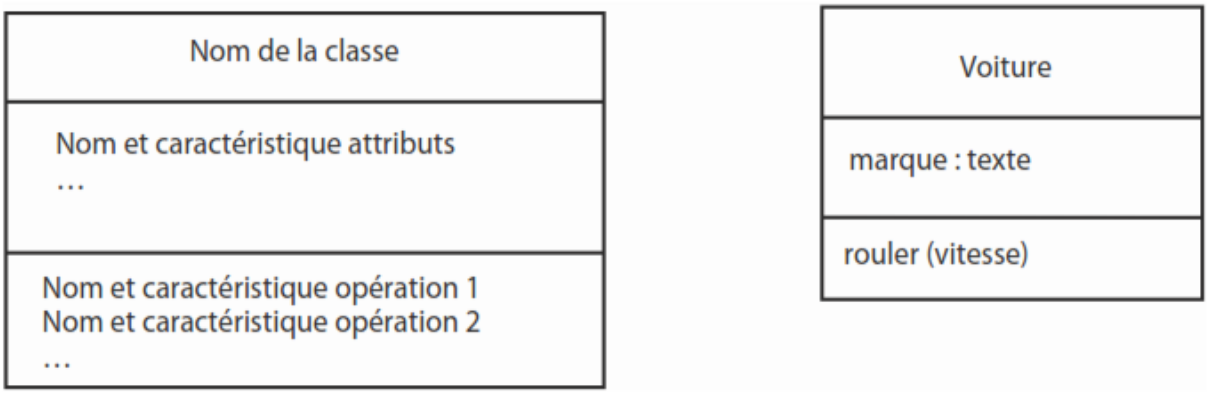


Figure 2.4 — Formalisme et exemple d'opérations de classe

- *Caractéristiques*

La description complète des opérations d'une classe comporte un certain nombre de caractéristiques qui doivent respecter le formalisme suivant :

- *Visibilité Nom d'opération (paramètres) [:[type résultat] {propriétés}]*
  - Visibilité : se reporter aux explications données plus loin sur ce point.
  - Nom d'opération : utiliser un verbe représentant l'action à réaliser.
  - Paramètres : liste de paramètres (chaque paramètre peut être décrit, en plus de son nom, par son type et sa valeur par défaut). L'absence de paramètre est indiquée par ( ).
  - Type résultat : type de (s) valeur(s) retourné(s) dépendant des types disponibles dans le langage d'implémentation. Par défaut, une opération ne retourne pas de valeur, ceci est indiqué par exemple par le mot réservé «void » dans le langage C++ ou Java.
  - {propriétés} : valeurs facultatives applicables (ex. : {query} pour un comportement sans influence sur l'état du système).

Exemples de classes et représentation d'objets

La figure 2.5 présente l'exemple d'une classe « Voiture ». La figure 2.6 donne le formalisme d'un objet.

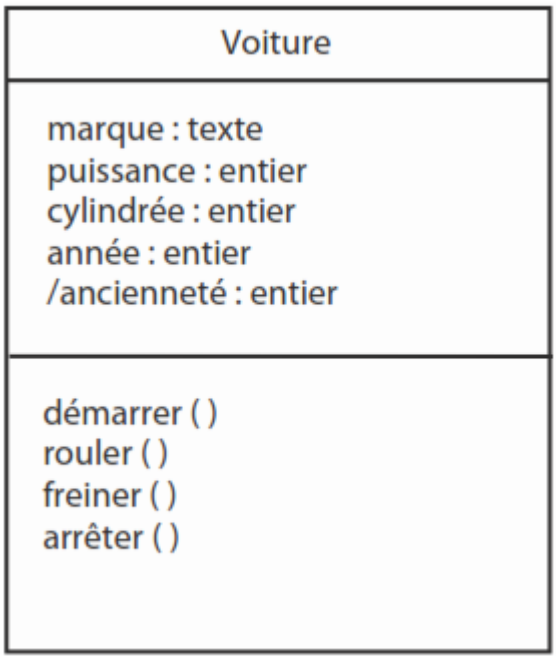


Figure 2.5 — Exemple de représentation d'une classe

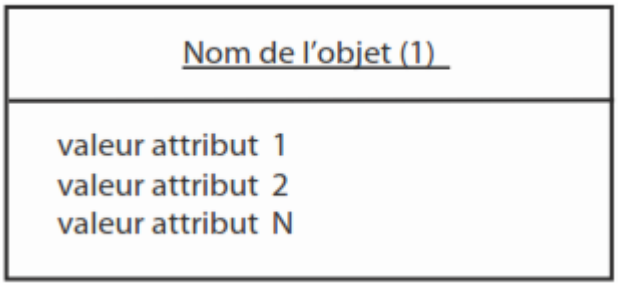
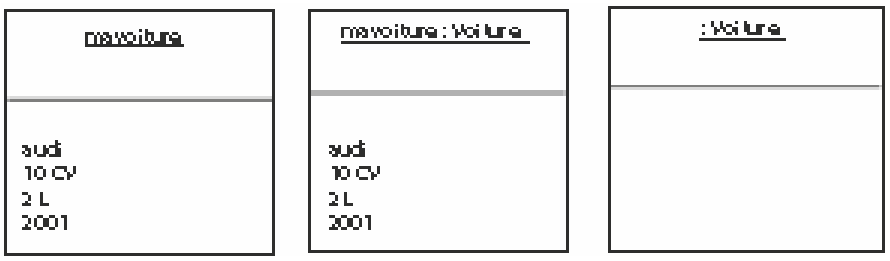


Figure 2.6 — Formalisme de représentation d'un objet

(1) Le nom d'un objet peut être désigné sous trois formes : nom de l'objet, désignation directe et explicite d'un objet ; nom de l'objet : nom de la classe, désignation incluant le nom de la classe ;
: nom de la classe, désignation anonyme d'un objet d'une classe donnée.

Il est utile de préciser que la représentation des objets sera utilisée dans plusieurs autres diagrammes importants d'UML. C'est le cas notamment du diagramme de séquence ou encore du diagramme d'état-transition.
La figure 2.7 présente des exemples d'objets.



- Visibilité des attributs et opérations

Chaque attribut ou opération d'une classe peut être de type public, protégé, privé ou paquetage. Les symboles + (public), # (protégé), - (privé) et ~ (paquetage) sont indiqués devant chaque attribut ou opération pour signifier le type de visibilité autorisé pour les autres classes.
Les droits associés à chaque niveau de confidentialité sont :

- Public (+) – Attribut ou opération visible par tous.
- Protégé (#) – Attribut ou opération visible seulement à l'intérieur de la classe et pour toutes les sous-classes de la classe.
- Privé (-) – Attribut ou opération seulement visible à l'intérieur de la classe.
- Paquetage (~) – Attribut ou opération ou classe seulement visible à l'intérieur du paquetage où se trouve la classe.

Exemple

La figure 2.8 montre un exemple d'utilisation des symboles de la visibilité des éléments d'une classe.
Dans cet exemple, tous les attributs sont déclarés de type privé, les opérations « démarrer » et « freiner » sont de type public, l'opération « rouler » est de type privé et l'opération « arrêter » est de type protégé.

- *Attribut ou opération de niveau classe*

Caractéristiques

Un attribut ou une opération peut être défini non pas au niveau des instances d'une classe, mais au niveau de la classe. Il s'agit soit d'un attribut qui est une constante pour toutes les instances d'une classe soit d'une opération d'une classe abstraite ou soit par exemple d'une opération « créer » qui peut être définie au niveau de la classe et applicable à la classe elle-même.

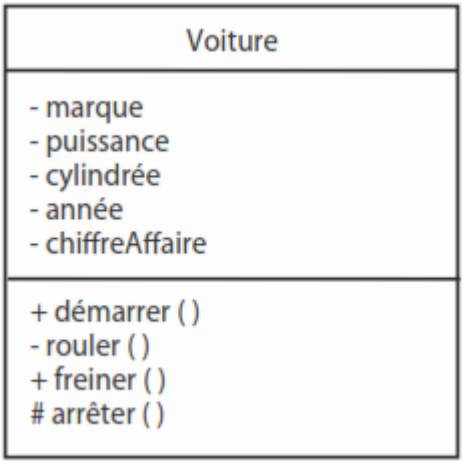


Figure 2.8 — Exemple de représentation des symboles de visibilité

- Formalisme et exemple

C'est le soulignement de l'attribut ou de l'opération qui caractérise cette propriété.
Dans l'exemple de la figure 2.9, l'attribut « ristourne » est de type classe et l'opération « créer » est une opération exécutable au niveau de la classe.

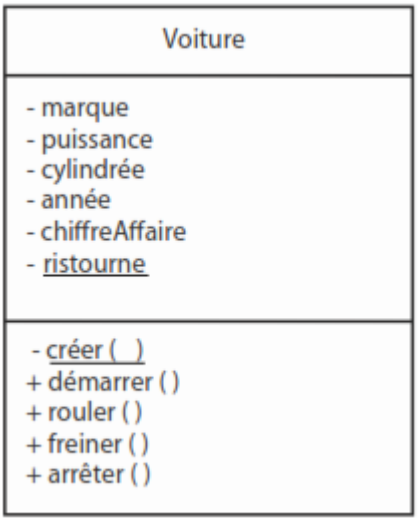


Figure 2.9 — Exemple d'attribut ou d'opération de niveau classe

1.1.3 Association, multiplicité, navigabilité et contraintes

- *Lien et association*

Un lien est une connexion physique ou conceptuelle entre instances de classes donc entre objets. Une association décrit un groupe de liens ayant une même structure et une même sémantique. Un lien est une instance d'une association. Chaque association peut être identifiée par son nom.
Une association entre classes représente les liens qui existent entre les instances de ces classes.

- *Formalisme et exemple*

La figure 2.10 donne le formalisme de l'association. Le symbole (facultatif) indique le sens de lecture de l'association. Dans cette figure est donné aussi un exemple de représentation d'une association.

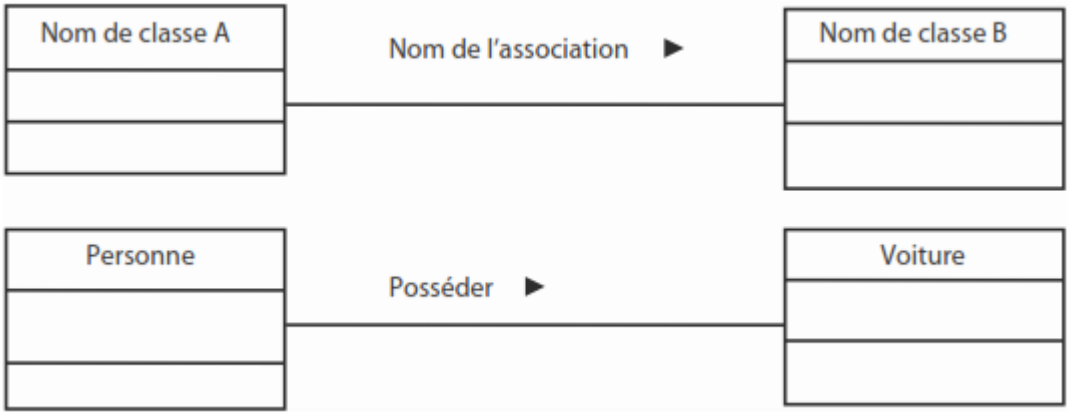


Figure 2.10 — Formalisme et exemple d'association

- Rôle d'association

Le rôle tenu par une classe vis-à-vis d'une association peut être précisé sur l'association.

Exemple

La figure 2.11 donne un exemple de rôle d'association.

Figure 2.11 — Exemple de rôles d'une association

- *Multiplicité*

La multiplicité indique un domaine de valeurs pour préciser le nombre d'instance d'une classe vis-à-vis d'une autre classe pour une association donnée. La multiplicité peut aussi être utilisée pour d'autres usages comme par exemple un attribut multivalué. Le domaine de valeurs est décrit selon plusieurs formes :

- Intervalle fermé – Exemple : 2, 3 ..15.
- Valeurs exactes – Exemple : 3, 5, 8.
- Valeur indéterminée notée * – Exemple : 1..*.

– Dans le cas où l'on utilise seulement *, cela traduit une multiplicité 0..*.

– Dans le cas de multiplicité d'associations, il faut indiquer les valeurs minimales et maximale d'instances d'une classe vis-à-vis d'une instance d'une autre classe.

- *Formalisme et exemple*

Nous donnons, à la figure 2.12, quelques exemples des principales multiplicités définies dans UML.

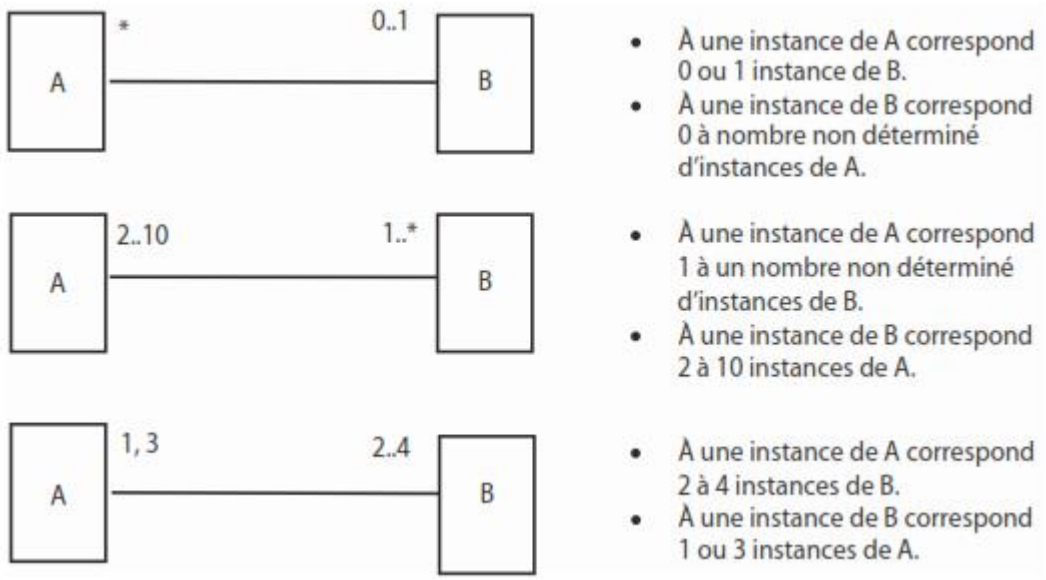


Figure 2.12 — Exemple de multiplicités

- *Navigabilité*

La navigabilité indique si l'association fonctionne de manière unidirectionnelle ou bidirectionnelle, elle est matérialisée par une ou deux extrémités fléchées. La non navigabilité se représente par un « X »

Les situations possibles de navigabilité sont représentées à la figure 2.13.

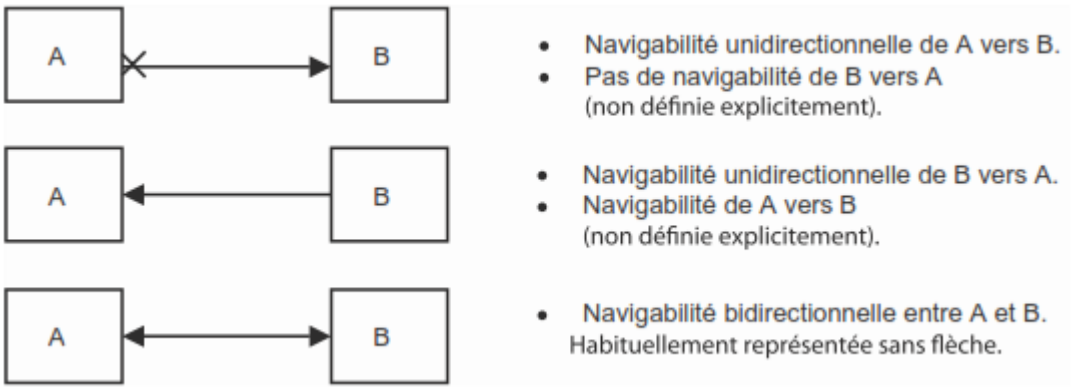


Figure 2.13 — Représentation de la navigabilité d'association

Par défaut, on admet qu'une navigabilité non définie correspond à une navigabilité implicite.

Dans l'exemple donné à la figure 2.14, à une personne sont associées ses copies d'examen mais l'inverse n'est pas possible (retrouver directement l'auteur de la copie d'examen, notamment avant la correction de la copie).

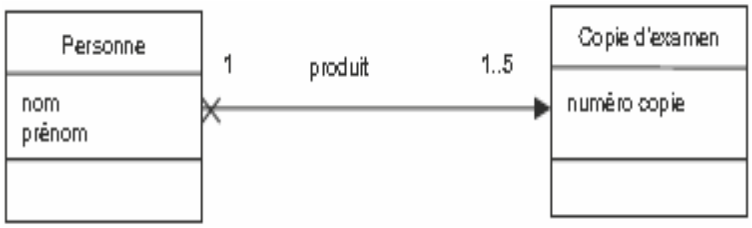


Figure 2.14 — Exemple de navigabilité d'une association

- **Contraintes**

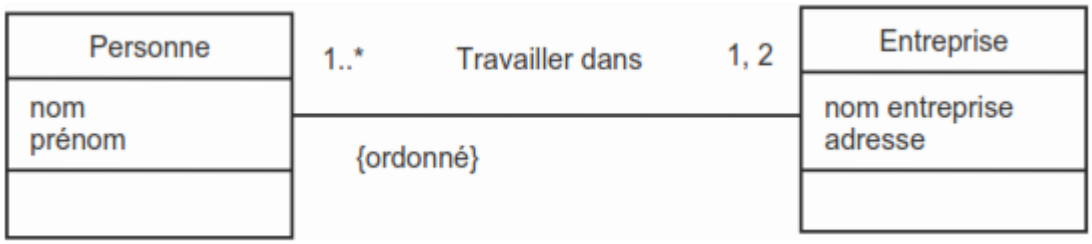
D'autres propriétés particulières (contraintes) sont proposées dans UML pour préciser la sémantique d'une association.

Ordre de tri

Pour une association de multiplicité supérieure à 1, les liens peuvent être :

- non ordonnés (valeur par défaut),
- ordonnés ou triés lorsque l'on est au niveau de l'implémentation (tri sur une valeur interne).

Un exemple est donné à la figure 2.15. Dans cet exemple, pour une entreprise donnée, les personnes seront enregistrées suivant un ordre qui correspondra à un des attributs de Personne.



Propriétés de mise à jour de liens

Il est possible d'indiquer des contraintes particulières relatives aux conditions de mise à jour des liens.

- {interdit} : interdit l'ajout, la suppression ou la mise à jour des liens.
- {ajout seul} : n'autorise que l'ajout de liens.

Association de dimension supérieure à 2 et classe-association

Une association de dimension supérieure à 2 se représente en utilisant un losange permettant de relier toutes les classes concernées.

Une classe-association permet de décrire soit des attributs soit des opérations propres à l'association. Cette classe-association est elle-même reliée par un trait en pointillé au losange de connexion. Une classe-association peut être reliée à d'autres classes d'un diagramme de classes.

Exemple

Un exemple d'une association de dimension 3 comprenant une classe-association « Affectation » est donné à la figure 2.16. La classe-association Affectation permet de décrire les attributs propres à l'association de dimension 3 représentée.

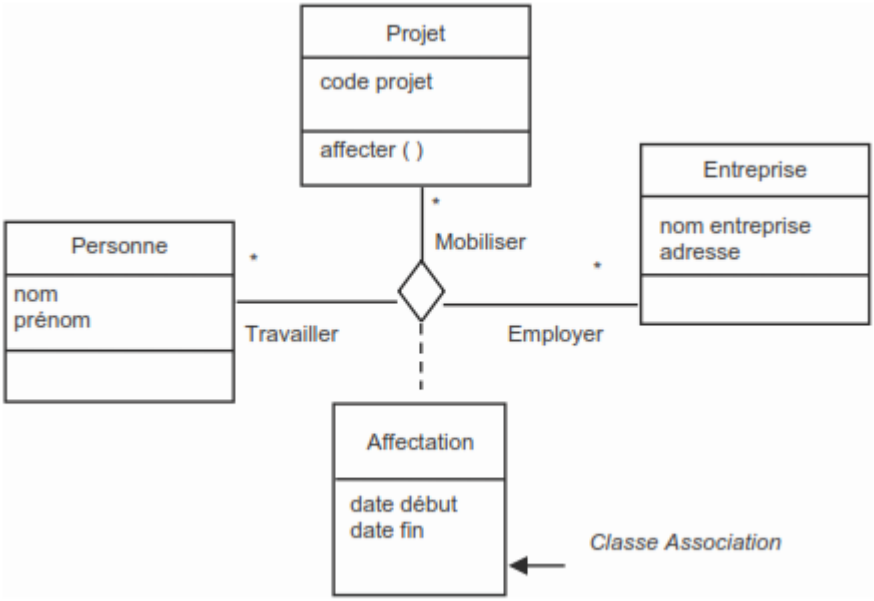


Figure 2.16 — Exemple d'une association de dimension 3 et d'une classe-association

1.1.4 Agrégation et composition entre classes

- *Agrégation*

L'agrégation est une association qui permet de représenter un lien de type « ensemble » comprenant des « éléments ». Il s'agit d'une relation entre une classe représentant le niveau « ensemble » et 1 à n classes de niveau « éléments ». L'agrégation représente un lien structurel entre une classe et une ou plusieurs autres classes.

- *Formalisme et exemple*

La figure 2.17 donne le formalisme général de l'agrégation.

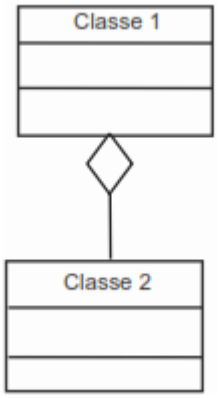


Figure 2.17 — Formalisme de l'agrégation

La figure 2.18 montre un exemple de relation d'agrégation. Dans cet exemple, nous avons modélisé le fait qu'un ordinateur comprend une UC, un clavier et un écran.

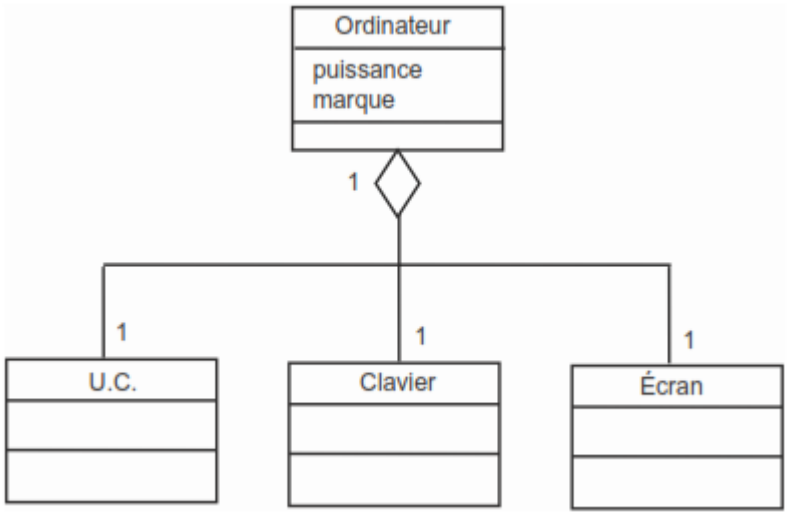


Figure 2.18 — Exemple d'agrégation

- **Composition**

La composition est une relation d'agrégation dans laquelle il existe une contrainte de durée de vie entre la classe « composant » et la ou les classes « composé ». Autrement dit la suppression de la classe « composé » implique la suppression de la ou des classes « composant ».

- ***Formalisme et exemple***

La figure 2.19 donne le formalisme général de la composition. La figure 2.20 montre un exemple de relation de composition. Une seconde forme de présentation peut être aussi utilisée, elle est illustrée à la figure 2.21.

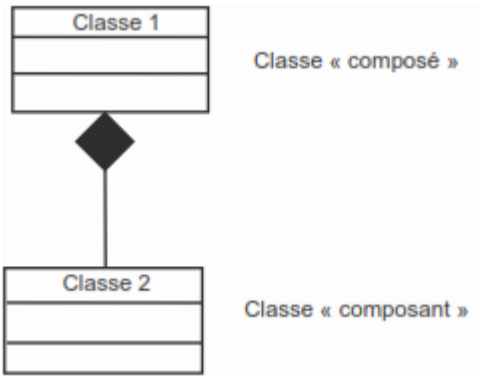


Figure 2.19 — Formalisme de la composition

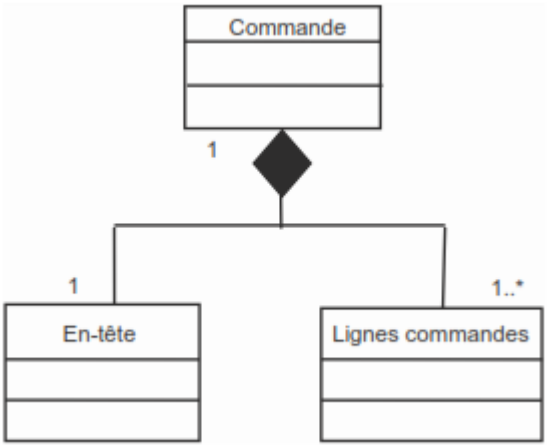


Figure 2.20 — Exemple d'une relation de composition

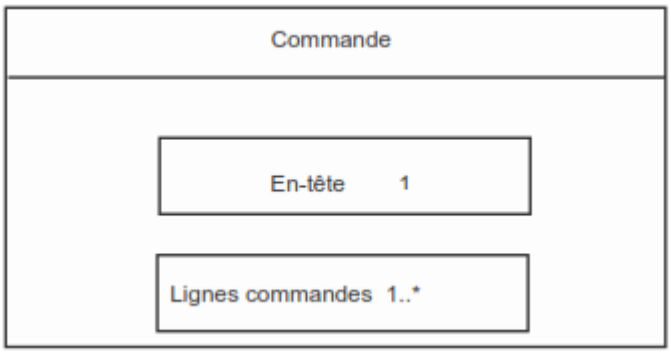


Figure 2.21 — Exemple de la seconde forme de représentation de la relation de composition

1.1.5 Association qualifiée, dépendance et classe d'interface

- *Qualification*

La qualification d'une relation entre deux classes permet de préciser la sémantique de l'association et de qualifier de manière restrictive les liens entre les instances.

Seules les instances possédant l'attribut indiqué dans la qualification sont concernées par l'association. Cet attribut ne fait pas partie de l'association.

- *Formalisme et exemple*

Soit la relation entre les répertoires et les fichiers appartenant à ces répertoires. À un répertoire est associé 0 à n fichiers. Si l'on veut restreindre cette association pour ne considérer qu'un fichier associé à son répertoire, la relation qualifiée est alors utilisée pour cela. La figure 2.22 montre la représentation de ces deux situations.

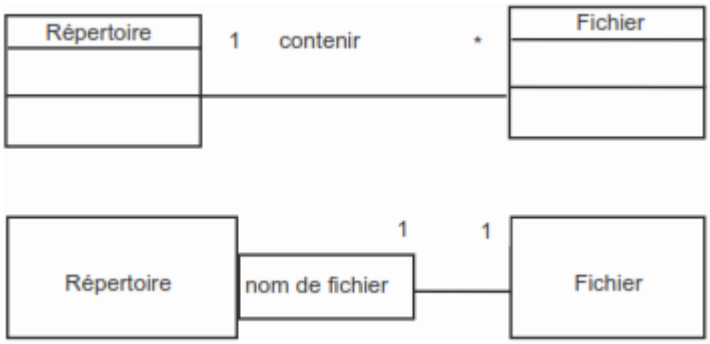


Figure 2.22 — Formalisme et exemple d'association qualifiée

- Dépendance

La dépendance entre deux classes permet de représenter l'existence d'un lien sémantique. Une classe B est en dépendance de la classe A si des éléments de la classe A sont nécessaires pour construire la classe B.

- *Formalisme et exemple*

La relation de dépendance se représente par une flèche en pointillé (fig. 2.23) entre deux classes.

Figure 2.23 — Formalisme de représentation d'un lien de dépendance

- Interface

Une classe d'interface permet de décrire la vue externe d'une classe. La classe d'interface, identifiée par un nom, comporte la liste des opérations accessibles par les autres classes. Le compartiment des attributs ne fait pas partie de la description d'une interface.

L'interface peut être aussi matérialisée plus globalement par un petit cercle associé à la classe source.

La classe utilisatrice de l'interface est reliée au symbole de l'interface par une flèche en pointillé. La classe d'interface est une spécification et non une classe réelle.

Une classe d'interface peut s'assimiler à une classe abstraite.

- *Formalisme et exemple*

La figure 2.24 donne le formalisme, sur un exemple, des deux types de représentation d'une interface.

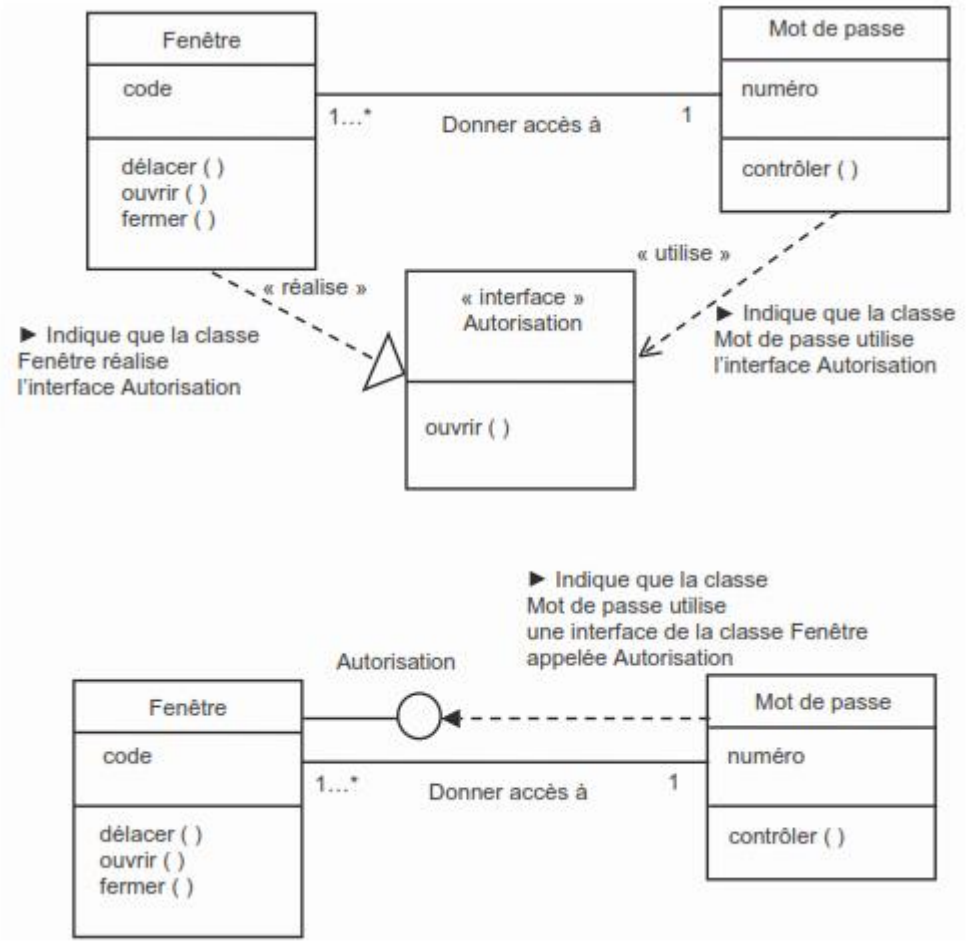


Figure 2.24 — Exemple de description d'une classe d'interface

1.1.6 Généralisation et spécialisation

- *La généralisation/spécialisation et l'héritage simple*

La généralisation est la relation entre une classe et deux autres classes ou plus partageant un sous-ensemble commun d'attributs et/ou d'opérations.

La classe qui est affinée s'appelle super-classe, les classes affinées s'appellent sous-classes. L'opération qui consiste à créer une super-classe à partir de

classes s'appelle la généralisation. Inversement la spécialisation consiste à créer des sous classes à partir d'une classe.

- *Formalisme et exemple*

La figure 2.25 montre le formalisme de la généralisation-spécialisation sous forme d'exemple général. Dans cet exemple :

- la sous-classe A1 hérite de A, c'est une spécialisation de A ;
- la sous-classe A2 hérite de A, c'est une spécialisation de A.

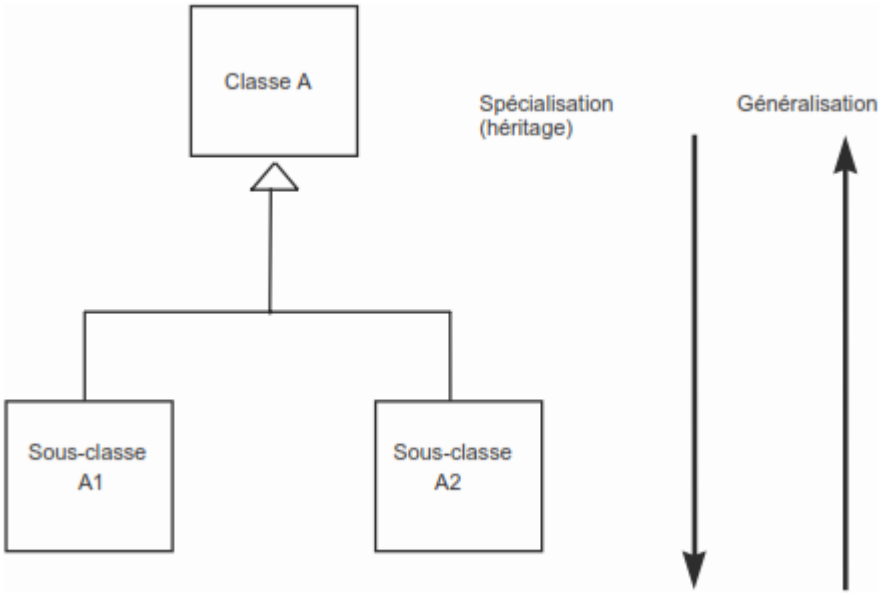


Figure 2.25 — Formalisme de la relation de généralisation

L'héritage permet à une sous-classe de disposer des attributs et opérations de la classe dont elle dépend. Un discriminant peut-être utilisé pour exploiter le critère de spécialisation entre une classe et ses sous-classes. Le discriminant est simplement indiqué sur le schéma, puisque les valeurs prises par ce discriminant correspondent à chaque sous-classe.

La figure 2.26 montre un exemple de relation de spécialisation. Dans cet exemple, les attributs nom, prénom et date de naissance et l'opération « calculer âge » de « Employé » sont hérités par les trois sous-classes : Employé horaire, Employé salarié, Vacataire.

- Classe abstraite

Une classe abstraite est une classe qui n'a pas d'instance directe mais dont les classes descendantes ont des instances. Dans une relation d'héritage, la super-classe est par définition une classe abstraite. C'est le cas de la classe Employé dans l'exemple présenté à la figure 2.26.

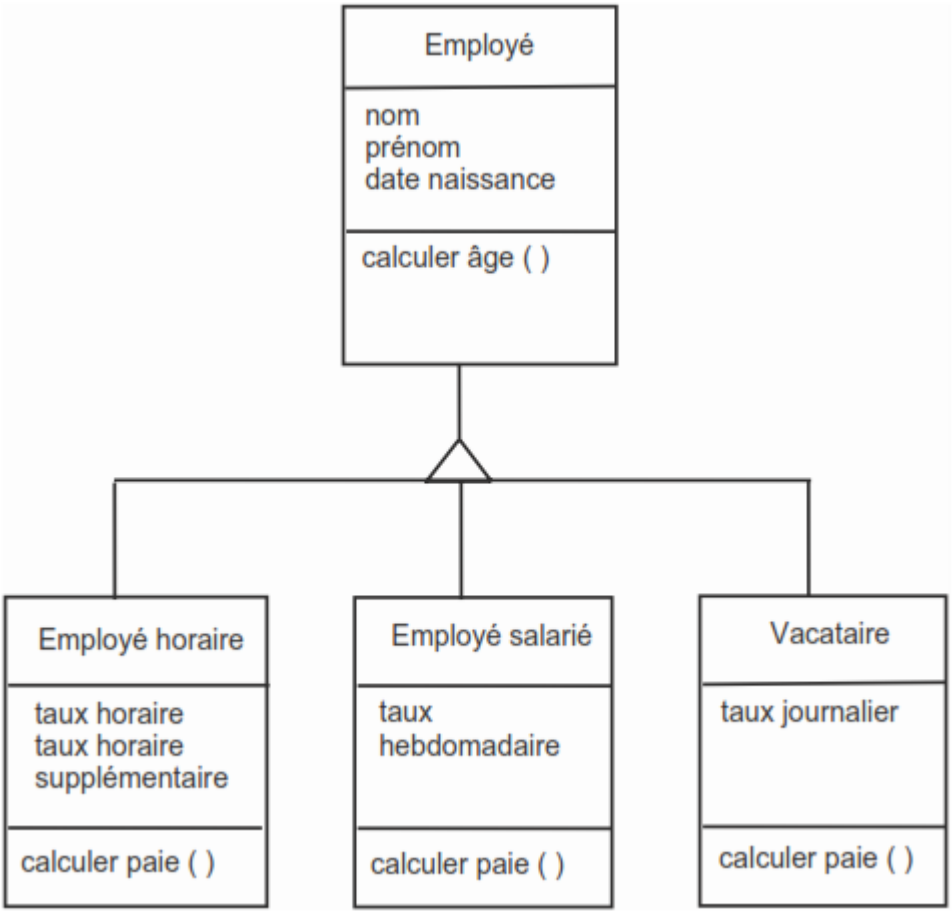


Figure 2.26 — Exemple de relation de spécialisation

- L'héritage avec recouvrement

Par défaut, les sous-classes ont des instances disjointes les unes par rapport aux autres. Dans certains cas, il existe un recouvrement d'instances entre les sous-classes.

D'une manière générale, quatre situations peuvent se rencontrer et se représentent sous forme de contraintes :

- {chevauchement} : deux sous-classes peuvent avoir, parmi leurs instances, des instances identiques ;
- {disjoint} : les instances d'une sous-classe ne peuvent être incluses dans une autre sous-classe de la même classe ;
- {complète} : la généralisation ne peut pas être étendue ;
- {incomplète} : la généralisation peut être étendue.

Dans certains cas, il est possible de ne pas citer toutes les sous-classes mais d'indiquer seulement des points de suspension (...).

- *Formalisme et exemple*

La figure 2.27 montre un exemple d'héritage avec recouvrement d'instances entre les classes Étudiant et Employé. En effet, une même personne peut être à la fois étudiante dans une université et employée dans une entreprise.

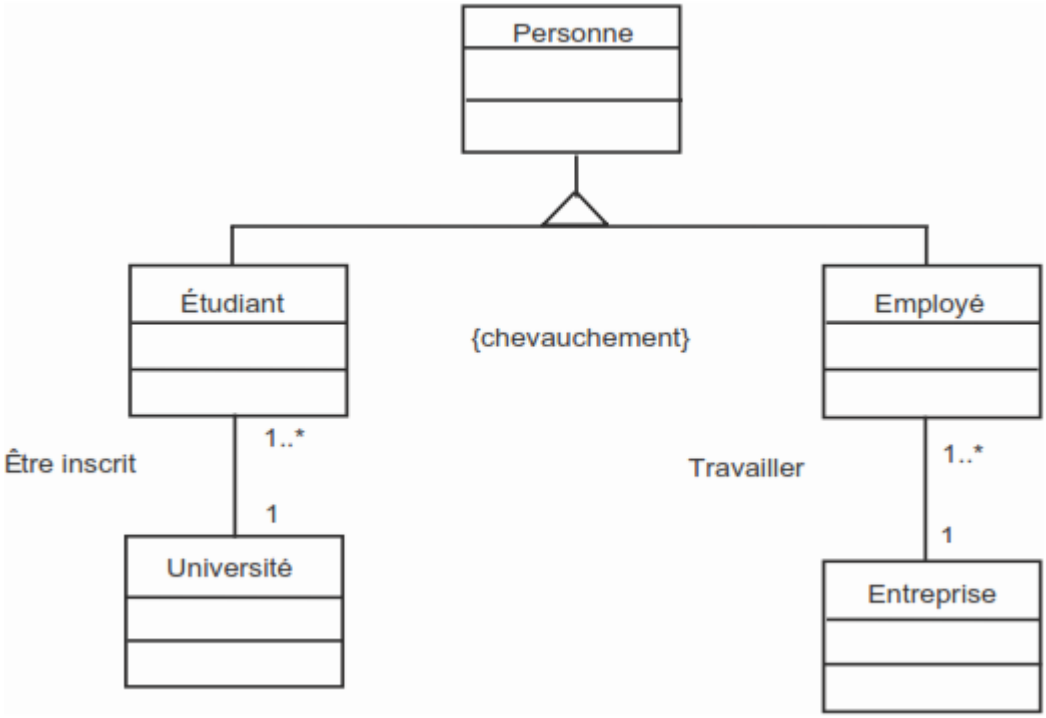


Figure 2.27 — Exemple d'héritage avec recouvrement d'instances

- Extension et restriction de classe

L'ajout de propriétés dans une sous-classe correspond à une extension de classe. Le masquage de propriétés dans une sous-classe correspond à une restriction de classe.

- *Formalisme et exemple*

La figure 2.28 montre un exemple d'héritage avec restriction et extension.

L'héritage multiple Dans certains cas, il est nécessaire de faire hériter une même classe de deux classes « parentes » distinctes. Ce cas correspond à un héritage multiple.

Exemple

La figure 2.29 montre un exemple classique d'héritage multiple où la classe « Véhicule amphibie » hérite des classes « Véhicule terrestre » et « Véhicule marin ».

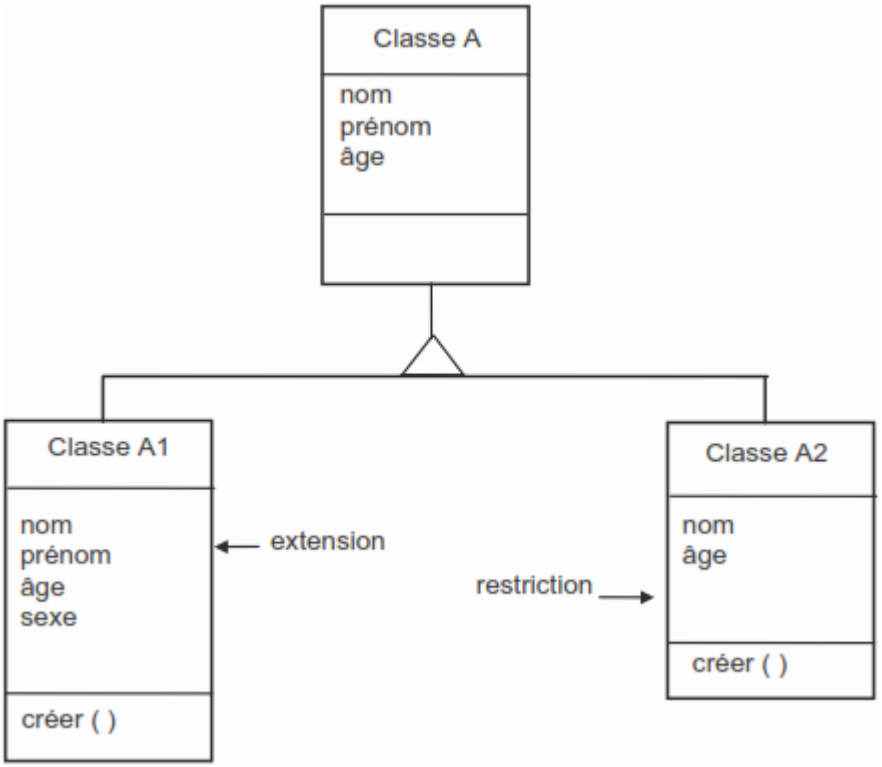


Figure 2.28 — Exemple d'héritage avec extension et restriction de propriétés

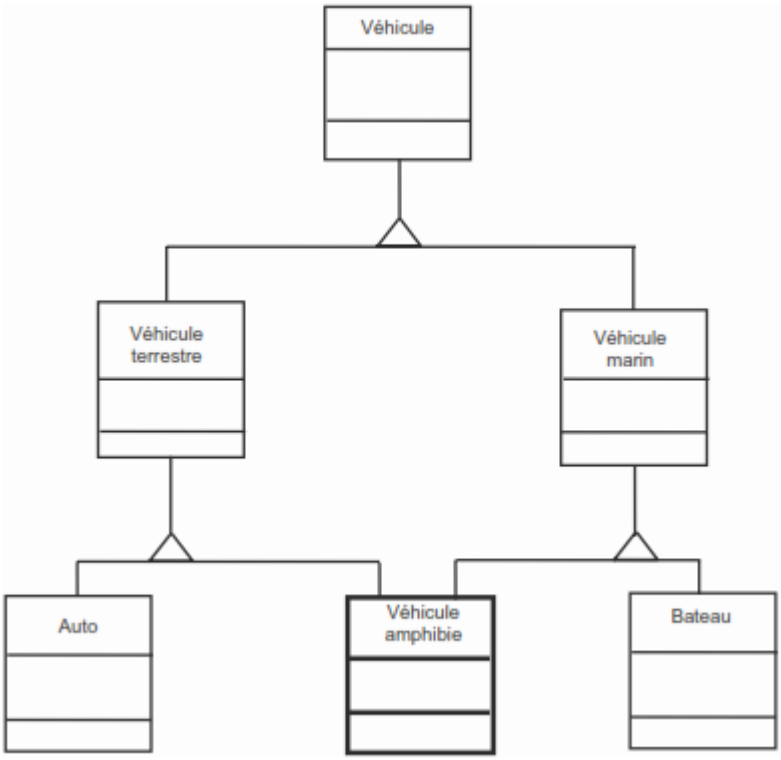


Figure 2.29 — Exemple de relation d'héritage multiple

1.1.7 Stéréotype de classe

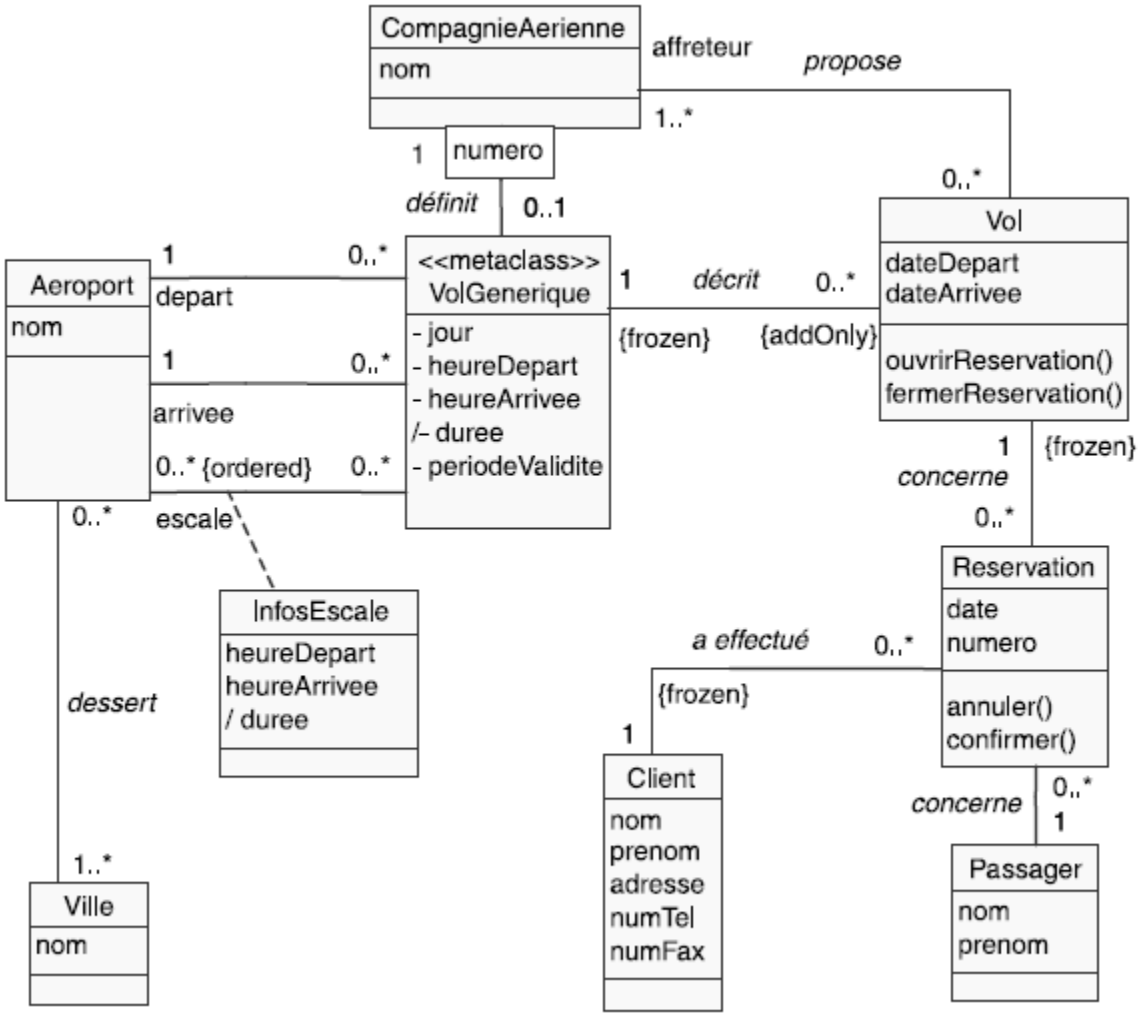
UML propose un certain nombre de stéréotypes qui permettent de qualifier les profils d'utilisation.

Parmi ces stéréotypes, nous présentons ci-après quatre d'entre eux :

- « Classe d'implémentation » – Ce stéréotype est utilisé pour décrire des classes de niveau physique.
- « Type » – Ce stéréotype permet de spécifier des opérations applicables à un domaine d'objets. Exemple : Type Integer d'un langage de programmation.
- «Utilitaire» – Ce stéréotype qualifie toutes les fonctions utilitaires de base utilisées par les objets.
- « MétaClasse » – Ce stéréotype permet de regrouper des classes dans une famille de classe.

Exemple :

Pour une compagnie aérienne, on peut avoir le diagramme de classe suivant :



1.2 REGLES DE PASSAGE D'UN DIAGRAMME DE CLASSE VERS UN MODELE RELATIONNEL

Les règles de passages ci-après s'appliquent lors de passage d'un diagramme de classes vers un modèle relationnel.

1.2.1 Classe avec attributs

Chaque classe sera représentée par une table dont les colonnes sont les attributs de cette classe. Si la classe possède un identifiant, il devient la clé primaire de la classe, sinon, il faut ajouter une clé primaire arbitraire.

1.2.2 Une association 1 vers 1

Pour représenter une association 1 vers 1 entre deux classes, la clé primaire de l'une des classes doit figurer comme clé étrangère dans l'autre.

1.2.3 Une association 1 vers plusieurs

Pour représenter une association 1 vers plusieurs, on procède comme pour une association 1 vers 1, excepté que c'est forcément la relation du côté plusieurs qui reçoit comme clé étrangère la clé primaire de la relation du côté 1.

1.2.4 Association plusieurs vers plusieurs

Pour représenter une association du type plusieurs vers plusieurs, il faut introduire une nouvelle relation dont les attributs sont les clés primaires des classes en association et dont la clé primaire est la concaténation de ces deux attributs.

1.2.5 Classe Association plusieurs vers plusieurs

Le cas est proche de celui d'une association plusieurs vers plusieurs, les attributs de la classe-association étant ajoutés à la troisième relation qui représente, cette fois-ci, la classe-association elle-même.

1.2.6 Pour la généralisation (héritage)

Les relations correspondant aux sous-classes ont comme clés étrangère et primaire la clé de la relation correspondant à la classe parente. Un attribut type est ajouté dans la relation correspondant à la classe parente. Cet attribut permet de savoir si les informations d'un t-uple de la relation

correspondant à la classe parente peuvent être complétées par un t-uple de l'une des relations correspondant à une sous-classe, et, le cas échéant, de quelle relation il s'agit. Ainsi, dans cette solution, un objet peut avoir ses attributs répartis dans plusieurs relations. Il faut donc opérer des jointures pour reconstituer un objet. L'attribut type de la relation correspondant à la classe parente doit indiquer quelles jointures faire.

Exemple d'un schéma relationnel de DCL

Soit le diagramme de classes ci-après pour une application de gestion de patrimoine d'une entreprise de la place.

Exemple d'un schéma relationnel de DCL

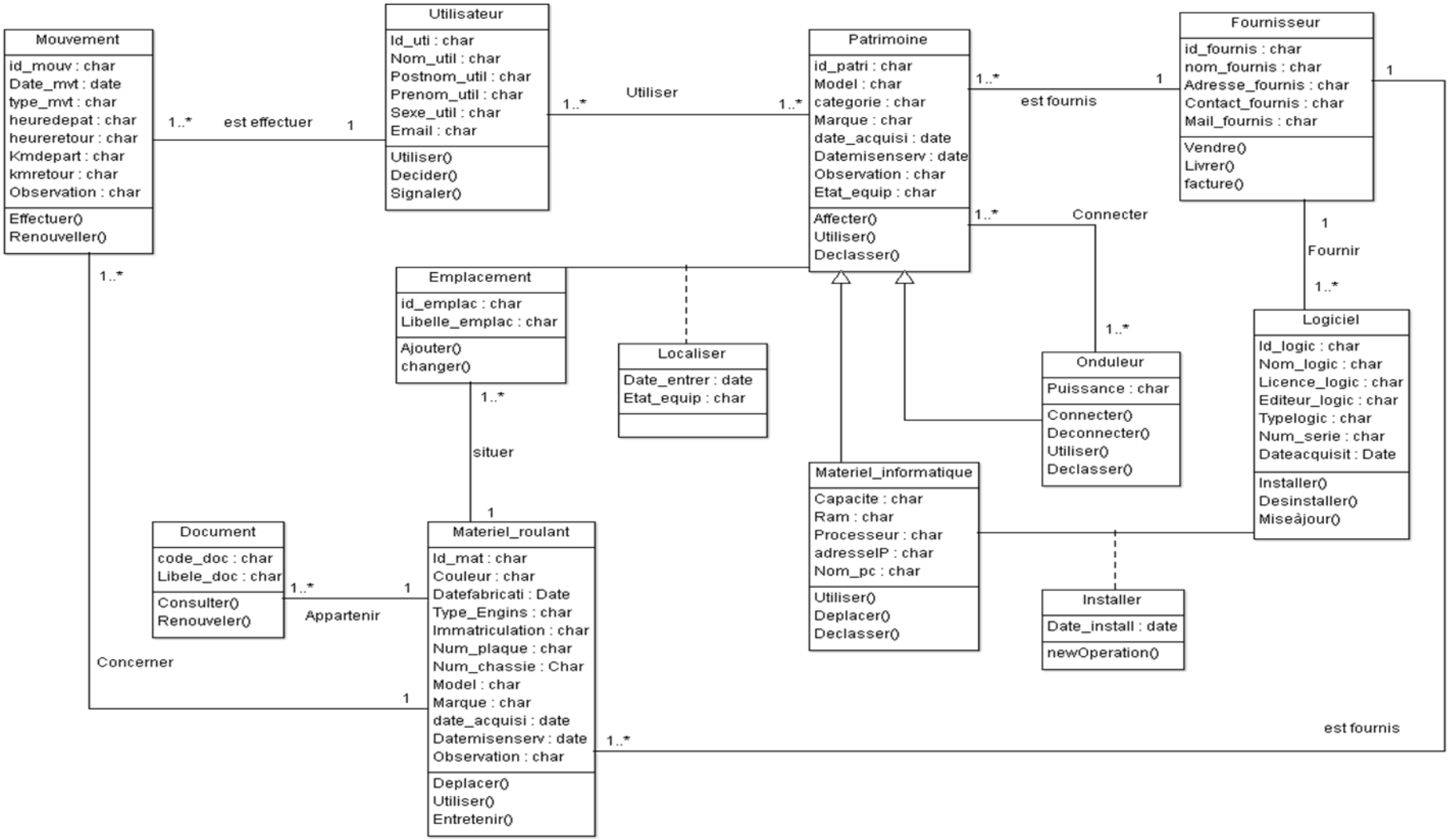
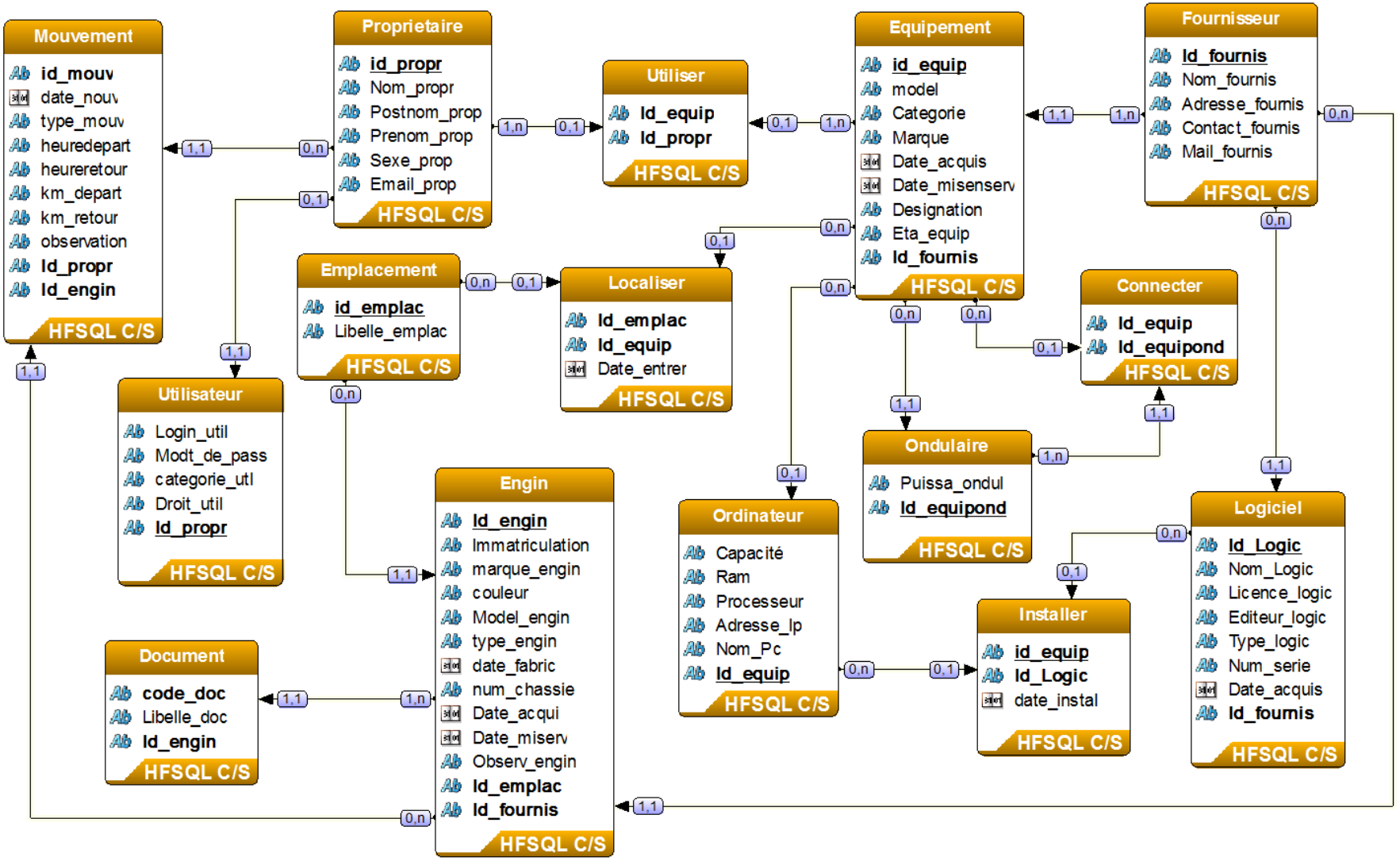


Schéma relationnel



1.2 DIAGRAMME DE COMPOSANT (DCP)

Le diagramme de composant permet de représenter les composants logiciels d'un système ainsi que les liens existant entre ces composants.
Les composants logiciels peuvent être de deux origines : soit des composants métiers propres à une entreprise soit des composants disponibles.

1.2.1 Composant

Chaque composant est assimilé à un élément exécutable du système. Il est caractérisé par :

- un nom ;
- une spécification externe sous forme soit d'une ou plusieurs interfaces requises, soit d'une ou plusieurs interfaces fournies ;
- un port de connexion.

Le port d'un composant représente le point de connexion entre le composant et une interface. L'identification d'un port permet d'assurer une certaine indépendance entre le composant et son environnement extérieur.

- *Formalisme général*

Un composant est représenté (fig. 2.36) par un classeur avec le mot-clé « composant » ou bien par un classeur comportant une icône représentant un module.

Figure 2.36 — Formalisme général d'un composant

1.2.2 Les deux types de représentation et exemples

Deux types de représentation sont disponibles pour modéliser les composants : une représentation « boîte noire » et une représentation « boîte blanche ». Pour chaque représentation, plusieurs modélisations des composants sont proposées.

Représentation « boîte noire »

C'est une vue externe du composant qui présente ses interfaces fournies et requises sans entrer dans le détail de l'implémentation du composant. Une boîte noire peut se représenter de différentes manières.

Connecteur d'assemblage

Une interface fournie se représente à l'aide d'un trait et d'un petit cercle et une interface requise à l'aide d'un trait et d'un demi-cercle. Ce sont les connecteurs d'assemblage.

Un exemple de modélisation avec les connecteurs d'assemblage est donné à la figure 2.37, le composant Commande possède deux interfaces fournies et deux interfaces requises.

Figure 2.37 — Représentation d'un connecteur d'assemblage

Connecteur d'interfaces

Une autre représentation peut être aussi utilisée en ayant recours aux dépendances d'interfaces utilise et réalise :

- pour une interface fournie, c'est une relation de réalisation partant du composant et allant vers l'interface ;
- pour une interface requise, c'est une dépendance avec le mot-clé « utilise » partant du composant et allant vers l'interface.

Un exemple de modélisation avec connecteurs d'interfaces est donné à la figure 2.38.

Le composant Commande possède une interface fournie « GestionCommande » et une interface requise « Produit ».

Figure 2.38 — Représentation d'un composant avec connecteur d'interfaces

Compartiment

Une dernière manière de modéliser un composant avec une représentation boîte noire est de décrire sous forme textuelle les interfaces fournies et requises à l'intérieur d'un second compartiment.

Un exemple de modélisation avec les compartiments est donné à la figure 2.39.

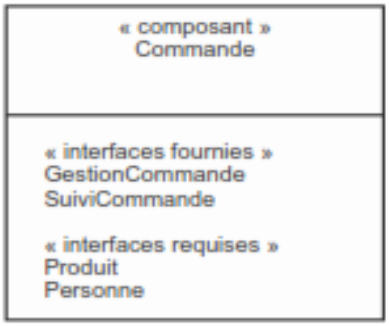


Figure 2.39 — Représentation d'un composant avec compartiments

- *Représentation « boîte blanche »*

C'est une vue interne du composant qui décrit son implémentation à l'aide de classificateurs (classes, autres composants) qui le composent. Plusieurs modélisations sont possibles pour la représentation boîte blanche.

- *Compartiment*

Une manière de modéliser un composant avec une représentation boîte blanche est de décrire sous forme textuelle les interfaces fournies et requises à l'intérieur d'un compartiment, les classificateurs (classes, autres composants) dans un autre compartiment, les artefacts qui représentent physiquement le composant dans un dernier compartiment.

Un exemple de modélisation avec les compartiments est donné à la figure 2.40.

Figure 2.40 — Représentation boîte blanche avec compartiments

Dépendance

Une autre représentation interne du composant peut être aussi utilisée en ayant recours aux dépendances. Ainsi, les classificateurs qui composent le composant sont reliés à celui-ci par une relation de dépendance.

Les relations entre les classificateurs (association, composition, agrégation) sont aussi modélisées. Néanmoins, si elles sont trop complexes, elles peuvent être représentées sur un diagramme de classe relié au composant par une note.

Un exemple de modélisation boîte blanche avec les dépendances est donné à la figure 2.41.

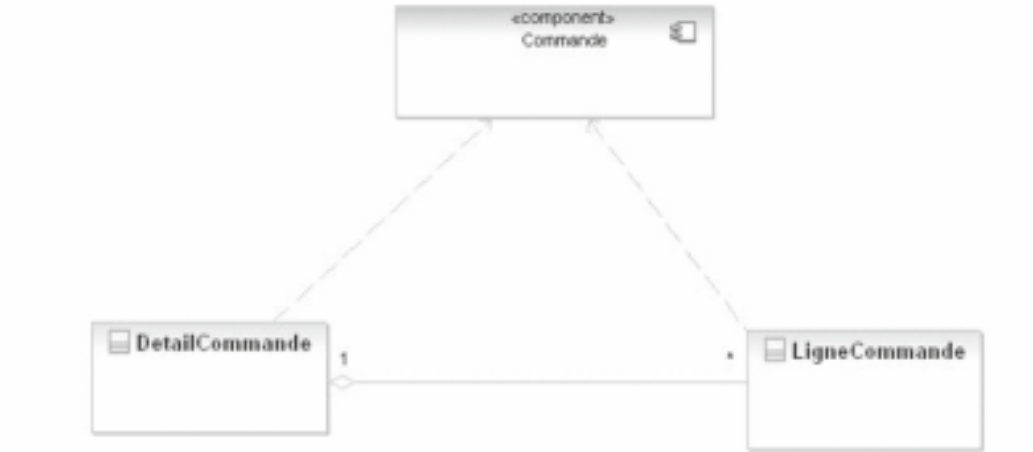


Figure 2.41 — Représentation boîte blanche avec dépendances

Ports et connecteurs

Le port est représenté par un petit carré sur le composant. Les connecteurs permettent de relier les ports aux classificateurs. Ils sont représentés par une association navigable et indiquent que toute information arrivée au port est transmise au classificateur.

Un exemple de représentation avec port et connecteur du composant Commande est donné à la figure 2.42.

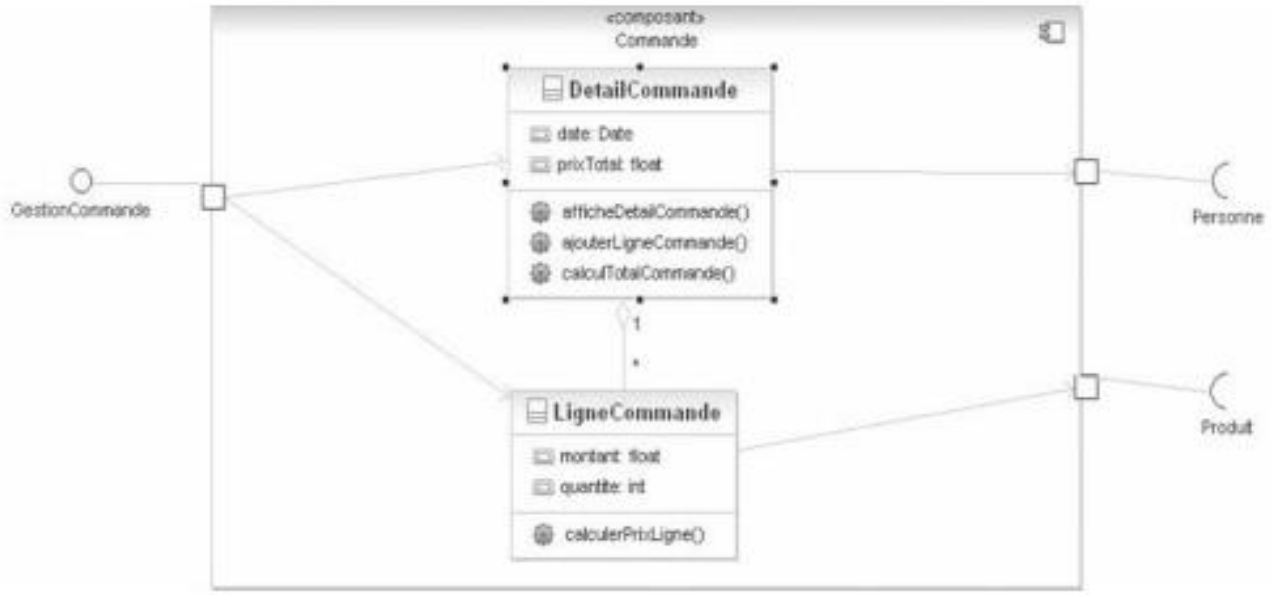


Figure 2.42 — Représentation boîte blanche avec connecteurs

Dans l'exemple de la figure 2.42, le composant Commande est constitué de deux classes (classificateur) reliées par une agrégation : DetailCommande et LigneCommande.

L'interface fournie GestionCommande est accessible de l'extérieur via un port et permet d'accéder via les connecteurs aux opérations des deux classes DetailCommande et LigneCommande (ajouterLigneCommande, calculTotal-Commande, calculPrixLigne).

L'interface requise Personne est nécessaire pour l'affichage du détail de la commande et est accessible via un port du composant Commande.

L'interface requise Produit est nécessaire pour le calcul du prix de la ligne de commande et est accessible via un port du composant Commande.

2. MODELISATION DYNAMIQUE

La modélisation dynamique s'occupe de la partie comportementale du système où l'on pense à des aspects liés à la manière dont les différents traitements seront effectués. Cette partie peut faire recours aux reste de diagrammes comportementaux selon le degré de précision à apporter au système à mettre en place. Ici le diagramme d'activité joue un rôle prépondérant.

2.1 DIAGRAMME D'ACTIVITÉ (DAC)

2.1.1 Présentation générale et concepts de base

Le diagramme d'activité présente un certain nombre de points communs avec le diagramme d'état-transition puisqu'il concerne le comportement interne des opérations ou des cas d'utilisation. Cependant le comportement visé ici s'applique aux flots de contrôle et aux flots de données propres à un ensemble d'activités et non plus relativement à une seule classe.

Les concepts communs ou très proches entre le diagramme d'activité et le diagramme d'état-transition sont :

- transition,
- ● nœud initial (état initial),
- ⊙ nœud final (état final),
- ⊗ nœud de fin flot (état de sortie),
- ◊ nœud de décision (choix).

Le formalisme reste identique pour ces nœuds de contrôle.

Les concepts spécifiques au diagramme d'activité sont :

- nœud de bifurcation,
- nœud de jonction,
- nœud de fusion,
- pin d'entrée et de sortie,
- flot d'objet,
- partition.

- *Action*

Une action correspond à un traitement qui modifie l'état du système. Cette action peut être appréhendée soit à un niveau élémentaire proche d'une instruction en termes de programmation soit à un niveau plus global correspondant à une ou plusieurs opérations.

- *Formalisme et exemple*

Une action est représentée par un rectangle dont les coins sont arrondis comme pour les états du diagramme d'état-transition (fig. 3.23).

Figure 3.23 — Formalisme et exemple d'une action

- *Transition et flot de contrôle*

Dès qu'une action est achevée, une transition automatique est déclenchée vers l'action suivante. Il n'y a donc pas d'événement associé à la transition.

L'enchaînement des actions constitue le flot de contrôle.

- Formalisme et exemple

Le formalisme de représentation d'une transition est donné à la figure 3.24.

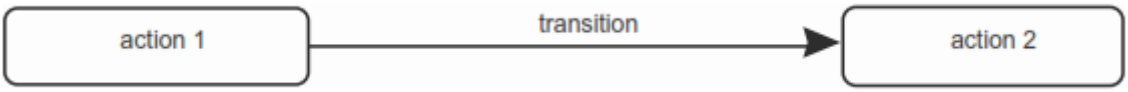


Figure 3.24 — Formalisme de base du diagramme d'activité : actions et transition

- *Activité*

Une activité représente le comportement d'une partie du système en termes d'actions et de transitions. Une activité est composée de trois types de nœuds :

- nœud d'exécution (action, transition),
- nœud de contrôle (nœud initial, nœud final, flux de sortie, nœud de bifurcation, nœud de jonction, nœud de fusion-test, nœud de test-décision, pin d'entrée et de sortie),
- nœud d'objet.

Une activité peut recevoir des paramètres en entrée et en produire en sortie.

- *Formalisme et exemple*

Nous donnons une première représentation simple à la figure 3.25.

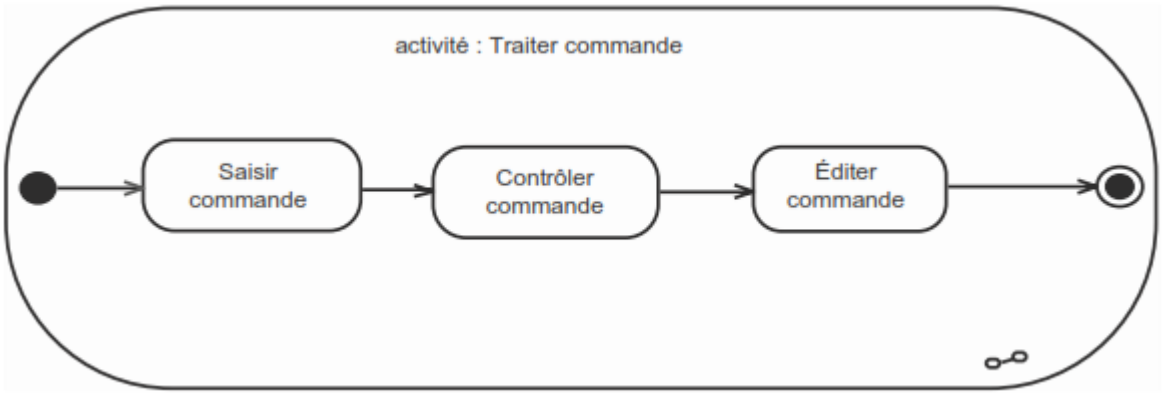


Figure 3.25 — Exemple de représentation d'une activité

- *Nœud de bifurcation (fourche)*

Un nœud de bifurcation (fourche) permet à partir d'un flot unique entrant de créer plusieurs flots concurrents en sortie de la barre de synchronisation.

- *Formalisme et exemple*

Le formalisme de représentation de nœud de bifurcation ainsi qu'un premier exemple sont donnés à la figure 3.26. Un second exemple avec nœud de bifurcation est donné à la figure 3.27.

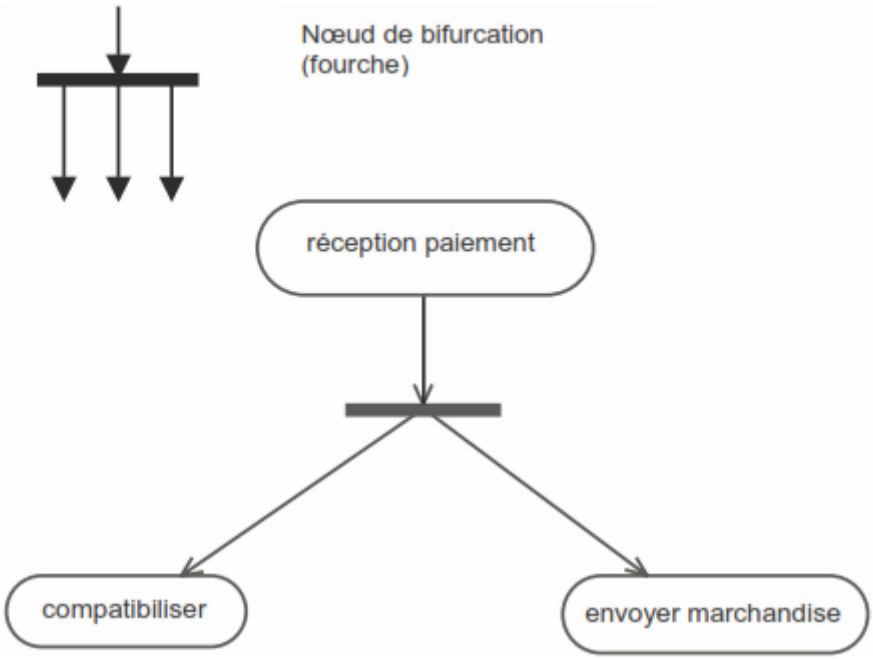


Figure 3.26 — Exemple 1 d'activités avec nœud de bifurcation

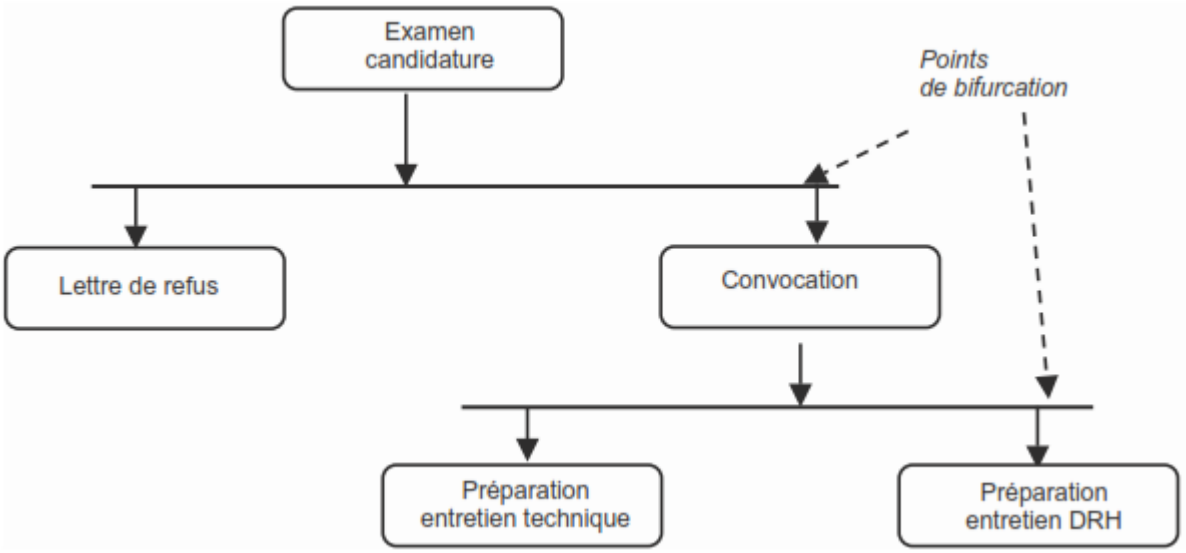


Figure 3.27 — Exemple 2 de diagramme d'activité avec bifurcation de flots de contrôle

- *Nœud de jonction (synchronisation)*

Un nœud de jonction (synchronisation) permet, à partir de plusieurs flots concurrents en entrée de la synchronisation, de produire un flot unique sortant. Le nœud de jonction est le symétrique du nœud de bifurcation.

- *Formalisme et exemple*

Le formalisme de représentation d'un nœud de jonction est donné à la figure 3.28.

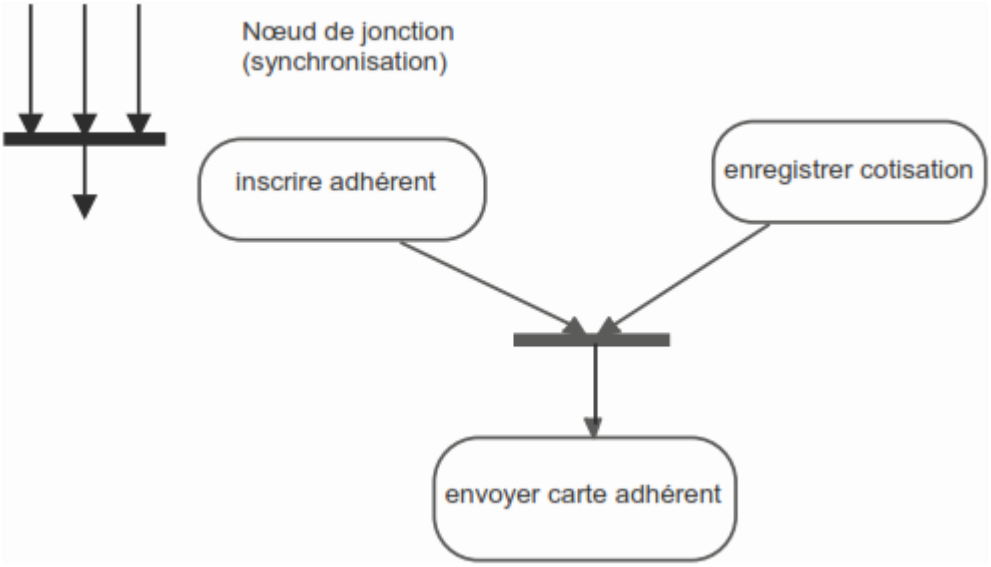


Figure 3.28 — Exemple d'activités avec nœud de jonction

- *Nœud de test-décision*

Un nœud de test-décision permet de faire un choix entre plusieurs flots sortants en fonction des conditions de garde de chaque flot. Un nœud de test-décision n'a qu'un seul flot en entrée. On peut aussi utiliser seulement deux flots de sortie : le premier correspondant à la condition vérifiée et l'autre traitant le cas sinon.

- *Formalisme et exemple*

Le formalisme de représentation d'un nœud de test-décision ainsi qu'un premier exemple sont donnés à la figure 3.29. Un second exemple avec nœud de test-décision est donné à la figure 3.30.

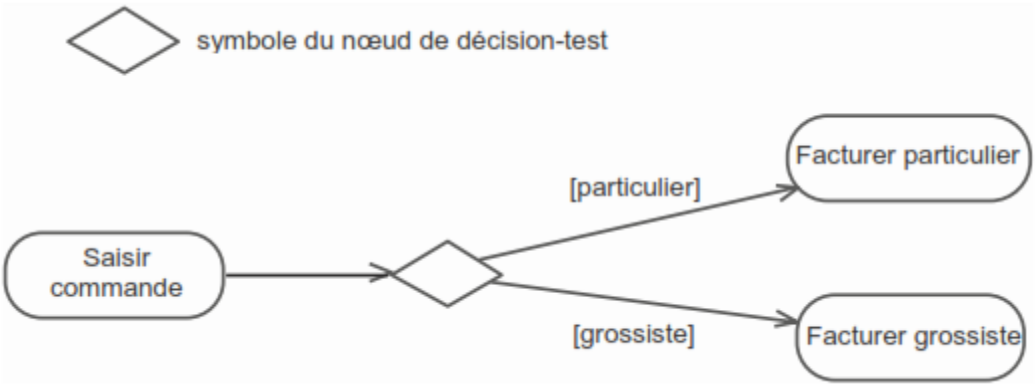


Figure 3.29 — Formalisme et exemple 1 d'activités avec nœud de test-décision

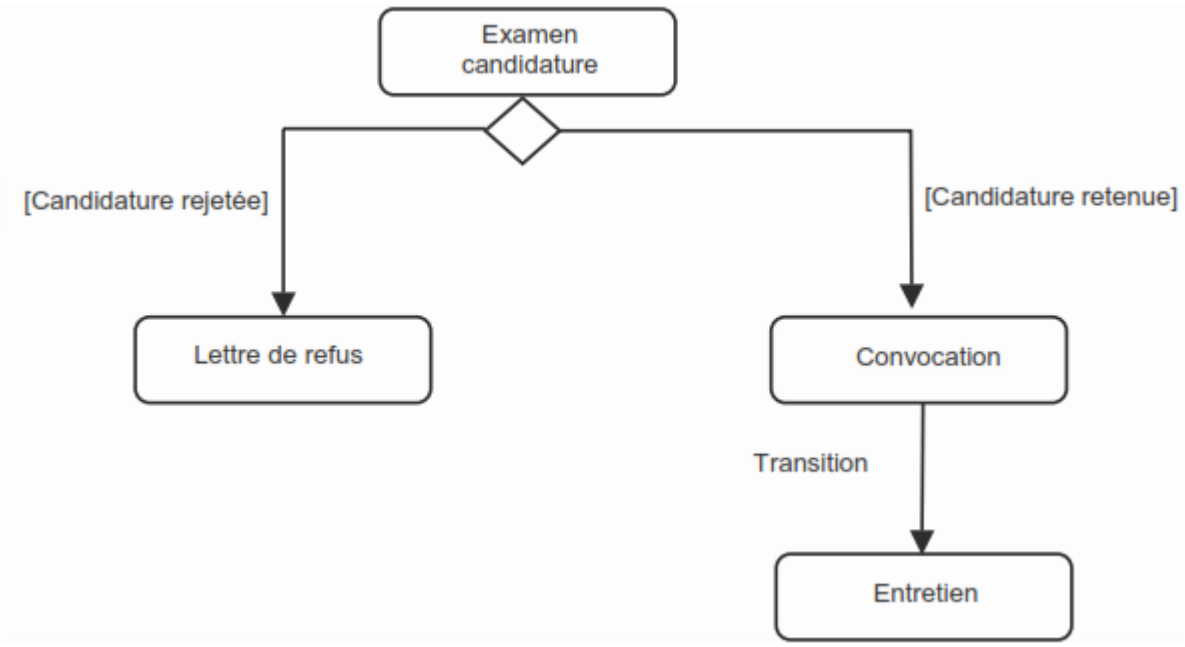


Figure 3.30 — Exemple 2 de diagramme d'activités avec un nœud de test-décision

- *Nœud de fusion-test*

Un nœud de fusion-test permet d'avoir plusieurs flots entrants possibles et un seul flot sortant. Le flot sortant est donc exécuté dès qu'un des flots entrants est activé.

- *Formalisme et exemple*

Le formalisme de représentation d'un nœud de fusion-test ainsi qu'un exemple sont donnés à la figure 3.31.

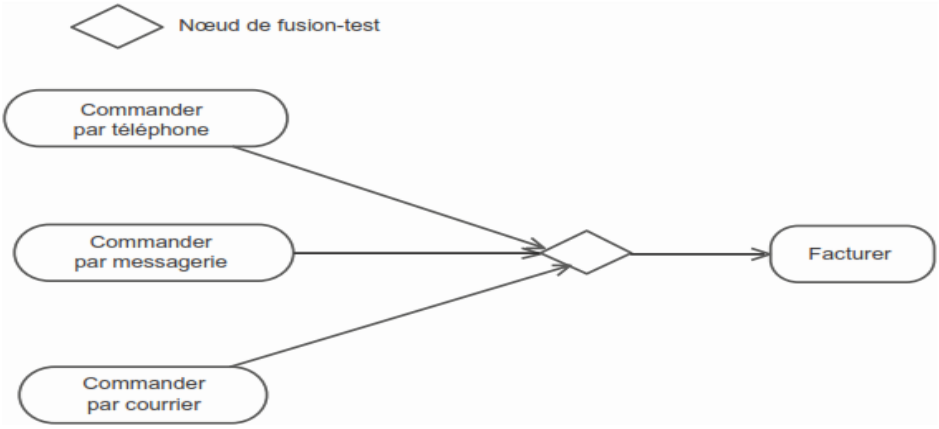


Figure 3.31 — Formalisme et exemple de diagramme d'activités avec un nœud de fusion-test

- *Pin d'entrée et de sortie*

Un pin d'entrée ou de sortie représente un paramètre que l'on peut spécifier en entrée ou en sortie d'une action. Un nom de donnée et un type de donnée peuvent être associés au pin. Un paramètre peut être de type objet.

- Formalisme et exemple

Chaque paramètre se représente dans un petit rectangle. Le nom du paramètre ainsi que son type sont aussi à indiquer. Le formalisme de représentation de pin d'entrée ou de sortie ainsi qu'un exemple sont donnés à la figure 3.32.

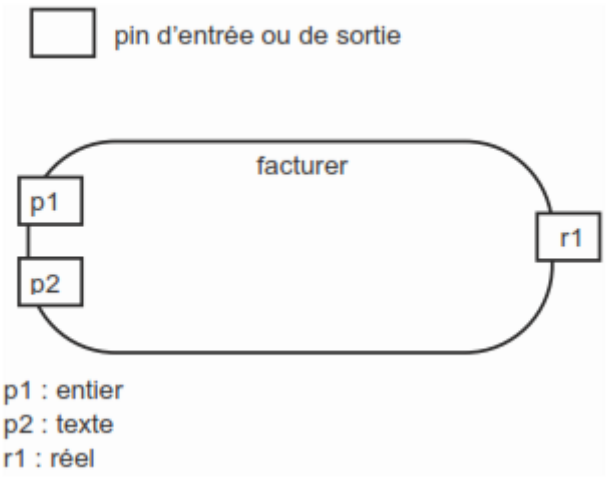


Figure 3.32 — Formalisme et exemple d'activité avec pin d'entrée et de sortie

- **Flot de données et nœud d'objet**

Un nœud d'objet permet de représenter le flot de données véhiculé entre les actions. Les objets peuvent se représenter de deux manières différentes : soit en utilisant le pin d'objet soit en représentant explicitement un objet.

- **Formalisme et exemple**

Le formalisme de représentation de flot de données et nœud d'objet est donné directement au travers d'un exemple (fig. 3.33).

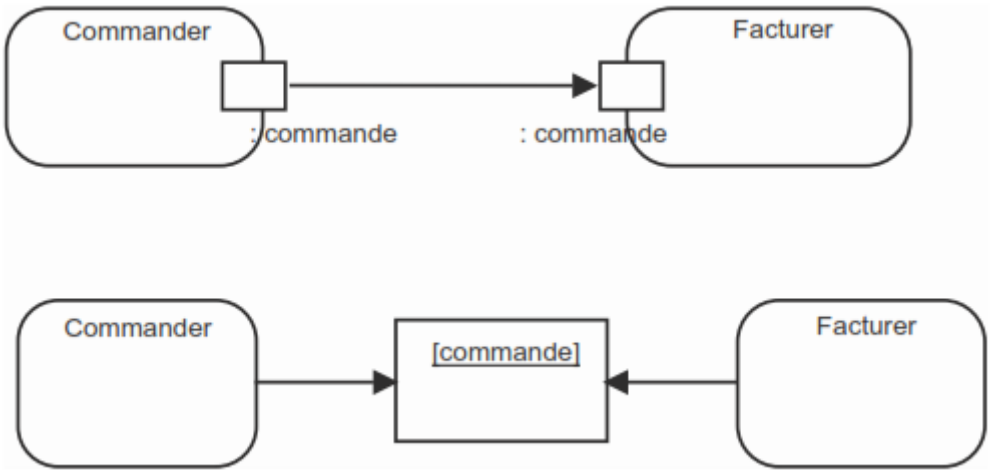


Figure 3.33 — Exemple de flot de données et de nœud d'objets

- *Partition*

UML permet aussi d'organiser la présentation du diagramme d'activité en couloir d'activités. Chaque couloir correspond à un domaine de responsabilité d'un certain nombre d'actions.

Les flots d'objets sont aussi représentés dans le diagramme. L'ordre relatif des couloirs de responsabilité n'est pas significatif.

2.1.2 Représentation du diagramme d'activité

Un exemple général de diagramme d'activité est donné à la figure 3.34.

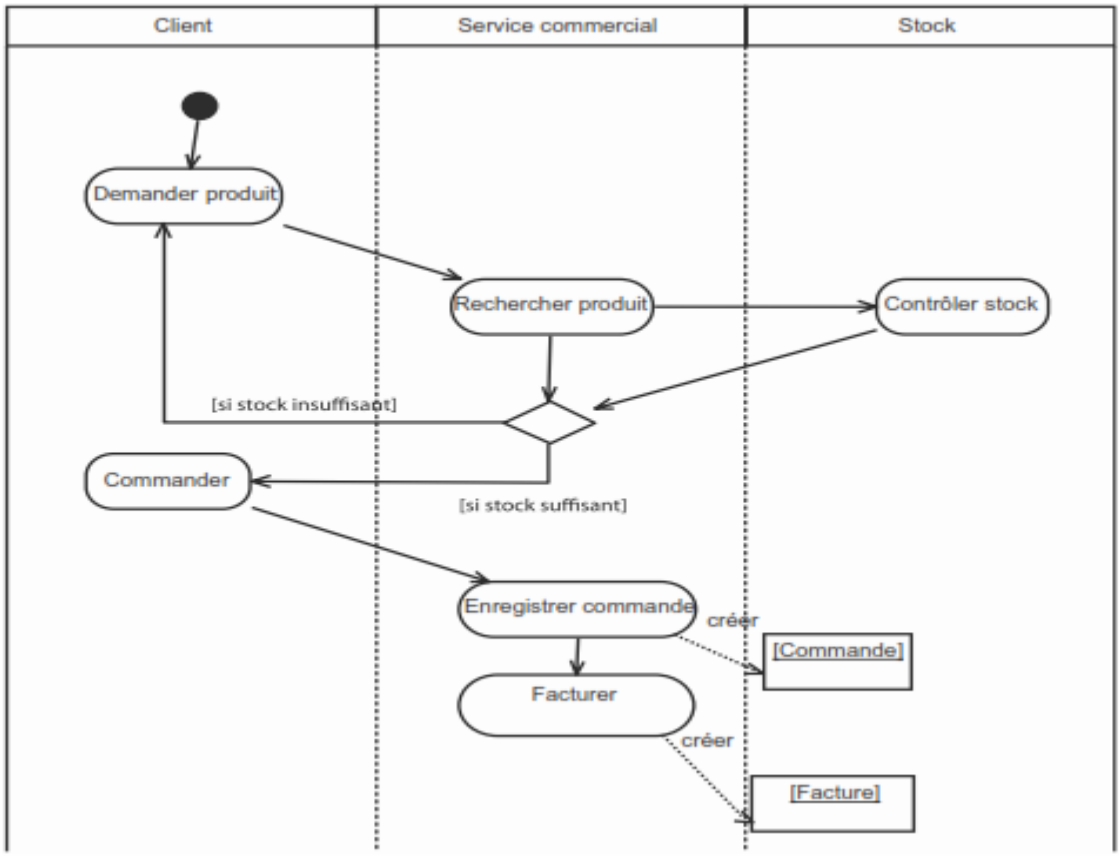


Figure 3.34 — Exemple de diagramme d'activité avec couloir d'activité

- **Représentation d'actions de communication**

Dans un diagramme d'activité, comme dans un diagramme de temps, des interactions de communication liées à certains types d'événement peuvent se représenter.

Les types d'événement concernés sont :

- signal,
- écoulement du temps.

- Formalisme et exemple

Le formalisme de représentation ainsi qu'un exemple d'actions de communication sont donnés à la figure 3.35.

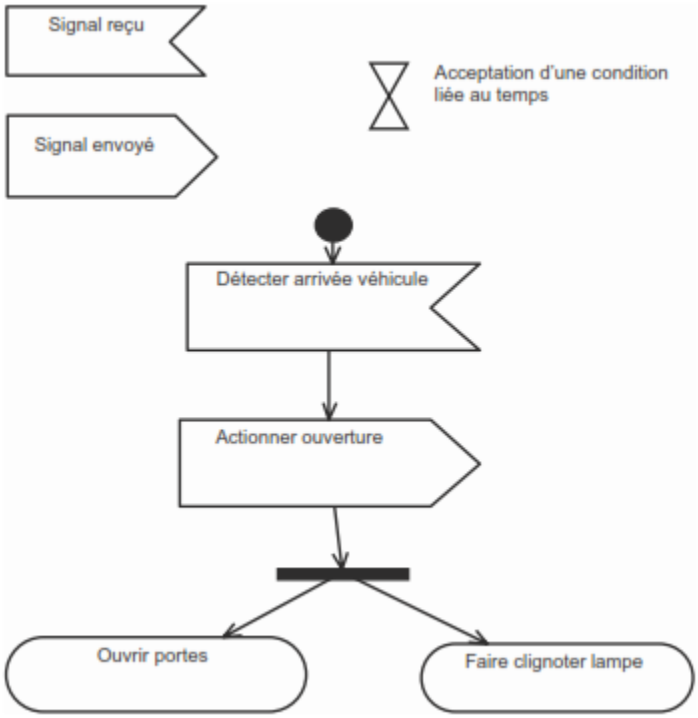


Figure 3.35 — Formalisme et exemple de diagramme d'activité avec actions de communication

Exercices

Exercice 1

Exercice relatif à la gestion de la bibliothèque

Deux acteurs ont été identifiés :

- Bibliothécaire chargé de l'approvisionnement des ouvrages, de la gestion du catalogue et de l'enregistrement des emprunts et retours d'ouvrages ;
- Gestionnaire, chargé de l'inscription des adhérents et de la relance des adhérents ayant dépassé le délai de restitution des ouvrages.

La représentation du diagramme d'activité est donnée à la figure 3.36.

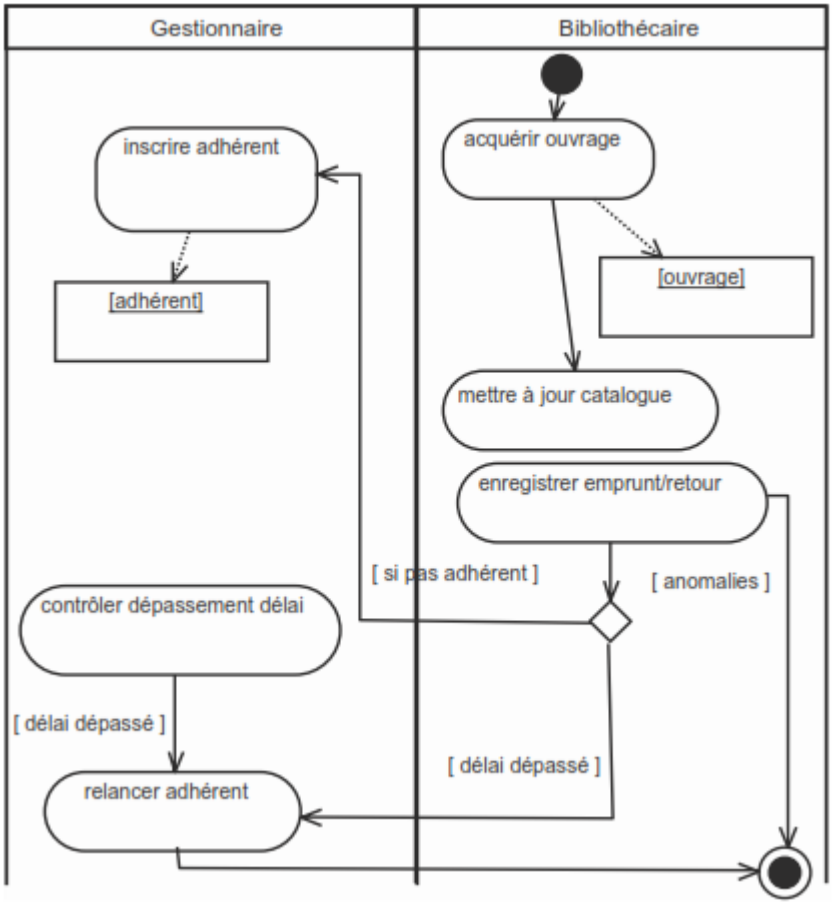


Figure 3.36 — Diagramme d'activité de l'exercice 1

2.2 DIAGRAMME D'ETAT-TRANSITION (DET)

2.2.1 Présentation générale et concepts de base

État-transition et événement

L'état d'un objet est défini, à un instant donné, par l'ensemble des valeurs de ses propriétés. Seuls certains états caractéristiques du domaine étudié sont considérés.

Le passage d'un état à un autre état s'appelle transition. Un événement est un fait survenu qui déclenche une transition.

Il existe quatre types d'événements :

- **Type appel de méthode (call)** - C'est le type le plus courant que nous traiterons dans la suite de la présentation.

- **Type signal** - Exemple: clic de souris, interruption d'entrées-sorties...La modélisation de la réception ou l'émission d'un signal est traitée dans le diagramme d'activité.
- **Type changement de valeur (vrai/faux)** - C'est le cas de l'évaluation d'une expression booléenne.
- **Type écoulement du temps** - C'est un événement lié à une condition de type after (durée) ou when (date).

Formalisme et exemple

Un objet reste dans un état pendant une certaine durée. La durée d'un état correspond au temps qui s'écoule entre le début d'un état déclenché par une transition *i* et la fin de l'état déclenché par la transition i+1. Une condition, appelée « garde », peut être associée à une transition.

Le formalisme de représentation d'état-transition est donné à la figure 3.9.

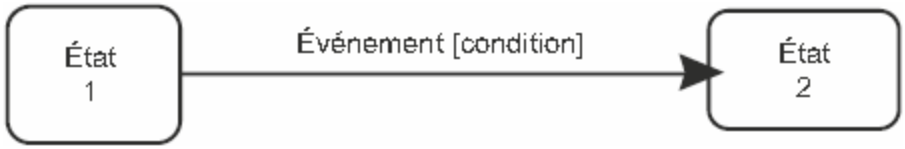


Figure 3.9 — Formalisme d'état-transition

La figure 3.10 donne un premier exemple d'état-transition. Dans cet exemple, pour un employé donné d'une entreprise, nous pouvons considérer les deux états significatifs suivants : état recruté, état en activité.

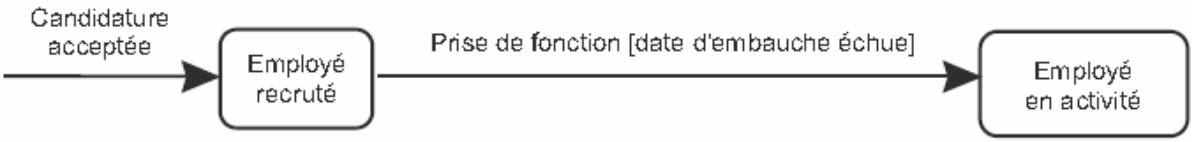


Figure 3.10 — Exemple d'état-transition

Action et activité

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

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

Formalisme et exemple

Le formalisme de représentation d'état-transition comprenant la représentation d'action et/ou activité est donné à la figure 3.11.

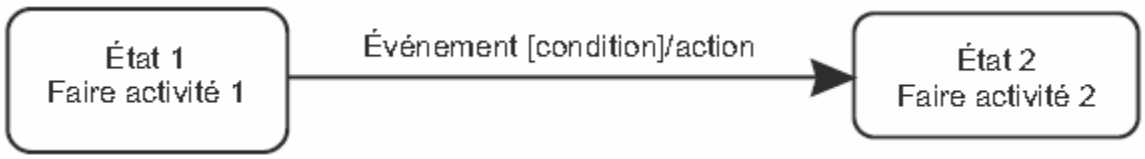


Figure 3.11 — Formalisme d'état-transition avec action et activité

La figure 3.12 montre un exemple des actions et activités d'états ainsi que la description complète d'une transition.

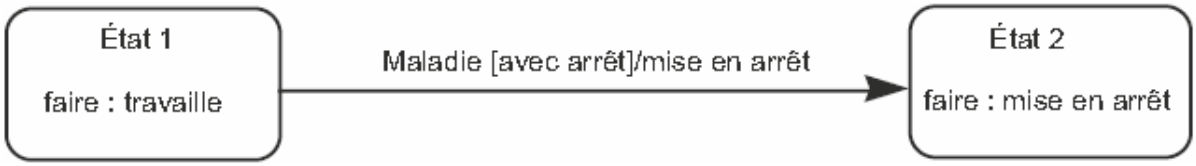


Figure 3.12 — Exemple d'état-transition avec action et activité

2.2.2 Représentation du diagramme d'état-transition d'un objet

L'enchaînement de tous les états caractéristiques d'un objet constitue le diagramme d'état. Un diagramme d'états débute toujours par un état initial et se termine par un ou plusieurs états finaux sauf dans le cas où le diagramme d'états représente une boucle. À un événement peut être associé un message composé d'attributs.

Formalisme et exemple

Le formalisme de représentation des états initial et final est donné à la figure 3.13.

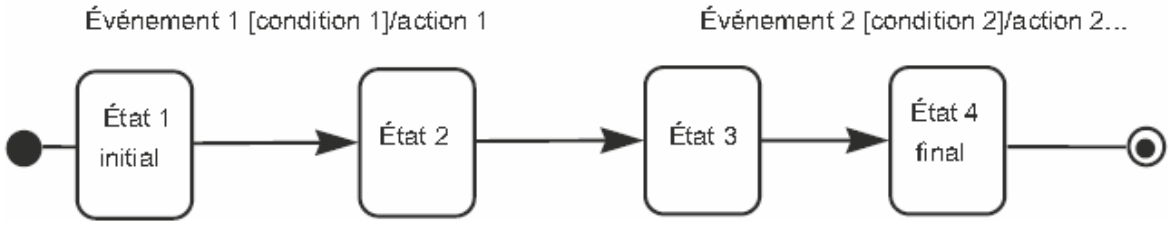


Figure 3.13 — Formalisme de représentation des états initial et final

Afin de nous rapprocher des situations réelles, nous proposons à la figure 3.14 un premier exemple tiré d'une gestion commerciale qui montre le diagramme d'état-transition de l'objet client.

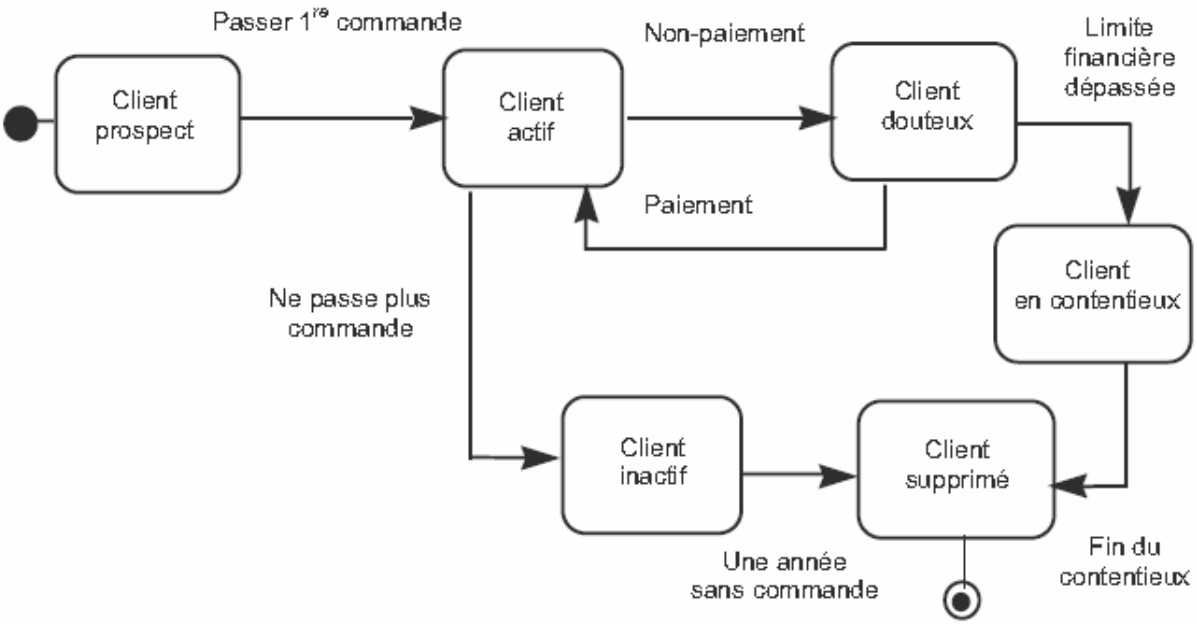
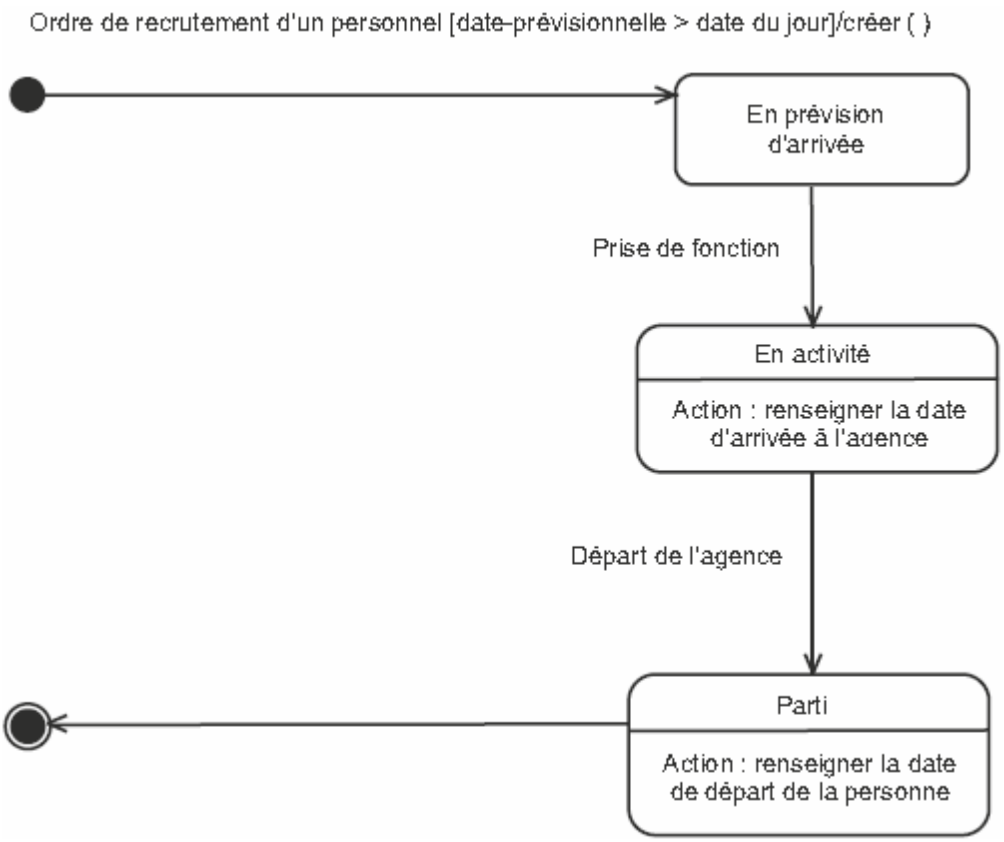


Figure 3.14 — Diagramme d'état-transition de l'objet client d'une gestion commerciale

Nous proposons comme second exemple, à la figure 3.15, le diagramme d'état-transition de l'objet « personnel » qui se caractérise par trois états :

- En prévision d'arrivée : si la date prévisionnelle est postérieure à la date du jour.
- En activité : état qui correspond à un personnel ayant une date d'arrivée renseignée.
- Parti : état qui correspond à un personnel ayant une date de départ renseignée.



2.2.3 Compléments sur le diagramme d'état-transition

Composition et décomposition d'etat

Il est possible de décrire un diagramme d'état-transition à plusieurs niveaux. Ainsi, à un premier niveau, le diagramme comprendra des états élémentaires et des états composites. Les états composites seront ensuite décrits à un niveau élémentaire dans un autre diagramme. On peut aussi parler d'état composé et d'état composant.

Formalisme et exemple

Le formalisme de représentation d'états composites est donné à la figure 3.16.

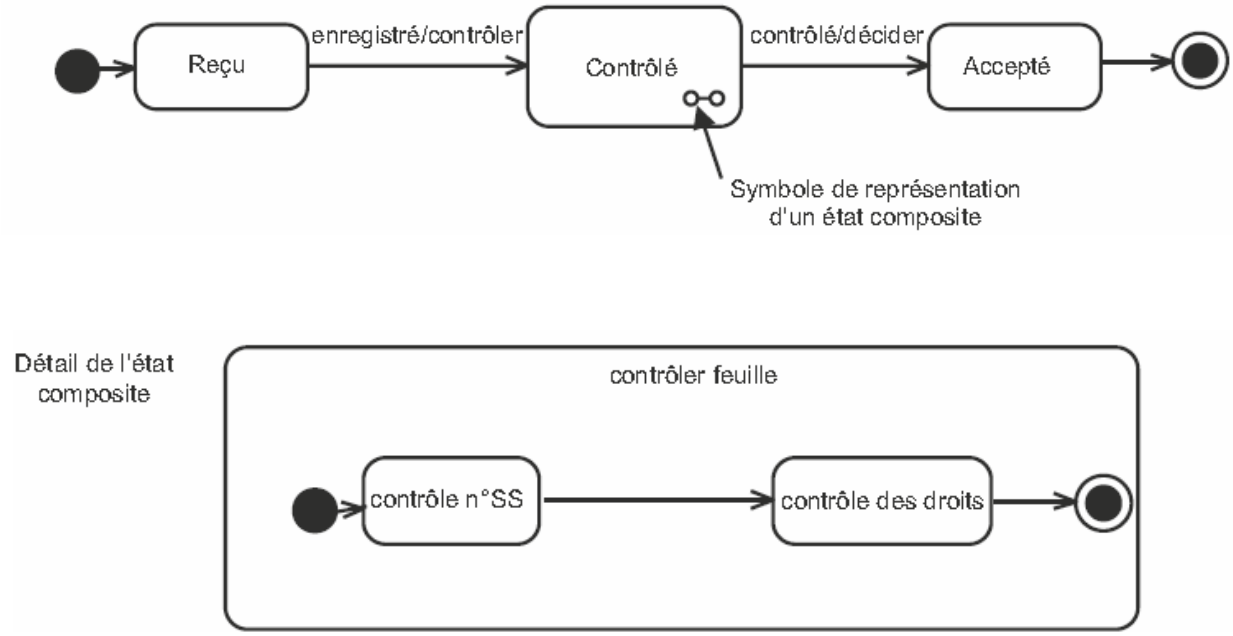


Figure 3.16 — Exemple d'état composite

Dans cet exemple, l'état contrôlé est un état composite qui fait l'objet d'une description individualisée à un second niveau que l'on appelle aussi sous-machine d'état.

Point d'entrée et de sortie

Sur une sous-machine d'état, il est possible de repérer un point d'entrée et un point de sortie estdonnéàlafigure3.

Formalisme et exemple

Le formalisme de représentation d'une sous-machine d'état avec point d'entrée et de sortie est donné à la figure 3.17.

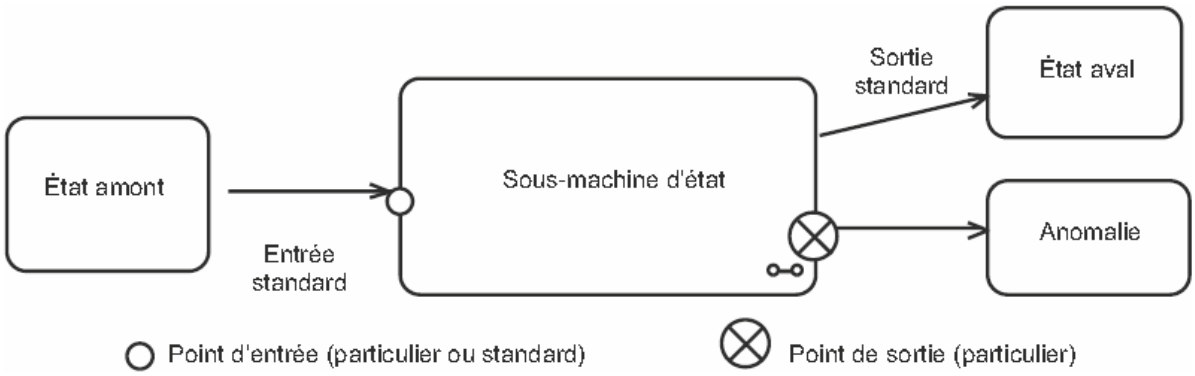


Figure 3.17 — Exemple d'une sous-machine d'état avec point d'entrée et de sortie

Point de jonction

Lorsque l'on veut relier plusieurs états vers d'autres états, un point de jonction permet de décomposer une transition en deux parties en indiquant si nécessaire les gardes propres à chaque segment de la transition.

À l'exécution, un seul parcours sera emprunté, c'est celui pour lequel toutes les conditions de garde seront satisfaites.

Formalisme et exemple

Le formalisme de représentation d'états-transitions avec point de jonction est donné à la figure 3.18.

Point de choix

Le point de choix se comporte comme un test de type : si condition faire action l sinon faire action2.

Formalisme et exemple

Le formalisme de représentation d'états composites est donné à la figure 3.19.

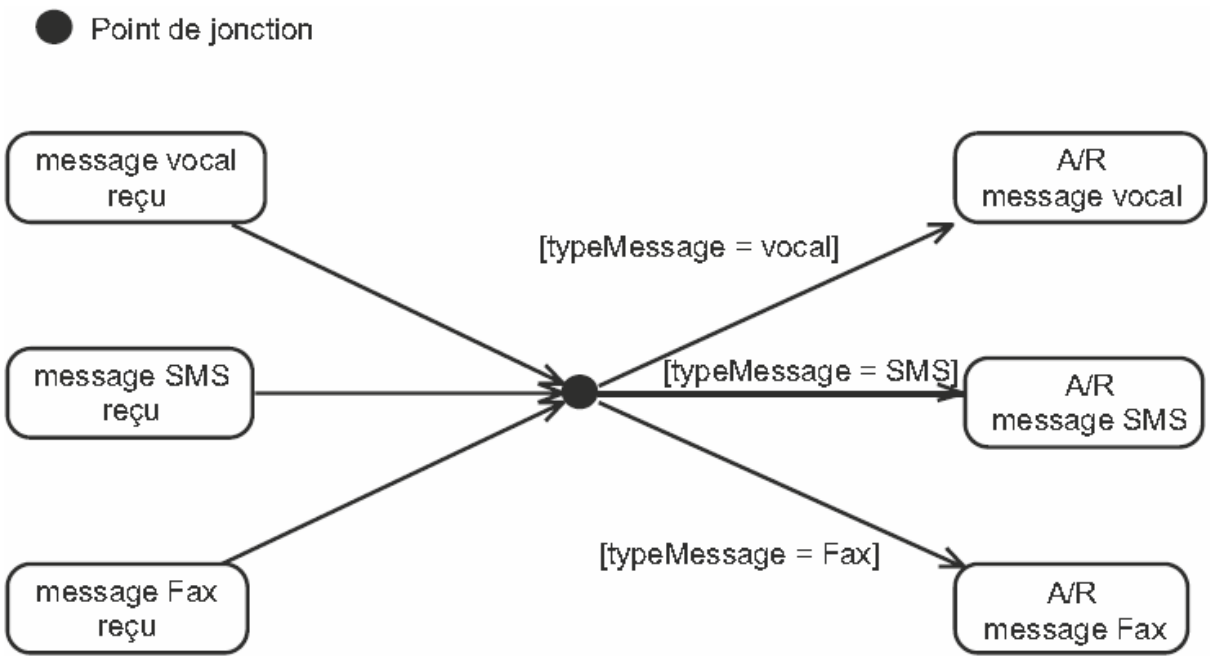


Figure 3.18 — Exemple d'états-transitions avec point de jonction

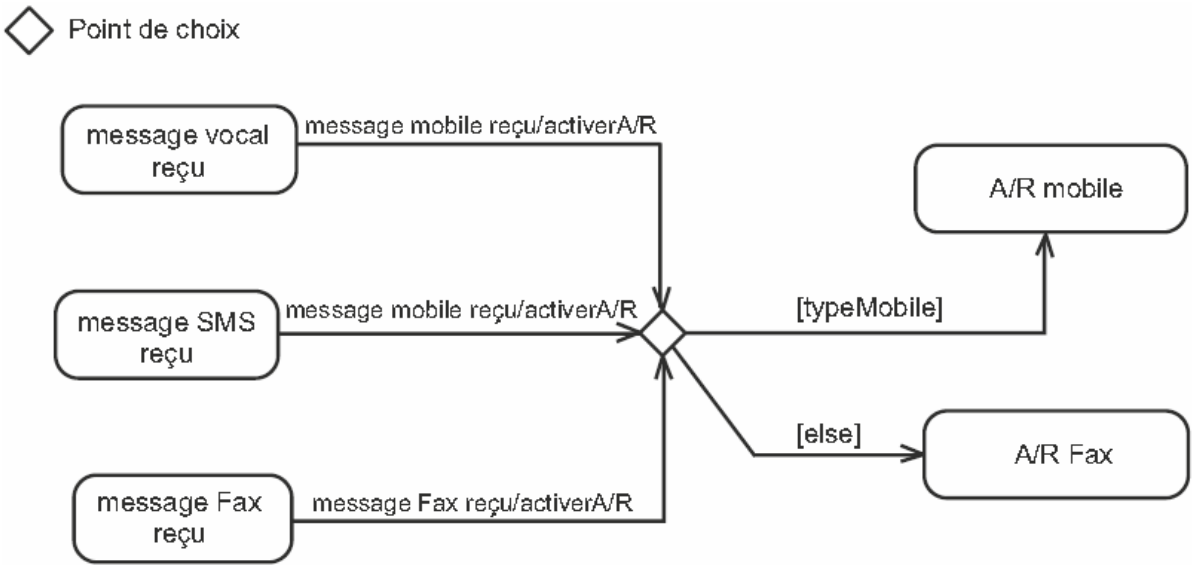


Figure 3.19 — Exemple d'états-transitions avec point de choix

État historique

La mention de l'historisation d'un état composite permet de pouvoir indiquer la réutilisation du dernier état historisé en cas de besoin.

Formalisme et exemple

Le formalisme de représentation d'états historisés est donné à la figure 3.20.

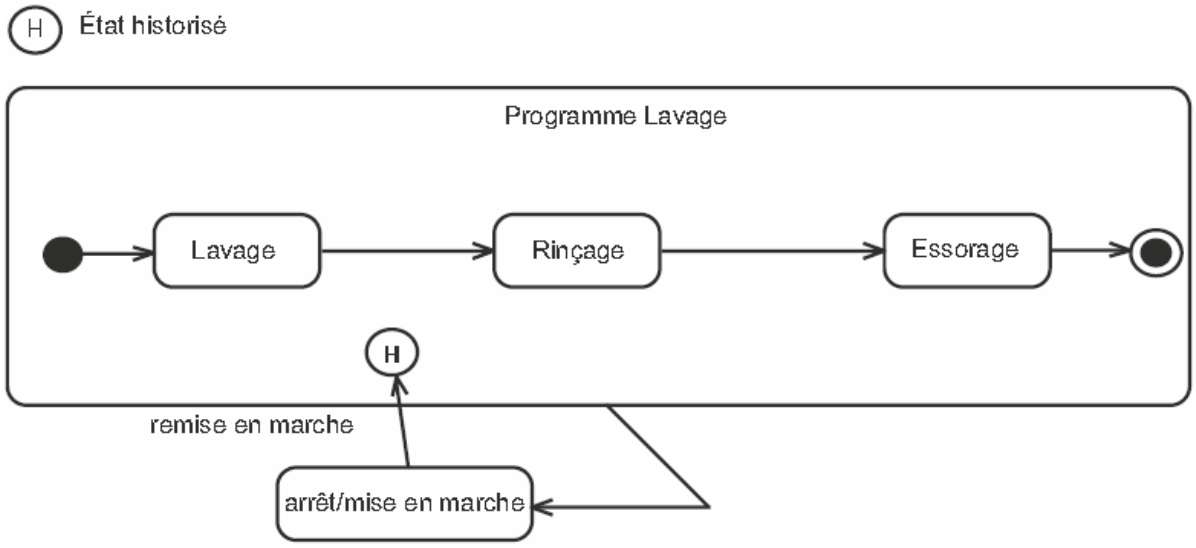


Figure 3.20 — Exemple d'états-transitions historisés

3.2.4 Exercices

Exercice 1

Énoncé

Soit à représenter le diagramme d'état-transition d'un objet personnel en suivant les événements de gestion depuis le recrutement jusqu'à la mise en retraite.

Après le recrutement, une personne est considérée en activité dès sa prise de fonction dans l'entreprise. Au cours de sa carrière, nous retiendrons seulement les événements : congé de maladie et prise de congé annuel. En fin de carrière, nous retiendrons deux situations : la démission et la retraite.

Corrigé

Nous proposons au lecteur un corrigé type à la figure 3.21. Ce corrigé ne représente qu'une solution parmi d'autres variantes possibles suivant la lecture faite de l'énoncé. Pour notre part, nous avons retenu les états caractéristiques : recruté, activité, en congé, en arrêt, parti et retraite.

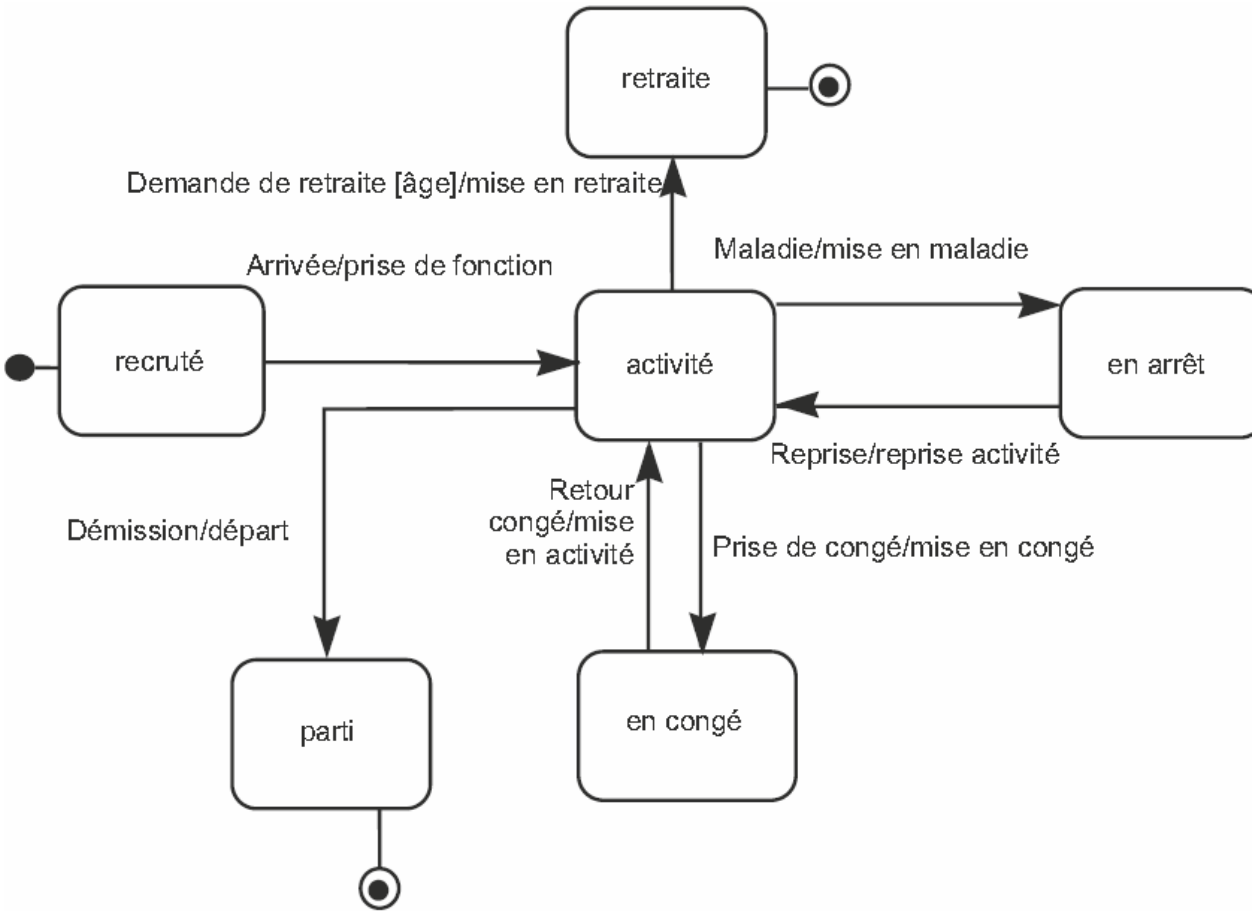


Figure 3.21 — Diagramme d'état-transition de l'exercice 1

2.3 DIAGRAMME DE COMMUNICATION (DCO)

2.3.1 Présentation générale et concepts de base

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

- *Rôle*

Chaque participant à un échange de message correspondant à une ligne de vie dans le diagramme de séquence se représente sous forme d'un rôle dans le diagramme de communication. Un rôle est identifié par :

<nom de rôle> : <nom du type>

Une des deux parties de cette identification est obligatoire ainsi que le séparateur « : ». Le nom du rôle correspond au nom de l'objet dans le cas où l'acteur ou la classe ont un rôle unique par rapport au système. Le nom du type correspond au nom de la classe lorsque l'on manipule des objets.

Exemple

administrateur : utilisateur

Pour un utilisateur qui est vu au travers de son rôle d'administrateur.

- *Message*

Un message correspond à un appel d'opération effectué par un rôle émetteur vers un rôle récepteur. Le sens du message est donné par une flèche portée au-dessus du lien reliant les participants au message (origine et destinataire). Chaque message est identifié par :

<numéro> : nom ( )

Plus précisément l'identification d'un message doit respecter la syntaxe suivante :

[n° du message préc. reçu] « . » n° du message [clause d'itération] [condition] « : » nom du message.

- Numéro du message précédent reçu : permet d'indiquer la chronologie des messages.
- Numéro du message : numéro hiérarchique du message de type 1.1,1.2... avec utilisation de lettre pour indiquer la simultanéité d'envoi de message.
- Clause d'itération : indique si l'envoi du message est répété. La syntaxe est * [spécification de l'itération].
- Condition : indique si l'envoi du message est soumis à une condition à satisfaire.

Exemples

1.2.1 * [3 fois] pour un message à adresser trois fois de suite. 1.2a et 1.2b pour deux messages envoyés en même temps.

Exemple récapitulatif de désignation de message :

1. 2a. 1. l[sit> 100]:lancer( )

Ce message signifie :

- 1.2a : numéro du message reçu avant l'envoi du message courant.
- 1.1 : numéro de message courant à envoyer.
- [si t > 100] : message à envoyer si t > 100.
- lancer( ) : nom du message à envoyer.

2.3.2 Formalisme et exemple

Les rôles correspondent à des objets. Le lien entre les rôles est représenté par un trait matérialisant le support des messages échangés. La figure 3.58 donne le formalisme de base du diagramme de communication.

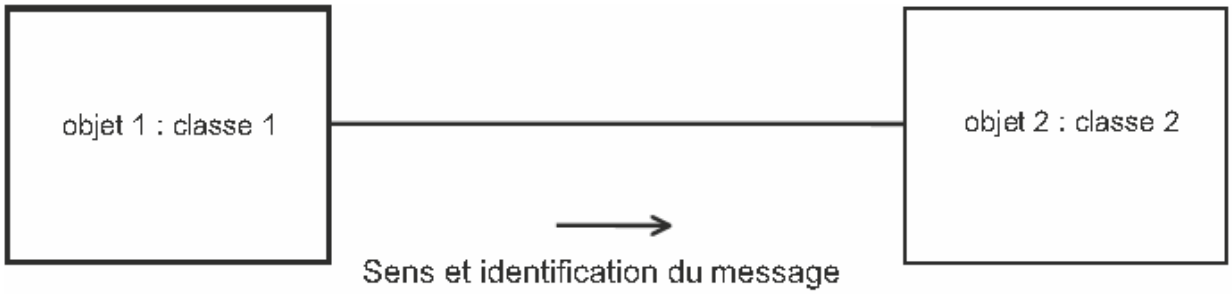


Figure 3.58 — Formalisme de base du diagramme de communication

Un exemple de diagramme de communication est donné à la figure 3.59.

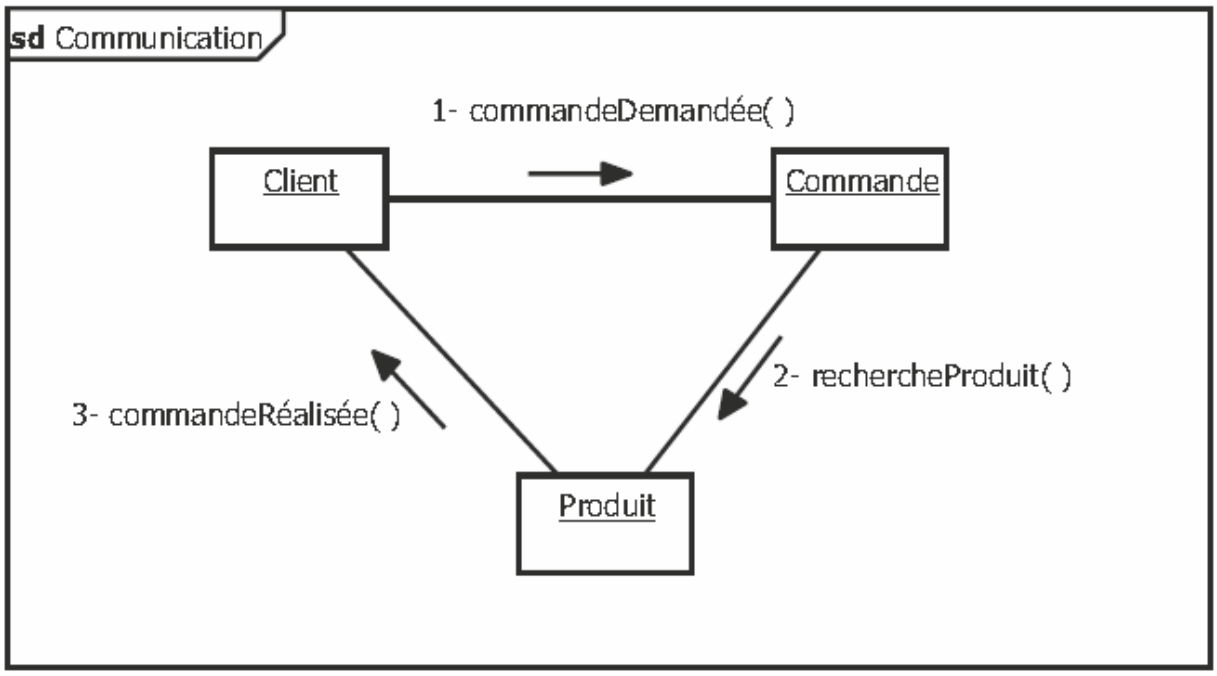


Figure 3.59 — Exemple de diagramme de communication

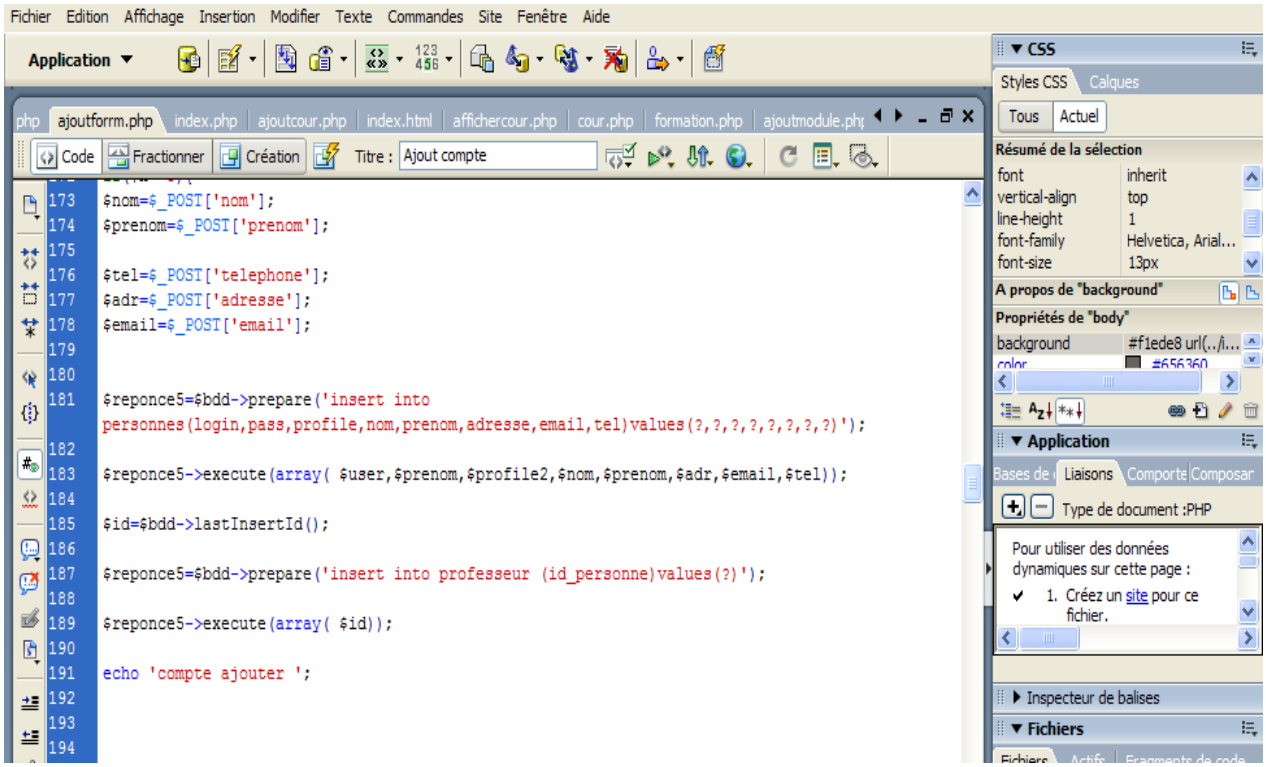
3.PRESENTATION D'ENVIRONNEMENT DE DEVELOPPEMENT ET DU SGBD ;

Dans cette partie il est question de faire une présentation sommaire de l'environnement de développement de l'application (IDE) du système de gestion de base de données à utiliser sans entré dans trop de détails.

4. CODIFICATION

Il sera aussi nécessaire de présenter quelques codes sources à ce niveau.

Par exemple les codes sources ci-dessous :



5. TEST

Pour s'assurer d'une bonne démarche qualité et pour être sûr que le produit correspond aux attentes prédéfinies au départ, des tests doivent être réalisés le long du cycle de développement du produit. Pour bien mener ces tests, les procédures de tests avec trois questions

prépondérantes : Quoi ? Quand ? Qui ? Comment ? ont été mise en place.

Exemple d'une procédure de test pour l'application en ligne de gestion de cours.

1. Quoi ? Qu'est ce qu'on teste ?

- **La plateforme** : nous avons testé toutes les fonctionnalités et modules de notre plate forme et ceci sur différents environnements techniques afin de prévoir des processus d'utilisation sans risque.
- **Les contenus** : nous avons testé le comportement du module avant et après son intégration dans la plateforme et ceci même pendant le développement ce qui nous a permis d'adapter notre méthodologie de développement.

2. Quand ? A Quel moment tester ?

Les tests doivent être réalisés le long du processus de création de notre plateforme. Nous avons démarré les tests dès la phase de construction et transition.

3. Qui? Qui est responsable des tests ?

Autant que chef de projet, nous avons été amené à préparer les procédures de tests ainsi que veiller à leurs bons déroulements.

Pour la validation, c'est l'équipe projet en entier qui s'en est occupée.

4.Comment ? Quels sont les types de tests à faire ?

Pour notre projet, nous avons programmés trois types de tests afin de pouvoir cerner toutes les facettes du dispositif et sa réalisation.

- Tests de fonctionnalités : tests des boutons de navigation, des liens, menus et sous menus.
- Tests d'ergonomie-navigation : tests sur l'interactivité, tests de tous les scénarios et dans tous les cas possibles, tests de tous les clicks et les éventuels messages d'erreurs.

- Tests techniques : tester la plateforme sur plusieurs configurations matérielles et logicielles, une gamme variée d'ordinateurs et la montée en charge sur la plateforme pressentie.

Les tests sont indispensables pour offrir une plateforme qui répond au mieux à nos attentes et surtout qui présente moins de bugs possible. Mais pour la maintenance et l'évolution de la plateforme, nous proposons un processus de suivi.

6. PRESENTATION DES INTERFACES

Il sera nécessaire de présenter quelques interfaces les plus pertinentes telles que : la page d'accueil. Le menu principal, formulaire, états de sortie...

Exemples :

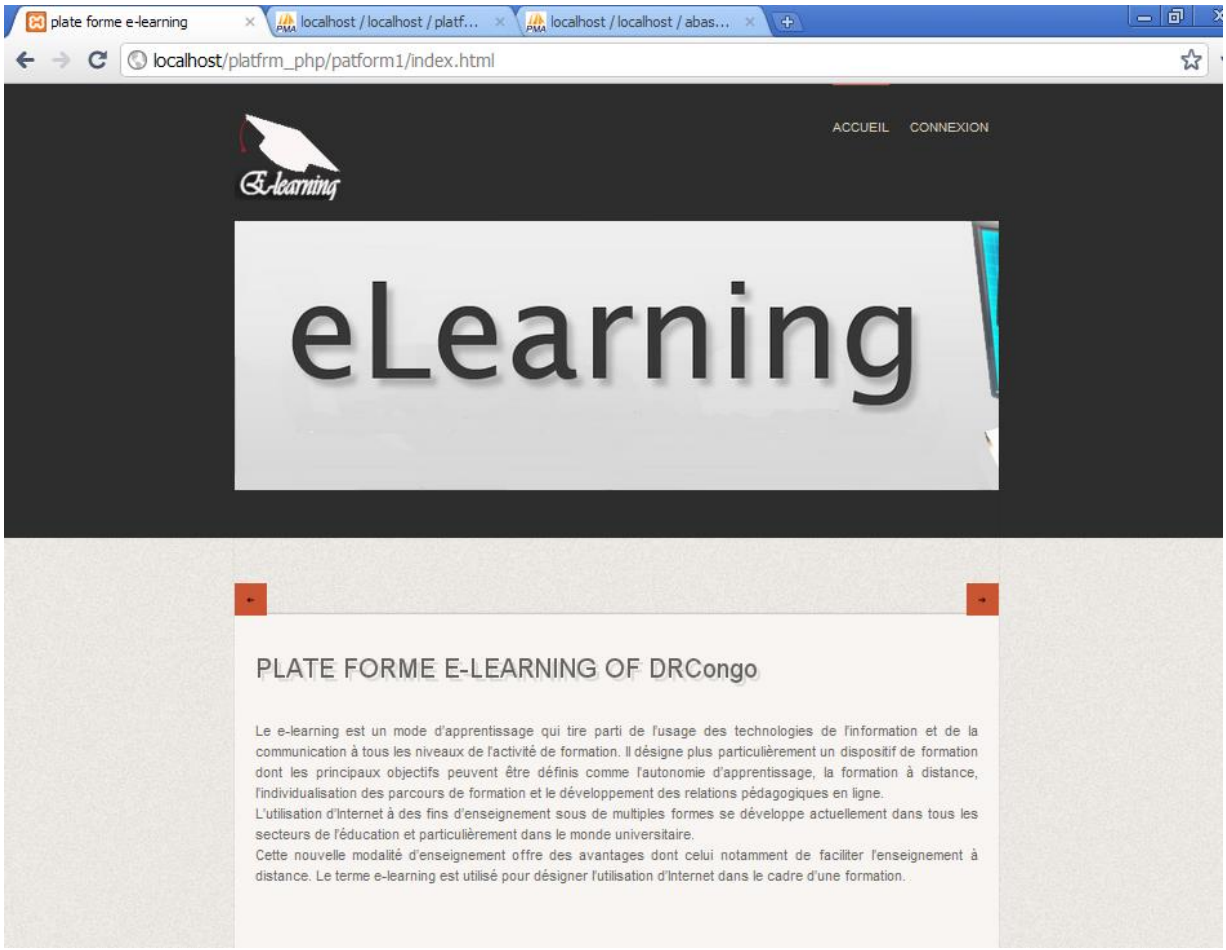


Figure : Page d'accueil

Figure: Page authentification

CHAPITRE 7 : TRANSITION

Après les opérations de test menées dans la phase précédente, il s'agit dans cette phase de livrer le produit pour une exploitation réelle. C'est ainsi que toutes les actions liées au déploiement sont traitées dans cette phase.

De plus, des « bêta tests » sont effectués pour valider le nouveau système auprès des utilisateurs.

Cette phase consiste aussi à présenter un schématiquement la manière dont le nouveau système sera implanté lors de la mise en œuvre chez le maitre d'ouvrage. Pour ce faire nous allons utiliser le diagramme de déploiement.

1. DIAGRAMME DE DÉPLOIEMENT (DPL)

Le diagramme de déploiement permet de représenter l'architecture physique supportant l'exploitation du système. Cette architecture comprend des nœuds correspondant aux supports physiques (serveurs, routeurs...) ainsi que la répartition des artefacts logiciels (bibliothèques, exécutables...) sur ces nœuds. C'est un véritable réseau constitué de nœuds et de connexions entre ces nœuds qui modélise cette architecture.

1.1 Nœud

Un nœud correspond à une ressource matérielle de traitement sur laquelle des artefacts seront mis en œuvre pour l'exploitation du système. Les nœuds peuvent être interconnectés pour former un réseau d'éléments physiques.

- *Formalisme et exemple*

Un nœud ou une instance de nœud se représente par un cube ou parallélépipède (fig. 2.43).

Figure 2.43 — Représentation de nœuds

- *Compléments sur la description d'un nœud*

Il est possible de représenter des nœuds spécialisés. UML propose en standard les deux types de nœuds suivants :

- Unité de traitement – Ce nœud est une unité physique disposant de capacité de traitement sur laquelle des artefacts peuvent être déployés. Une unité de traitement est un nœud spécialisé caractérisé par le mot-clé «device».
- Environnement d'exécution – Ce nœud représente un environnement d'exécution particulier sur lequel certains artefacts peuvent être exécutés. Un environnement d'exécution est un nœud spécialisé caractérisé par le mot clé «executionEnvironment ».

1.2 Artefact

Un artefact est la spécification d'un élément physique qui est utilisé ou produit par le processus de développement du logiciel ou par le déploiement du système. C'est donc un élément concret comme par exemple : un fichier, un exécutable ou une table d'une base de données.

Un artefact peut être relié à d'autres artefacts par notamment des liens de dépendance.

- *Formalisme et exemple*

Un artefact se représente par un rectangle (fig. 2.44) caractérisé par le mot-clé « artifact » et/ou une icône particulière dans le coin droit du rectangle.

Figure 2.44 — Représentation d'un artefact

Deux compléments de description sont proposés par UML pour la représentation des artefacts.

1.3 Spécification de déploiement

Une spécification de déploiement peut être associée à chaque artefact. Elle permet de préciser les conditions de déploiement de l'artefact sur le nœud sur lequel il va être implanté.

- Formalisme et exemple

Une spécification de déploiement se représente par un rectangle avec le mot-clé « deployment spec ». Un exemple d'un artefact avec une spécification de déploiement est donné à la figure 2.45.

Figure 2.45 — Exemple d'un artefact avec spécification de déploiement

1.4 Liens entre un artefact et les autres éléments du diagramme

Il existe deux manières de représenter le lien entre un artefact et son nœud d'appartenance :

- Représentation inclusive – Dans cette représentation, un artefact est représenté à l'intérieur du nœud auquel il se situe physiquement. Un exemple est donné à la figure 2.46.

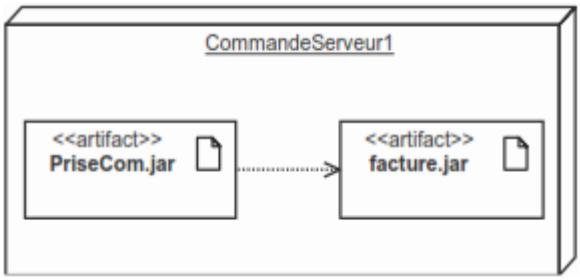


Figure 2.46 — Exemple de représentation inclusive d'artefact

- Représentation avec un lien de dépendance typé « deploy » – Dans ce cas l'artefact est représenté à l'extérieur du nœud auquel il appartient avec un lien de dépendance entre l'artefact et le nœud typé avec le mot-clé « deploy ». Un exemple est donné à la figure 2.47.

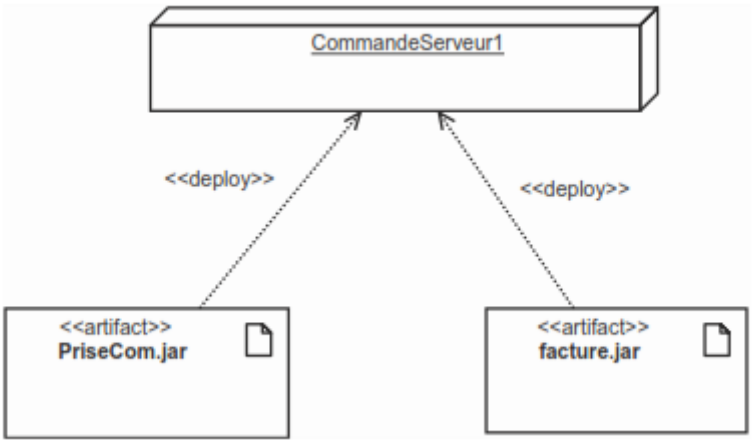


Figure 2.47 — Exemple de représentation d'artefact avec lien de dépendance

Un artefact peut représenter un ou plusieurs éléments d'un modèle. Le qualificatif «manifest » permet d'indiquer ce type de dépendance.

1.5 Représentation et exemples

Le diagramme de déploiement représente les nœuds de l'architecture physique ainsi que l'affectation des artefacts sur les nœuds conformément aux règles de déploiement définies. Un premier exemple est donné à la figure 2.48.

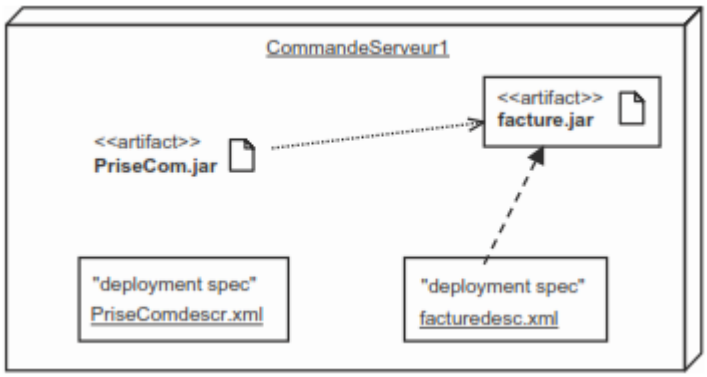


Figure 2.48 — Exemple de représentation d'un diagramme de déploiement

Un second exemple relatif à une implémentation d'une architecture J2EE avec quatre nœuds est donné à la figure 2.49.

Dans l'exemple de la figure 2.49, plusieurs composants sont déployés.

- Un serveur web où se trouvent les éléments statiques du site dans une archive : images, feuilles de style, pages html (static.zip).
- Un serveur d'application « front » sur le lequel est déployée l'archive « front.ear » composée de l'application web « front.war » et d'autres composants nécessaires au fonctionnement de cette archive web

comme «client-ejb.jar » (classes permettant l'appel aux EJB) et « commun.jar » (classes communes aux deux serveurs d'application).

- Un serveur d'application métier sur lequel sont déployés les composants : « ejb.jar ». Ils sont packagés dans l'archive « metier.ear ». Deux autres archives sont nécessaires au fonctionnement des EJB : « dao.jar » (classes qui permettent l'accès à la base de données) et « commun.jar » (classes communes aux deux serveurs d'application).
- Un serveur BDD (base de données) sur lequel sont stockées des procédures stockées PL/SQL : « scripts.sql ».

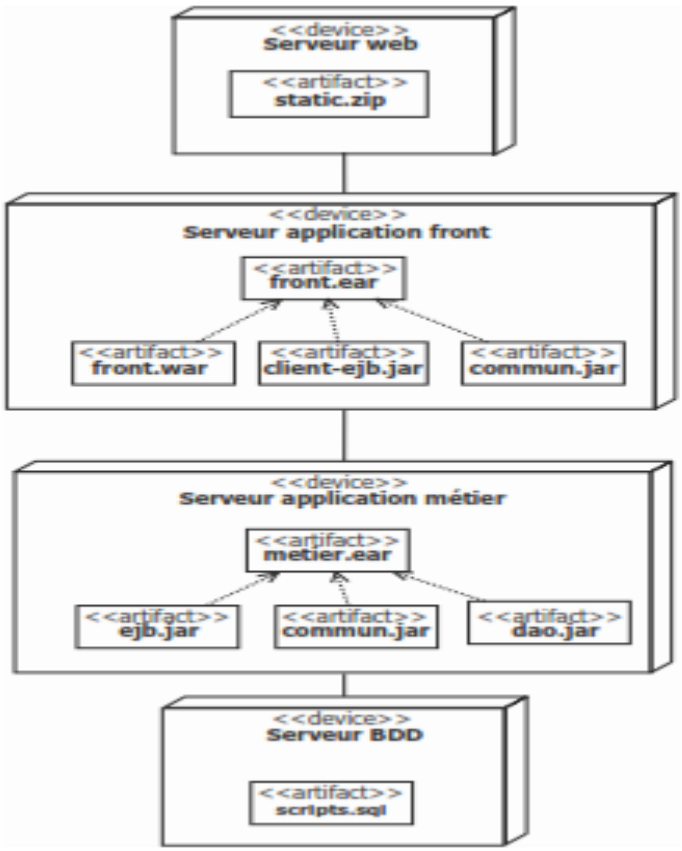


Figure 2.49 — Exemple de diagramme de déploiement comportant quatre nœuds