Ingénierie logicielle

L3 Info
2024-2025

Déroulement

- 1 séance de 4h chaque mercredi
- 1 séance de 2h jeudi 19/12

- Evaluation :
  - 3 projets notés en groupe
    - 1 cahier des charges pour le 11 décembre
    - 1 projet codé pour le 19 décembre
    - 1 projet codé pour le 8 janvier
  - Rattrapage : 1 devoir sur table

Introduction, définition

- *Concepts clefs*
- Le **génie logiciel** est un domaine des sciences de l'ingénieur dont l'objet d'étude est la conception, la fabrication, et la maintenance des systèmes informatiques complexes.
- Un **système** est un ensemble d'éléments interagissant entre eux suivant un certains nombres de principes et de règles dans le but de réaliser un objectif.
- Un **logiciel** est un ensemble d'entités nécessaires au fonctionnement d'un processus de traitement automatique de l'information. Parmi ces entités, on trouve par exemple : des programmes (en format code source ou exécutables);des documentations d'utilisation ;des informations de configuration.
- Un **modèle** : est une représentation schématique de la réalité.
- Une **base de données**: ensemble des données structurées et liées entre elles : stocké sur support à accès direct (disque magnétique) ; géré par un SGBD (Système de Gestion de Bases de Données), et accessible par un ensemble d'applications.
- Une **analyse** : c'est un processus d'examen de l'existant
- Une **conception** : est un processus de définition de la future application informatique.
- Un **système d'information** : ensemble des moyens (humains et matériels) et des méthodes se rapportant au traitement de l'information d'une organisation.

Introduction, définition

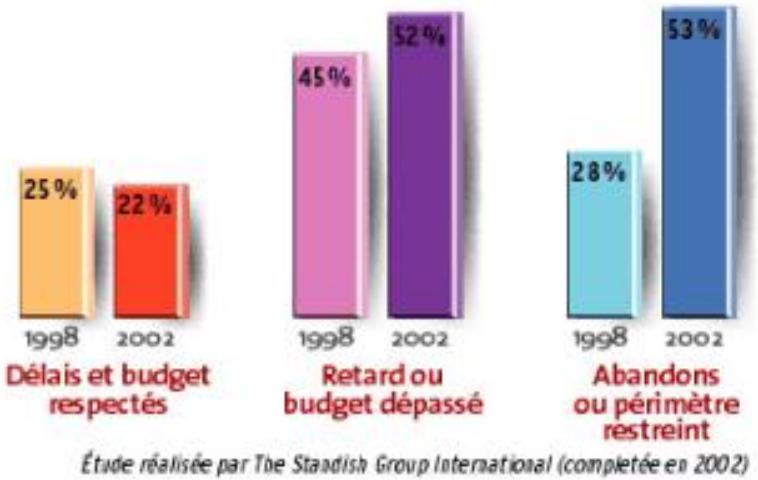
- Le Génie Logiciel est à rapprocher du Génie civil, Génie mécanique ou Génie chimique. La réalisation d'un pont ne peut être menée sans méthodologie, de même la réalisation d'un logiciel nécessite un minimum de précautions qui garantissent un certains nombre de propriétés.
- Le terme génie logiciel a été choisi pour exprimer le fait que le développement d’un logiciel doit se fonder sur des bases théoriques et sur un ensemble de méthodes et outils validés par la pratique, comme c’est le cas en génie civil, génie industriel, etc. Le génie logiciel considère ainsi le logiciel comme un objet complexe. La production du logiciel implique de ce fait des environnements de développement, avec toute la variété d'outils et d'approches dont on peut disposer, les méthodes et les techniques de gestion de processus, mais aussi les aspects humains au sein de l'équipe de développement et les relations que celle-ci entretient avec les commanditaires et les utilisateurs du produit.

Introduction, définition

- L'utilisation d'une méthodologique pour produire un logiciel s'est montrée incontournable par la crise de l'industrie du logiciel (crise du logiciel). Cette crise est apparue dans les années 70 lorsque l'on a pris conscience (caractéristiques de la crise du logiciel) de l'absence de la maîtrise des projets au niveau des coûts (le coût du logiciel dépassait déjà celui du matériel) et des délais, la mauvaise qualité des produits (les produits ne répondent pas aux besoins définis et des erreurs persistent dans le produit final). Ces problèmes suscitent des risques humains et économiques comme l'illustrent les exemples célèbres suivants :
  - TAURUS, un projet d'informatisation de la bourse londonienne : définitivement abandonné après 4 années de travail et 100 millions de £ (livres sterling) de pertes,
  - L'avion C17 (1993) de McDonnell Douglas livré avec un dépassement de 500 millions de $ (19 calculateurs hétérogènes et 6 langages de programmation différents),
  - Les origines de l'échec d'Ariane 5 (1996) sont liées à une exception logicielle qui a provoqué l'autodestruction d'un autre module empêchant ainsi la transmission des données d'attitude correctes
  - En 2007, Express Union avec le passage de la numérotation téléphonique à 8 chiffres
  - Et de nombreuses d'autres erreurs que l'on ignore …
- Remarque : Une erreur, petite soit-elle peut être dangereuse par exemple pour la chirurgie à distance, commande temps réel de l'avion, …

Introduction, définition

- D'après le cabinet de conseil en technologies de l'information Standish Group International, les pannes causées par des problèmes de logiciel ont coûté en 2000 aux entreprises du monde entier environ 175 milliards de dollars.
- Une autre étude menée par le même groupe sur la conduite des projets informatiques montre le résultat suivant



Introduction, définition

- Le Génie Logiciel est né en Europe dans les années 70. Il a été défini par un groupe de scientifiques pour répondre à un problème qui devenait de plus en plus évident : le logiciel n’est pas fiable et il est difficile de réaliser dans des délais prévus des logiciels satisfaisant leurs besoins initiaux.

- 1968 : naissance GL à la conférence de l'OTAN à Garmisch-Partenkirchen (Allemagne)
- 1973 : première conférence sur le GL
- 1975 : première revue sur le GL (IEEE Transaction of Software Engineering)
- 1980 : début des AGL

Introduction, définition

Les objectifs ? Comment les satisfaire ?

- Le logiciel est aujourd'hui présent partout, sa taille et sa complexité augmentent de façon exponentielle, les exigences en besoins et en qualité sont de plus en plus sévères. L'objectif du génie logiciel est de développer dans les délais les logiciels de qualité. Le Génie Logiciel se préoccupe des procédés de fabrication de logiciels de façon à s'assurer que le produit qui est fabriqué :
  - réponde aux besoins des utilisateurs : **Fonctionnalités**
  - reste dans les limites financières prévues au départ : **Coût**
  - corresponde au contrat de service initial : **Qualité** (la notion de qualité de logiciel est multi-forme)
  - reste dans les limites de temps prévues au départ : **Délai**

Introduction, définition

- La solution imaginée pour répondre à cette crise a été l'industrialisation de la production du logiciel. Cette industrialisation vise la maîtrise du processus de développement et définit pour cela des procédés de fabrication de manière à satisfaire les besoins des utilisateurs, la qualité du logiciel, les coûts et les délais de production.
- L'IEEE définit le génie logiciel comme l'application d'une approche systématique, disciplinée et quantifiable au développement, à l'exploitation et à la maintenance des logiciels; c'est-à-dire l'application de l'ingénierie aux logiciels

Introduction, définition

- Remarquons que cette définition fait référence à deux aspects indispensables dans la conception d'un logiciel :
  - Le **développement** des logiciels - ensemble de formalités, des marches à suivre et des démarches pour obtenir un résultat déterminé et ;
  - La **maintenance** et le suivi des logiciels - ensemble d'opérations permettant de maintenir le fonctionnement d'un équipement informatique.
- Le génie logiciel englobe les tâches suivantes :
  - La **spécification** : capture des besoins, cahier des charges, spécifications fonctionnelles et techniques
  - La **conception** : analyse, choix de la modélisation, définition de l'architecture, définition des modules et interfaces, définition des algorithmes
  - L'**implantation** : choix d'implantations, codage du logiciel dans un langage cible
  - L'**intégration** : assemblage des différentes parties du logiciel
  - La **documentation** : manuel d'utilisation, aide en ligne
  - La **vérification** : tests fonctionnels, tests de la fiabilité, tests de la sûreté
  - La **validation** : recette du logiciel, conformité aux exigences du CDC
  - Le **déploiement** : livraison, installation, formation
  - La **maintenance** : corrections, et évolutions

Introduction, définition

- Malgré les efforts menés dans le domaine du GL, il reste encore aujourd'hui moins avancé par rapport aux autres sciences de l'ingénieur (génie civil, …). Le GL souffre des maux majeurs :
  - Le logiciel est un objet immatériel (abstrait) : comment apprécier sa complexité ?
  - Le logiciel est très flexible, malléable au sens de facile à modifier : facile de modifier une ligne de code, mais infiniment difficile de garantir que le programme continuera à fonctionner correctement
  - Ses caractéristiques attendues sont difficiles à figer au départ et souvent remises en cause en cours du développement : difficulté de spécifier les besoins ;
  - L'évolution rapide des technologies et besoins de plus en plus complexes : complexité liée à l'environnement informatique ;
  - Le logiciel ne s'use pas, il devient obsolète (par rapport aux concurrents, par rapport au contexte technique, par rapport aux autres logiciels, ...) ;

  Le GL est un domaine très vaste faisant intervenir plusieurs acteurs, aspects techniques d'où des problème de communication, organisation, … => Management de projet
  - Le niveau de formalisation du savoir-faire informatique est très faible (=>les projets logiciels diffèrent et l'expérience compte beaucoup)

Cependant, grâce au GL des progrès ont été réalisés (compilateur C : Unix, logiciels 2D/3D, systèmes embarqués, …), mais la complexité des systèmes ne cesse de s'accroître. Du fait de cette complexité liée à l'élaboration des logiciels, il est difficile qu'un logiciel soit exempt de défauts

Introduction, définition

- Le GL est en forte relation avec presque tous les autres domaines de l'informatique : langages de programmation, bases de données, informatique théorique (automates, réseaux de Petri, types abstraits,...), etc. Le GL utilise l'informatique comme un outil pour résoudre des problèmes de traitement de l'information.
- Dans sa partie technique, le GL présente un spectre très large depuis des approches très formelles (spécifications formelles, approches transformationnelles, preuves de programmes) jusqu'à des démarches absolument empiriques (démarches qui s'appuient seulement sur l'expérience). Cette variété reflète la diversité des types de systèmes à produire :
  - gros systèmes de gestion : le plus souvent des systèmes transactionnels construits autour d'une base de donnée centrale ;
  - systèmes temps-réel, qui doivent répondre à des événements dans des limites de temps prédéfinies et strictes ;
  - systèmes distribués sur un réseau de machines (distribution des données et/ou des traitements) ;
  - systèmes embarqués et systèmes critiques, interfacés avec un système à contrôler (aéronautique, centrales nucléaires, ...).

Introduction, définition

- Pour produire ces différents systèmes, le GL en tant que vaste domaine faisant intervenir plusieurs acteurs s'appuie sur :
  - Informatique
  - Droit (PI, Travail, …)
  - Gestion : RH, Finances, Organisation
  - Sciences Sociales et Humaines : communication, formation, accompagnement, …
- La spécialisation en GL peut se faire suivant ses différentes branches (métiers du GL) :
  - Ingénierie des besoins (requirement Engeneering)
  - Architecte (software achitect)
  - Programmeur (software programming)
  - Testeur (software testing)
  - Gestionnaire de projet logiciel (software management : people management and team organisation, quality management, cost estimation, software planning and control)

Introduction, définition

La qualité du logiciel

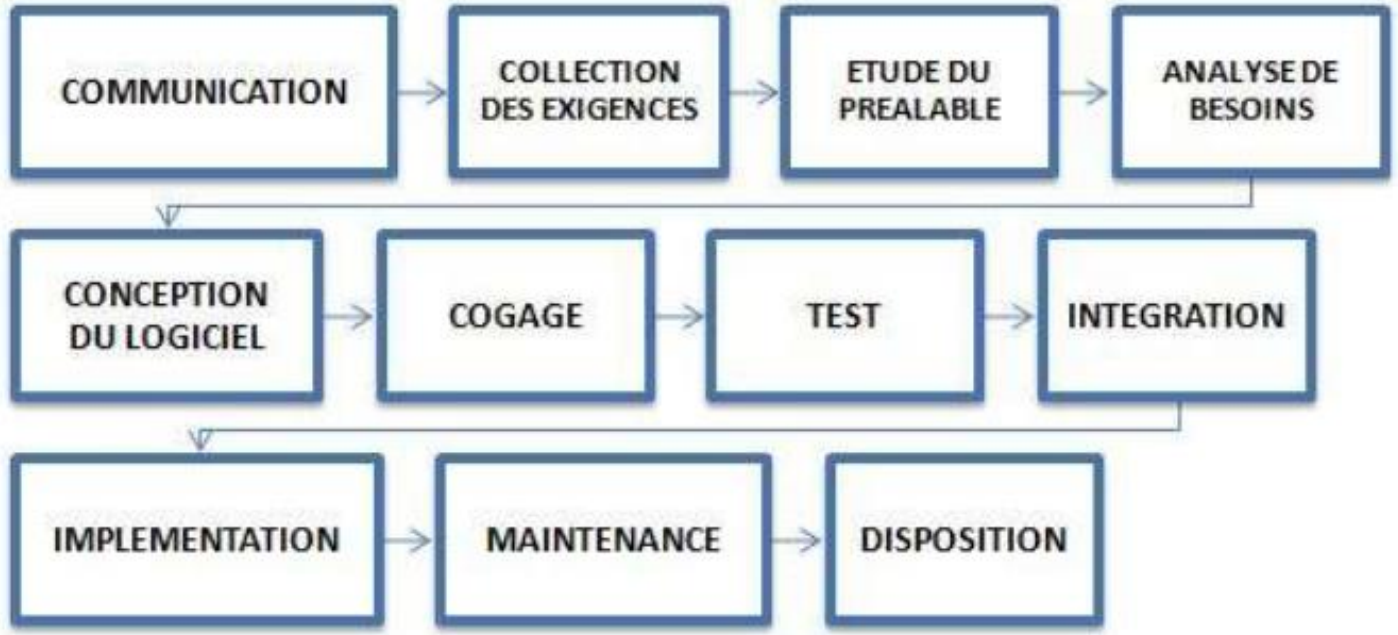
- La qualité est un processus qui dépasse la notion de logiciel. On peut l'appliquer à tout processus industriel. La difficulté d'application au niveau des logiciels réside sur le caractère abstrait et invisible du logiciel.
- On distingue :
  - La qualité externe qui exprime le point de vue des utilisateurs : elle peut concerner les fonctionnalités (besoins satisfaits, …), les performances (temps de réponse, …) et l'ergonomie (facilité d'utilisation, …).
  - La qualité interne qui exprime le point de vue du technicien : Elle concerne principalement l'optimisation, l'adaptabilité, la réutilisabilité, l'efficacité, la portabilité, …
- Outre ces deux qualités liées au logiciel, il existe la qualité liée au coût (Business value, ROI : retour sur investissement - Si je ne gagne rien, le logiciel n'est pas de bonne qualité, …) et la qualité liée au processus de développement (Est-ce un développement professionnel ? Est-ce que toutes les étapes de développement sont menées avec succès ? …).
- Remarque : Ces qualités sont parfois contradictoires et il est difficile de les satisfaire toutes. Il faut les pondérer selon les circonstances (logiciel critique/logiciel grand public, …). Il faut aussi distinguer les systèmes sur mesure et les produits logiciels de grande diffusion. Un logiciel de qualité n'est pas un logiciel sans erreur, ni un logiciel contenant des éléments pour faire plaisir au client/utilisateur, ni celui contenant ce qui est à la mode.

Cycle de vie du logiciel

- De par sa définition, Le Cycle de vie du développement d'un logiciel est un ensemble d'étapes de la réalisation, de l'énoncé des besoins à la maintenance au retrait du logiciel sur le marché informatique. De façon générale, on peut dire que le cycle de vie du logiciel est la période de temps s'étalant du début à la fin du processus du logiciel. Il commence donc avec a proposition ou la décision de développer un logiciel et se termine avec sa mise hors service.
- L'origine de ce découpage provient du constat que les erreurs ont un coût d'autant plus élevé qu'elles sont détectées tardivement dans le processus de réalisation. C'est ainsi que l'objectif principal du cycle de vie du développement d'un logiciel est de permettre la détection des erreurs au plus tôt et par conséquent, maîtriser la qualité du produit, les délais de sa réalisation et les coûts associés

Cycle de vie du logiciel

- Le développement de logiciel impose d'effectuer un certain nombre d'étapes. Il existe de nombreux modèles de cycle de vie du développement d'un logiciel, les plus courants comportent les phases suivantes :
- La communication ;
- La collection des exigences ;
- L'étude du préalable (faisabilité ou opportunité) ;
- Définition et analyse de besoins (Spécification) ;
- La conception du logiciel ;
- Le codage ;
- Les tests ;
- L'intégration ;
- L'installation ;
- La maintenance ;
- La disposition.



Cycle de vie du logiciel

La communication

- C’est la première étape du cycle de vie de développement logiciel où l’utilisateur initie la demande du produit logiciel souhaité. Ce dernier contact le service du fournisseur et essaie de négocier les termes du contrat. Il soumet alors sa demande à l’entreprise du fournisseur des services par écrit.

Cycle de vie du logiciel

La collection des exigences

- Cette étape fait avancer le travail de l'équipe du développement du logiciel jusqu'à la visibilité du projet informatique. A ce niveau, l'équipe de développement discute avec certains partenaires des divers problèmes du domaine et essaie de proposer autant d'informations possibles sur les différentes exigences. A ce niveau, les exigences sont considérées et isolées en fonction des besoins des utilisateurs ; les exigences du système et les exigences fonctionnelles. Les exigences sont collectées en utilisant un nombre donné des pratiques telles que :
  - Etude du système ou logiciel existant ou obsolète ;
  - Conduite des interviews auprès des utilisateurs et développeurs ;
  - Référencer aux différentes bases de données ;
  - Et la collection des réponses au moyen de questionnaires.

Cycle de vie du logiciel

L'étude du préalable

- Le développement est précédé d'une étude d'opportunité ou étude préalable. Cette phase a comme objectif de répondre aux questions suivantes : Pourquoi développer le logiciel ? Comment procéder pour faire ce développement ? Quels moyens faut-il mettre en œuvre ?
- Elle comprend à la fois des aspects techniques et de gestion. Parmi les tâches techniques, groupées sous le terme étude préalable, on peut citer :
  - Dresser un état de l'existant et faire une analyse de ses forces et faiblesses ;
  - Identifier les idées ou besoins de l'utilisateur ;
  - Formuler des solutions potentielles ;
  - Faire des études de faisabilité ;
  - Planifier la transition entre l'ancien logiciel et le nouveau, s'il y a lieu ;
  - Affiner ou finaliser l'énoncé des besoins de l'utilisateur

Cycle de vie du logiciel

La définition et analyse du besoin

- Lors de la phase d'analyse, également appelée phase de spécification (requirements phase, analysis phase, definition phase), on analyse les besoins de l'utilisateur ou du système englobant et on définit ce que le logiciel devra faire. Le résultat de la phase d'analyse est consigné dans un document appelé « cahier des charges du logiciel ou spécification du logiciel », en anglais : « software requirements, software specification ou requirements specification ». Il est essentiel qu'une spécification ne définisse que les caractéristiques essentielles du logiciel pour laisser de la place aux décisions de conception (ne pas faire de choix d'implémentation à ce niveau). Une spécification comporte les éléments suivants :
  - description de l'environnement du logiciel ;
  - spécification fonctionnelle (functional specification), qui définit toutes les fonctions que le logiciel doit offrir ;
  - comportement en cas d'erreurs, c'est-à-dire dans les cas où le logiciel ne peut pas accomplir une fonction ;
  - performances requises (performance requirements), par exemple : temps de réponse, encombrement en mémoire, sécurité de fonctionnement ;
  - les interfaces avec l'utilisateur (user interface), en particulier le dialogue sur terminal, la présentation des écrans, la disposition des états imprimés, etc.
  - interfaces avec d'autres logiciels et le matériel ;
  - contraintes de réalisation, telles que l'environnement de développement, le langage de programmation à utiliser, les procédures et normes à suivre, etc.

Cycle de vie du logiciel

La définition et analyse du besoin

- Il est judicieux de préparer pendant la phase d'analyse les procédures qui seront mises en œuvre pour vérifier que le logiciel, une fois construit, est conforme à la spécification, que nous l'appellerons test de réception (acceptance test). Durant la phase d'analyse, on produit également une version provisoire des manuels d'utilisation et d'exploitation du logiciel. Les Points clés :
- Pour les gros systèmes, il est difficile de formuler une spécification définitive. C'est pourquoi on supposera que les besoins initiaux du système sont incomplets et inconsistants.
- La définition des besoins et la spécification des besoins constituent des moyens de description à différents niveaux de détails, s'adressant à différents lecteurs.
- La définition des besoins est un énoncé, en langue naturelle, des services que le système est sensé fournir à l'utilisateur. Il doit être écrit de manière à être compréhensible par les décideurs côté client et côté contractant, ainsi que par les utilisateurs et acheteurs potentiels du système.
- La spécification des besoins est un document structuré qui énonce les services de manière plus détaillée. Ce document doit être suffisamment précis pour servir de base contractuelle entre le client et le fournisseur du logiciel. On peut utiliser des techniques de spécification formelle pour rédiger un tel document, mais cela dépendra du bagage technique du client.
- Il est difficile de détecter les inconsistances ou l'incomplétude d'une spécification lorsqu'elle est décrite dans un langage naturel non structuré. On doit toujours imposer une structuration du langage lors de la définition des besoins.
- Les besoins changent inévitablement. Le cahier des charges doit donc être conçu de manière à être facilement modifiable

Cycle de vie du logiciel

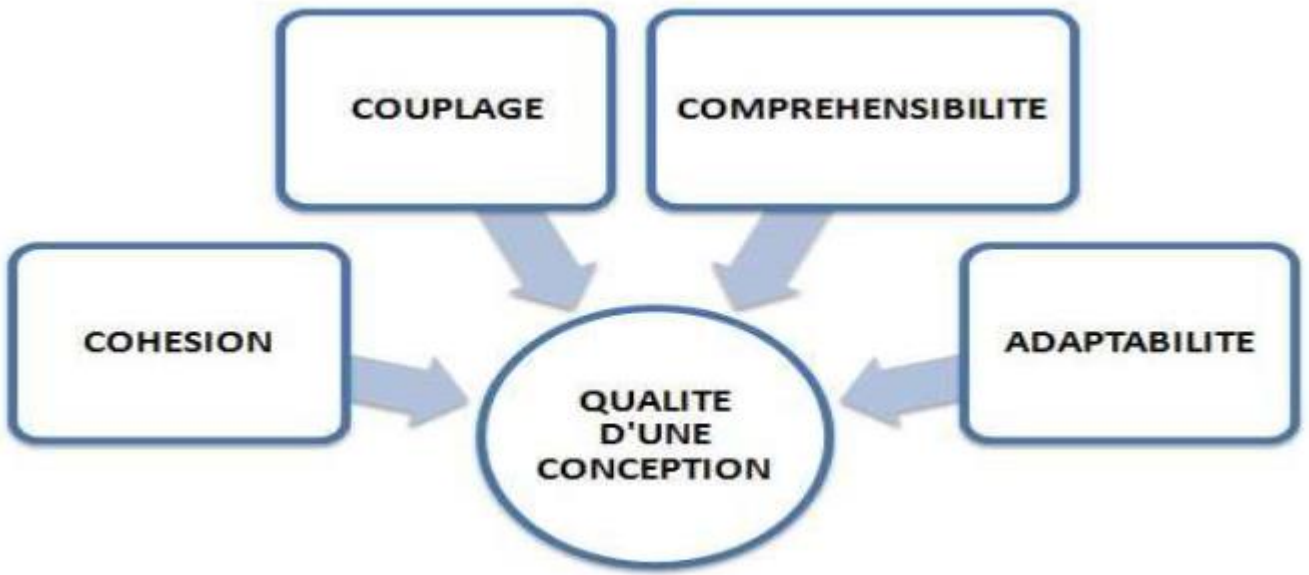
La conception du logiciel

- La phase d'analyse est suivie de la phase de conception, généralement décomposée en deux phases successives :
  - Conception générale ou conception architecturale (preliminary design ou architectural design) : Si nécessaire, il faut commencer par l'ébauche de plusieurs variantes de solutions et choisir celle qui offre le meilleur rapport entre coûts et avantages. Il faut ensuite figer la solution retenue, la décrire et la détailler. En particulier, il faut décrire l'architecture de la solution, c'est-à-dire son organisation en entités, les interfaces de ces entités et les interactions entre ces entités. Ce processus de structuration doit être poursuivi jusqu'à ce que tous les éléments du document de spécification aient été pris en compte. Le résultat de cette démarche est « un document de conception générale ». Durant la phase de conception générale, il faut également préparer la phase d'intégration. A cet effet, il faut élaborer un plan d'intégration, y compris un plan de test d'intégration.
  - Conception détaillée (detailed design) : La conception détaillée affine la conception générale. Elle commence par décomposer les entités découvertes lors de la conception générale en entités plus élémentaires. Cette décomposition doit être poursuivie jusqu'au niveau où les entités sont faciles à implémenter et à tester, c'est-à-dire correspondent à des composants logiciels élémentaires. Ce niveau dépend fortement du langage de programmation retenu pour l'implémentation. Il faut ensuite décrire chaque composant logiciel en détail : son interface, les algorithmes utilisés, le traitement des erreurs, ses performances, etc. L'ensemble de ces descriptions constitue le document de conception détaillée. Pendant la conception détaillée, il faut également préparer la vérification des composants logiciels élémentaires qui fera l'objet de la phase des tests unitaires. Le résultat est consigné dans un document appelé plan de tests unitaires. Si nécessaire, il faut de plus compléter le plan d'intégration, car de nouvelles entités ont pu être introduites pendant la conception détaillée.

Cycle de vie du logiciel

La qualité d'une conception logicielle

- La composante la plus importante de la qualité d'une conception est la maintenabilité. C'est en maximisant la cohésion à l'intérieur des composants et en minimisant le couplage entre ces composants que l'on parviendra à une conception maintenable



Cycle de vie du logiciel

La qualité d'une conception logicielle

A. La cohésion

- La cohésion d'un composant permet de mesurer la qualité de sa structuration, autrement dit, c'est un ensemble des forces solidaires développées à l'intérieur d'un logiciel. Un composant devrait implémenter une seule fonction logique ou une seule entité logique. La cohésion est une caractéristique désirable car elle signifie que chaque unité ne représente qu'une partie de la résolution du problème.

Cycle de vie du logiciel

La qualité d'une conception logicielle

A. La cohésion

- La cohésion fonctionnelle : Ici, Le module assure une seule fonction et tous les éléments du composant contribuent à atteindre un seul objectif (si un élément est supprimé, l‟objectif ne sera pas atteint).
- La cohésion séquentielle : Dans ce type de cohésion, la sortie d'un élément est utilisée en entrée d'un autre (dans ce cas, l‟ordre des actions est important)
- La cohésion de communication : Lorsque tous les éléments d'un composant travaillent sur les mêmes données.
- La cohésion procédurale : Dans ce cas, les éléments d'un composant constituent une seule séquence de contrôle.
- La cohésion temporelle : On parle de cohésion temporelle quand dans un même composant sont regroupés tous les éléments qui sont activés au même moment, par exemple, à l'initialisation d'un programme ou encore en fin d'exécution.
- La cohésion logique : Tous les éléments d'un composant effectuent des opérations semblables comme, par exemple, module qui édite tous les types de transactions et difficile à modifier.
- La cohésion occasionnelle : Le découpage en modules conduit à ce qu'une fonction se retrouve assurée par plusieurs modules. Dans ce cas, il n'y a pas de relation entre les éléments du composant.

Cycle de vie du logiciel

La qualité d'une conception logicielle

B. Le couplage

- C'est la situation de deux ou plusieurs états dont les impératifs de défense sont assurés par l'intégration de leurs forces respectives dans le cadre d'une stratégie commune. Le couplage est relatif à la cohésion. C'est une indication de la force d'interconnexion des différents composants d'un système. En règle générale, des modules sont fortement couplés lorsqu'ils utilisent des variables partagées ou lorsqu'ils échangent des informations de contrôle.

C. L'adaptabilité

- Si l'on doit maintenir une conception, cette dernière doit être facilement adaptable. Bien sûr, il faut pour cela que les composants soient faiblement couplés. En plus de ça, la conception doit être bien documentée, la documentation des composants doit être facilement compréhensible et consistante avec l'implémentation, cette dernière devant elle aussi être écrite de manière lisible.

Cycle de vie du logiciel

La qualité d'une conception logicielle

D. La compréhensibilité

- Pour modifier un composant dans une conception, il faut que celui qui est responsable de cette modification comprenne l'opération effectuée par ce composant.
- Cette compréhensibilité dépend des caractéristiques :
  - La cohésion : Le composant peut-il être compris sans que l'on fasse référence à d'autres composants?
  - L'appellation : Les noms utilisés dans le composant sont-ils significatifs ? Des noms significatifs reflètent les noms des entités du monde réel que l'on modélise.
  - La documentation : Le composant est-il documenté de manière à ce que l'on puisse établir une correspondance claire entre le monde réel et le composant ? Est-ce que cette correspondance est résumée quelque part.
  - La complexité : Les algorithmes utilisés pour implémenter le composant sont-ils complexes ?

Cycle de vie du logiciel

Le codage

- Après la conception détaillée, on peut passer à la phase de codage, également appelée phase de construction, phase de réalisation ou phase d'implémentation (implémentation phase, construction phase, coding phase). Lors de cette phase, la conception détaillée est traduite dans un langage de programmation.

Cycle de vie du logiciel

Les tests

- La phase d'implémentation est suivie de la phase de test (test phase). Durant cette phase, les composants du logiciel sont évalués et intégrés, et le logiciel lui-même est évalué pour déterminer s'il satisfait la spécification élaborée lors de la phase d'analyse. Cette phase est en général subdivisée en plusieurs phases.
- Lors des tests unitaires, on évalue chaque composant individuellement pour s'assurer qu'il est conforme à la conception détaillée. Si ce n'est déjà fait, il faut élaborer pour chaque composant un jeu de données de tests. Il faut ensuite exécuter le composant avec ce jeu, comparer les résultats obtenus aux résultats attendus, et consigner le tout dans le document des tests unitaires. S'il s'avère qu'un composant comporte des erreurs, il est renvoyé à son auteur, qui devra diagnostiquer la cause de l'erreur puis corriger le composant. Le test unitaire de ce composant est alors à reprendre.

Cycle de vie du logiciel

L'intégration

- Après avoir effectué avec succès la phase des tests de tous les composants, on peut procéder à leur assemblage, qui est effectué pendant la phase d'intégration (intégration phase). Pendant cette phase, on vérifie également la bonne facture des composants assemblés, ce qu'on appelle le test d'intégration (intégration test). On peut donc distinguer les actions suivantes :
  - construire par assemblage un composant à partir de composants plus petits ;
  - exécuter les tests pour le composant assemblé et enregistrer les résultats ;
  - comparer les résultats obtenus aux résultats attendus ;
  - si le composant n'est pas conforme, engager la procédure de modification ;
  - si le composant est conforme, rédiger les compte-rendus du test d'intégration et archiver sur support informatique les sources, objets compilés, images exécutables, les jeux de tests et leurs résultats.

Cycle de vie du logiciel

L'installation

- Après avoir intégré le logiciel, on peut l'installer dans son environnement d'exploitation, ou dans un environnement qui simule cet environnement d'exploitation, et le tester pour s'assurer qu'il se comporte comme requis dans la spécification élaborée lors de la phase d'analyse.

Cycle de vie du logiciel

La maintenance

- Après l'installation suit la phase d'exploitation et de maintenance (operation and maintenance phase). Le logiciel est maintenant employé dans son environnement opérationnel, son comportement est surveillé et, si nécessaire, il est modifié. Cette dernière activité s'appelle la maintenance du logiciel (software maintenance).
- Il peut être nécessaire de modifier le logiciel pour corriger des défauts, pour améliorer ses performances ou autres caractéristiques, pour adapter le logiciel à un nouvel environnement ou pour répondre à des nouveaux besoins ou à des besoins modifiés. On peut donc distinguer entre la maintenance corrective, la maintenance perfective et la maintenance adaptative. Sauf pour des corrections mineures, du genre dépannage, la maintenance exige en fait que le cycle de développement soit réappliqué, en général sous une forme simplifiée

Cycle de vie du logiciel

La maintenance

- Voici les différentes formes de maintenance qui peuvent exister en génie logiciel :
  - La **maintenance corrective** : c'est une maintenance qui corrige les erreurs et les défauts d'utilité, d'utilisabilité, de fiabilité... cette maintenance Identifie également les défaillances, et les dysfonctionnements en localisant la partie du code responsable. Elle Corrige et estime l'impact d'une modification. Attention, la plupart des corrections introduisent de nouvelles erreurs et les coûts de correction augmentent exponentiellement avec le délai de détection. Bref, la maintenance corrective donne lieu à de nouvelles livraisons (release).
  - La **maintenance adaptative** : c'est une maintenance qui ajuste le logiciel pour qu'il continue à remplir son rôle compte tenu du l'évolution des environnements d'exécution, des fonctions à satisfaire et des conditions d'utilisation. C'est par exemple le changement de SGBD, de machine, de taux de TVA, an 2000 ...
  - La **maintenance perfective** et d'extension : c'est une maintenance qui sert à accroître et améliorer les possibilités du logiciel afin de donne lieu à de nouvelles versions. C'est par exemple ; les services offerts, l'interface utilisateur, les performances...

Cycle de vie du logiciel

La maintenance

- Une fois qu'une version modifiée du logiciel a été développée, il faut bien entendu la distribuer. De plus, il est en général nécessaire de fournir à l'exploitant du logiciel une assistance technique et un support de consultation. En résumé, on peut dire que la maintenance et le support du logiciel comprennent les tâches suivantes :
  - effectuer des dépannages pour des corrections mineures ;
  - réappliquer le cycle de développement pour des modifications plus importantes ;
  - distribuer les mises à jour ;
  - fournir l'assistance technique et un support de consultation ;
  - maintenir un journal des demandes d'assistance et de support.

Cycle de vie du logiciel

La disposition

- Comme en informatique, le temps est un facteur indispensable qui doit être surveillé quant à tout ce qui concerne le développement des produits informatiques. C'est ainsi qu'un logiciel peut subir une détérioration proportionnellement à sa performance. Il peut être complètement obsolète ou exigera une intense mise à jour. Cette phase inclut l'archivage des données et les exigences de composants logiciels ; la fermeture du système ; la disposition planifiée des activités et de la terminaison du système quant à la période appropriée de la terminaison du logiciel. Bref, cette phase concerne la gestion de versions.
- A un moment donné, on décide de mettre le logiciel hors service. Les tâches correspondantes sont accomplies durant la phase de retrait (retirement phase) et comprennent :
  - avertir les utilisateurs ;
  - effectuer une exploitation en parallèle du logiciel à retirer et de son successeur ;
  - arrêter le support du logiciel.
- Remarque : ces étapes ne doivent pas être vues comme se succédant les unes aux autres de façon linéaire. Il y a en général (et toujours) des retours sur les phases précédentes, en particulier si les tests ne réussissent pas ou si les besoins évoluent.

Le processus de développement du logiciel

- La modélisation de développement logiciel aide les développeurs à sélectionner la stratégie pour développer le produit logiciel. Ainsi, chaque modélisation de développement possède ses outils propres, ses méthodes et ses procédures qui permettent d'exprimer clairement et définissent le cycle de vie de développement du logiciel.
- Les modèles du cycle de vie du logiciel sont des « plans de travail » qui permettent de planifier le développement. Plus le logiciel à développer est complexe (taille, algorithmes) et critique, plus il est important de bien contrôler le processus de développement et plus les documents qui accompagnent le logiciel doivent être précis et détaillés.

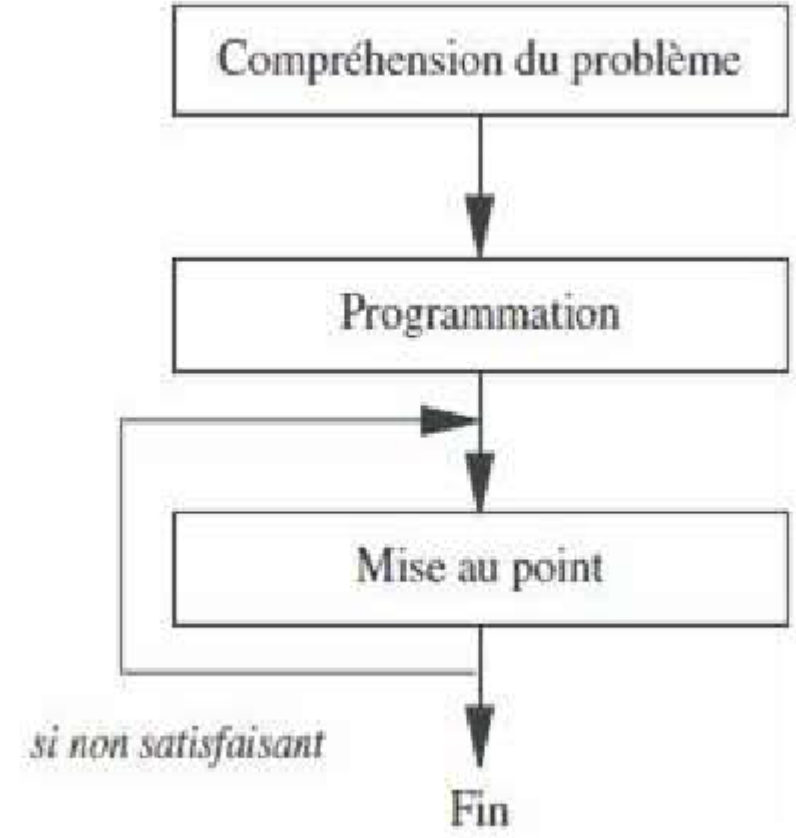
Le processus de développement du logiciel

- Il s'avère de nos jours, que nous ne pouvons plus avoir une démarche unique dans le développement de projets informatiques, mais qu'il faut construire le découpage temporel en fonction des caractéristiques de l'entreprise et du projet. On s'appuie pour cela sur des découpages temporels génériques, appelés modèles de développement (process models) ou modèles de cycle de vie d'un projet informatique. Les principaux modèles sont :
  - le modèle du code-and-fix ;
  - le modèle de la transformation automatique ;
  - le modèle de la cascade ;
  - le modèle en V ;
  - le modèle par incrément;
  - le modèle par prototypage ;
  - le modèle de la spirale.

Le processus de développement du logiciel

Le modèle du code-and-fix

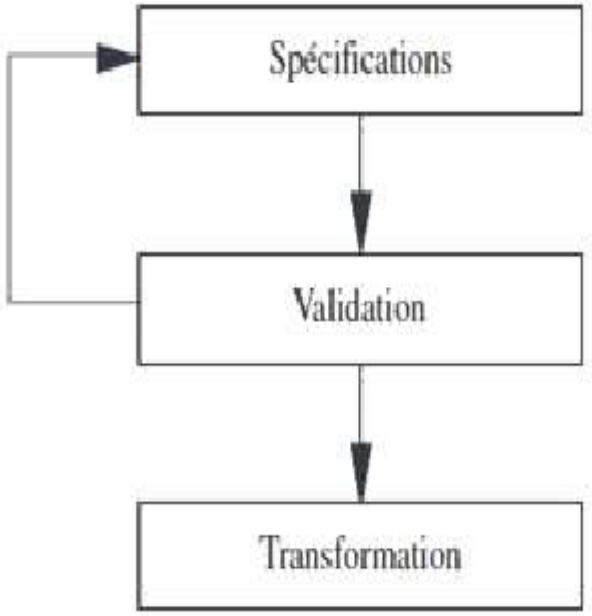
- C’est un modèle qui repose sur la possibilité d’une détermination facile des besoins : une première phase de compréhension du problème est suivie d’une phase de programmation ; puis une phase de mise au point, parfois en collaboration avec l'utilisateur du futur système, est répétée jusqu’à l’atteinte du résultat visé



Le processus de développement du logiciel

Le modèle de la transformation automatique

- C’est un modèle basé sur la possibilité de transformer automatiquement des spécifications en programmes. L’essentiel de l’effort va donc porter sur une description exhaustive des spécifications qui devront être complètement validées. Une succession de cycles phase de spécifications – phase de validation s’achève par la phase de transformation, où s’effectue la génération du code.



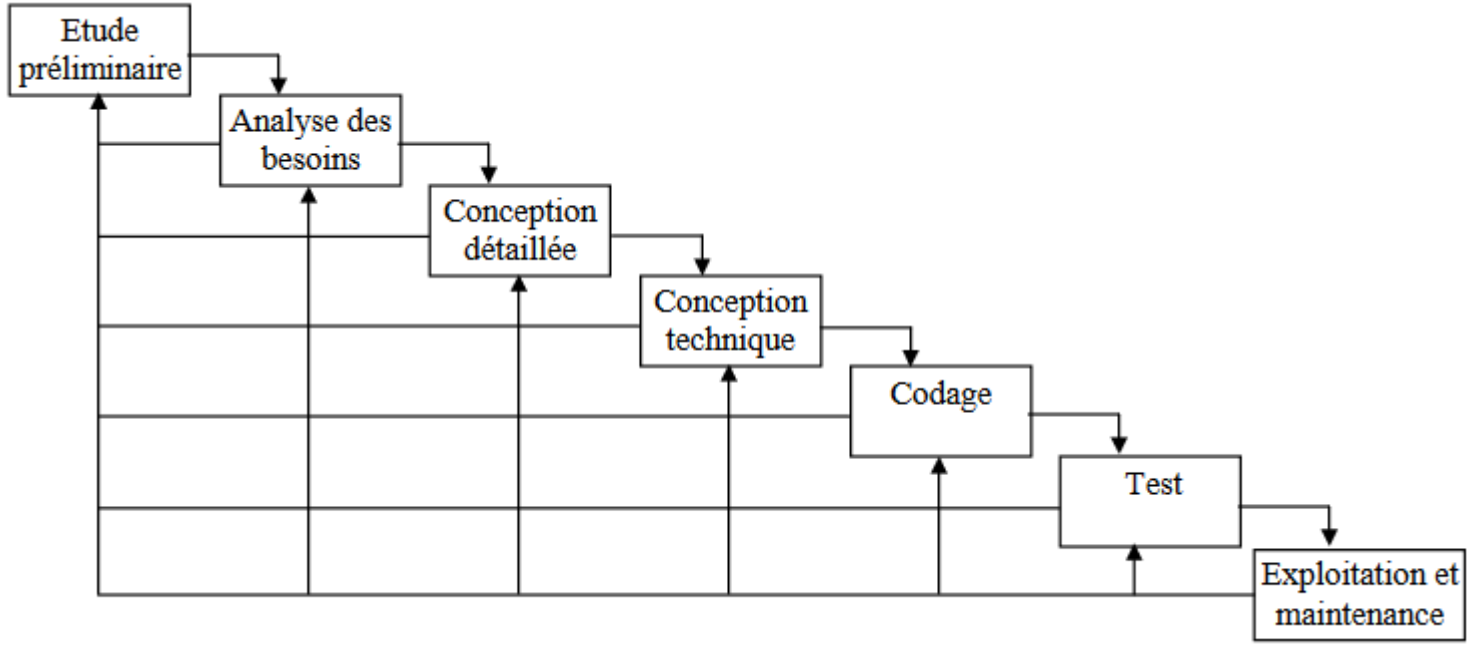
Le processus de développement du logiciel

Le modèle en cascade

- C'est un modèle qui a comme objectif majeur de jalonner (présenter sobrement) rigoureusement le processus de développement et de définir de façon précise les rôles respectifs du fournisseur qui produit un livrable et du client qui accepte ou refuse le résultat. Le découpage temporel se présente comme une succession de phases affinant celles du découpage classique : étude de faisabilité, définition des besoins, conception générale, conception détaillée, programmation, intégration, mise en œuvre. Chaque phase donne lieu à une validation officielle. Si le résultat du contrôle n'est pas satisfaisant, on modifie le livrable. En revanche, il n'y a pas de retour possible sur les options validées à l'issue de phases antérieures.
- Le cycle de vie dit de la « cascade » date de 1970. Ce cycle de vie est linéaire (présentoir) sans aucune évaluation entre le début du projet et la validation. Ici, Le projet est découpé en phases successives dans le temps et à chaque phase correspond une activité principale bien précise produisant un certain nombre de livrables et on ne passe à l'étape suivante que si les résultats de l'étape précédente sont jugés satisfaisants.
- L'activité d'une étape se réalise avec les résultats fournis par l'étape précédente ; ainsi, chaque étape sert de contrôle du travail effectué lors de l'étape précédente et chaque phase ne peut remettre en cause que la phase précédente ce qui, dans la pratique, s'avère insuffisant. L'élaboration des spécifications est une phase particulièrement critique : les erreurs de spécifications sont généralement détectées au moment des tests, voire au moment de la livraison du logiciel à l'utilisateur. Leur correction nécessite alors de reprendre toutes les phases du processus. Ce modèle est mieux adapté aux petits projets ou à ceux dont les spécifications sont bien connues et fixes. Dans le modèle en cascade, on effectue les différentes étapes du logiciel de façon séquentielle.

Le processus de développement du logiciel

Le modèle en cascade



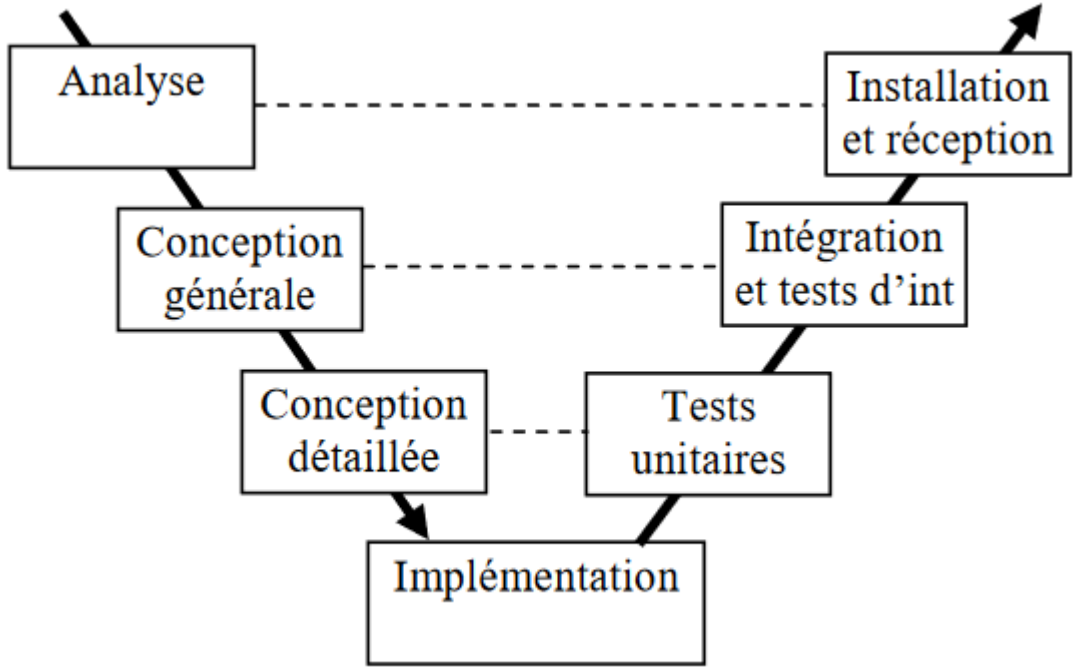
Le processus de développement du logiciel

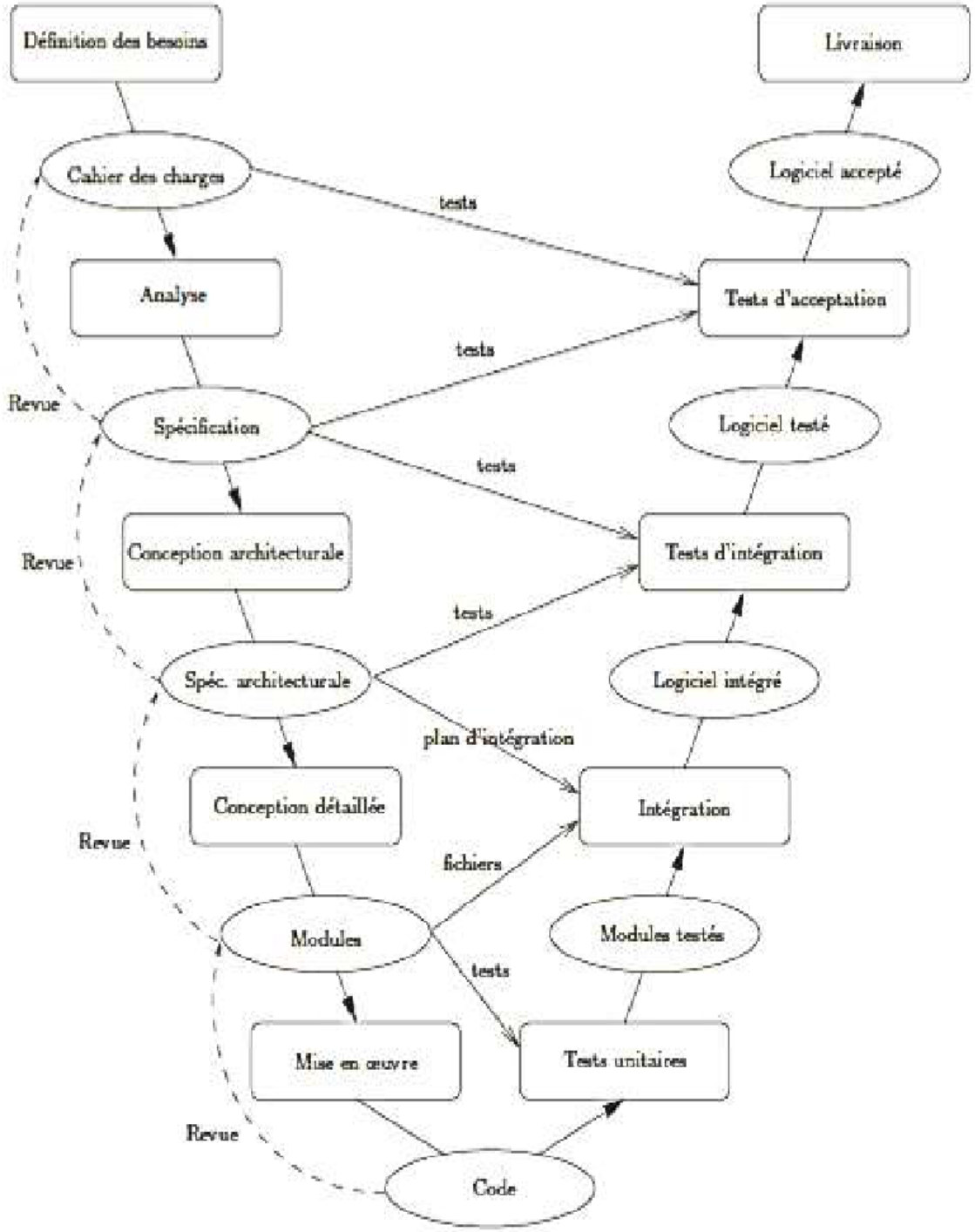
Le modèle en V

- Le modèle en V du cycle de vie du logiciel précise la conception des tests : (les tests système sont préparés à partir de la spécification; les tests d'intégration sont préparés à partir de la conception architecturale ; les tests unitaires sont préparés à partir de la conception détaillée des composants). Dérivé du modèle de la cascade, le modèle en V du cycle de développement montre non seulement l'enchaînement des phases successives, mais aussi les relations logiques entre phases plus éloignées. Ce modèle fait apparaître le fait que le début du processus de développement conditionne ses dernières étapes. Le modèle du cycle de vie en V est souvent adapté aux projets de taille et de complexité moyenne.
- La première branche correspond à un modèle en cascade classique. Toute description d'un composant est accompagnée de définitions de tests. Avec les jeux de tests préparés dans la première branche, les étapes de la deuxième branche peuvent être mieux préparées et planifiées. La seconde branche correspond à des tests effectifs effectués sur des composants réalisés. L'intégration est ensuite réalisée jusqu'à l'obtention du système logiciel final. L'avantage d'un tel modèle est d'éviter d'énoncer une propriété qu'il est impossible de vérifier objectivement une fois le logiciel réalisé. Le cycle en V est le cycle qui a été normalisé, il est largement utilisé, notamment en informatique industrielle et en télécommunication. Ce modèle fait également apparaitre les documents qui sont produits à chaque étape, et les «revues» qui permettent de valider les différents produits.

Le processus de développement du logiciel

Le modèle en V




Définition des besoins
Livraison
Cahier des charges
Logiciel accepté
tests
Analyse
Tests d'acceptation
tests
Revue
Spécification
Logiciel testé
tests
Conception architecturale
Tests d'intégration
Revue
tests
Spéc. architecturale
Logiciel intégré
plan d'intégration
Conception détaillée
Intégration
Revue
fichiers
Modules
Modules testés
tests
Mise en œuvre
Tests unitaires
Revue
Code

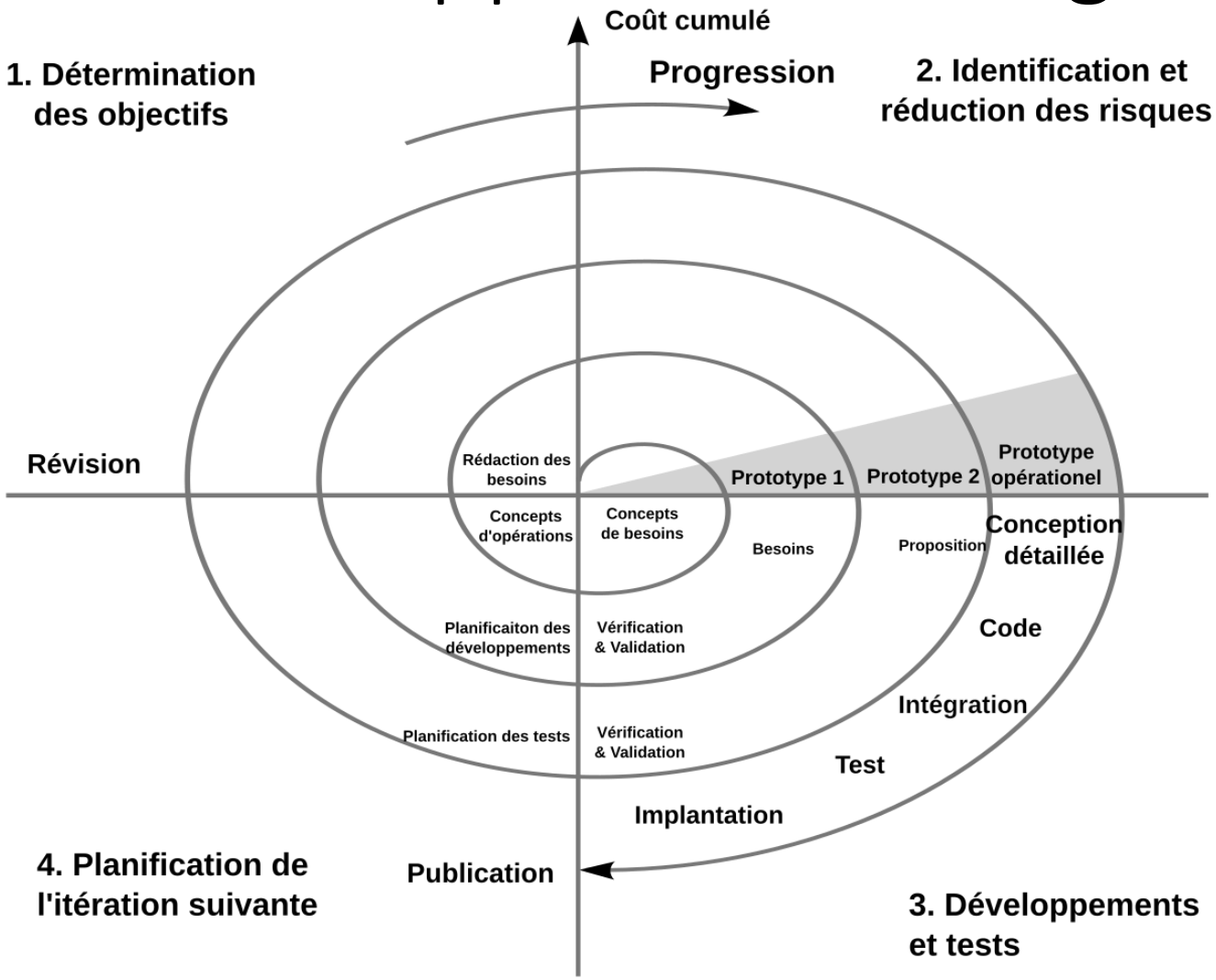
Le processus de développement du logiciel

Le modèle en spirale

- Ce modèle repose sur le même principe que le modèle évolutif, mais il s'inscrit dans une relation contractuelle entre le client et le fournisseur. De ce fait les engagements et validations présentent un caractère formalisé. Chaque cycle donne lieu à une contractualisation préalable, s'appuyant sur les besoins exprimés lors du cycle précédent. Un cycle comporte six phases :
  - Analyse du risque ;
  - Développement d'un prototype (modèle, archétype…) ;
  - Simulation et essais du prototype ;
  - Détermination des besoins à partir des résultats des essais ;
  - Validation des besoins par un comité de pilotage ;
  - Planification du cycle suivant.

Le processus de développement du logiciel

Le modèle en spirale



Le processus de développement du logiciel

Le modèle en spirale

- Pour corriger les travers de la démarche en cascade sont apparus des modèles dits en spirales où les risques, quels qu'ils soient, sont constamment traités au travers de bouclages successifs :
  - chaque spire confirme et affine les spires précédentes en menant des activités de même nature successivement ;
  - l'analyse ou la conception ne sont plus effectuées dans une seule phase ou étape mais sont conduites en tant qu'activités qui se déroulent sur de multiples phases ;
  - à chaque étape, après avoir défini les objectifs et les alternatives, celles-ci sont évaluées par différentes techniques (prototypage, simulation, ...), l'étape est réalisée et la suite est planifiée.
  - le nombre de cycles est variable selon que le développement est classique ou incrémental ;
  - ce modèle met l'accent sur l'analyse des risques tels que les exigences démesurées par rapport à la technologie, la réutilisation de composants, calendriers et budgets irréalistes, la défaillance du personnel, etc.
  - ce modèle est utilisé pour des projets dont les enjeux (risques) sont importants

Le processus de développement du logiciel

Le modèle en spirale

- Chaque cycle de la spirale se déroule en quatre phases :
  - 1. Un cycle de la spirale commence par l'élaboration d'objectifs tels que la performance, la fonctionnalité, etc. on énumère ensuite les différentes manières de parvenir à ces objectifs, ainsi que les contraintes. On évalue ensuite chaque alternative en fonction de l'objectif.
  - 2. L'étape suivante consiste à évaluer les risques pour chaque activité, comme l'analyse détaillée, le prototypage, la simulation, etc.
  - 3. Après avoir évalué le risque, on choisit un modèle de développement pour le système. Par exemple, si les principaux risques concernent l'interface utilisateur, le prototypage évolutif pourrait s'avérer un modèle de développement approprié. Le modèle de la cascade peut être le plus approprié si le principal risque identifié concerne l'intégration des sous-systèmes. Il n'est pas nécessaire d'adopter un seul modèle à chaque cycle de la spirale ou même pour l'ensemble d'un système. Le modèle de la spirale englobe tous les autres modèles.
  - 4. La situation est ensuite réévaluée pour déterminer si un développement supplémentaire est nécessaire, auquel cas il faudrait planifier la prochaine étape. (on estime au cours d'une procédure de revue, si on doit passer au prochain cycle de la spirale ou non).

Le processus de développement du logiciel

Le modèle en spirale

- Il n'est pas nécessaire d'adopter un seul modèle à chaque cycle de la spirale ou même pour l'ensemble d'un système. Le modèle de la spirale englobe tous les autres modèles. Le prototypage peut être utilisé dans une spirale pour résoudre le problème de la spécification des besoins, puis il peut être suivi d'un développement basé sur le modèle conventionnel de la cascade. On peut utiliser la transformation formelle pour une partie du système à haute sécurité, et une approche basée sur la réutilisation pour l'interface utilisateur.
- Le modèle du cycle de vie en spiral est un modèle itératif (répété), où la planification de la version se fait selon une analyse de risques. L'idée est de s'attaquer aux risques les plus importants assez tôt, afin que ceux-ci diminuent rapidement. De façon générale, les risques liés au développement de logiciels peuvent être répartis en quatre catégories :
  - les risques commerciaux (placement du produit sur le marché, concurrence);
  - les risques financiers (capacités financières suffisantes pour réaliser le produit);
  - les risques techniques (la technologie employée est-elle éprouvée ?) ;
  - les risques de développement (l'équipe est-elle suffisamment expérimentée ?).

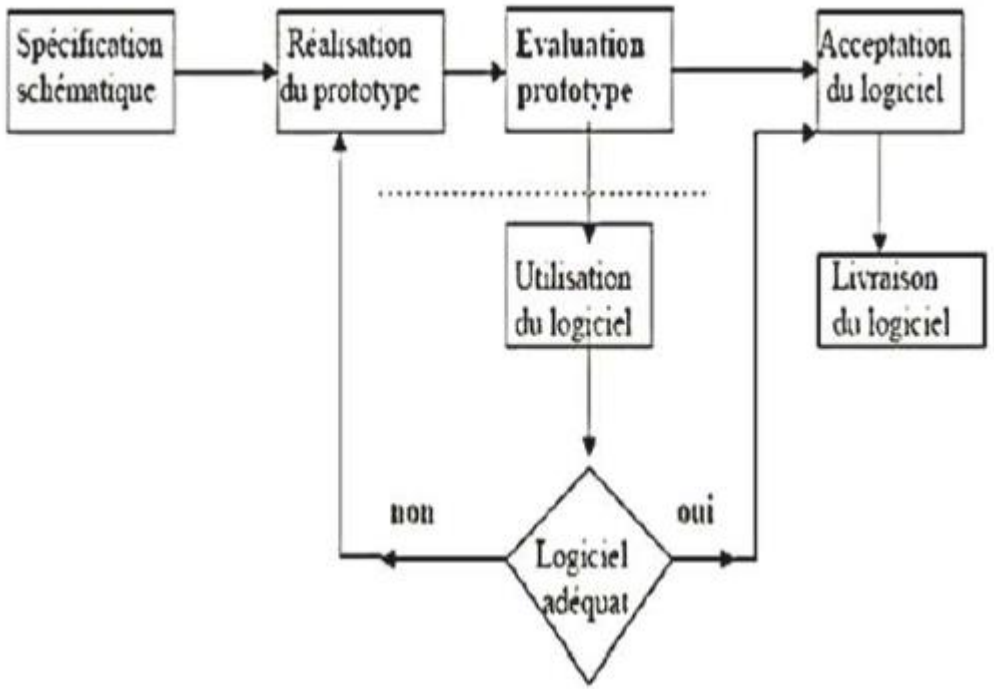
Le processus de développement du logiciel

Le modèle par prototypage

- Il est quelquefois difficile de formuler une esquisse des besoins, surtout lorsque l'on connaît peu le domaine. Dans ces cas-là, on ne peut pas espérer de manière réaliste définir les besoins de manière définitive avant le début du développement du logiciel.
- Un modèle de processus basé sur le prototypage se révèle alors plus approprié que le modèle classique de la cascade. Le prototypage permet de contourner la difficulté de la validation liée à l'imprécision des besoins et caractéristiques du système à développer. Cela veut dire que lorsqu'il est difficile d'établir une spécification détaillée, on a recours au prototypage qui est considéré, dans ce cas, comme un modèle de développement de logiciels.
- Il s'agit d'écrire une première spécification et de réaliser un sous-ensemble du produit logiciel final. Ce sous ensemble est alors raffiné incrémentalement et évalué jusqu'à obtenir le produit final. On en distinguera deux types de prototypage :
  - Le prototypage jetable : ici, le squelette du logiciel n'est créé que dans un but et dans une phase particulière du développement.
  - Le prototypage évolutif : ici, on conserve tout, au long du cycle de développement. Il est amélioré et complété pour obtenir le logiciel final

Le processus de développement du logiciel

Le modèle par prototypage



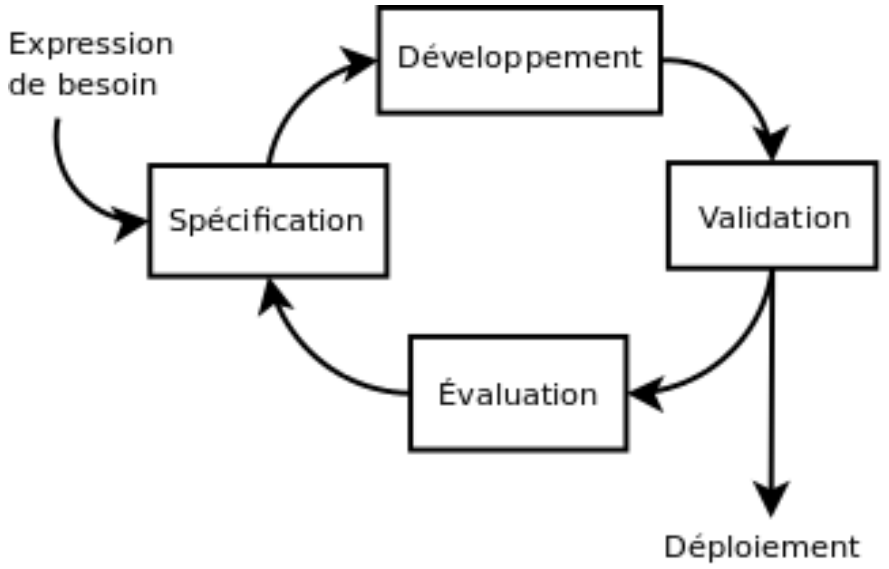
Le processus de développement du logiciel

Le modèle incrémental

- Le modèle incrémental est un modèle itératif, qui consiste à sélectionner successivement plusieurs incréments. Un incrément du logiciel est un sous-ensemble du logiciel complet, qui consiste en un petit nombre de fonctionnalités.
- Dans les modèles spirales, en V ou en cascade, un logiciel est décomposé en composants développés séparément et intégrés à la fin du processus. Dans le modèle par incréments, seul un sous ensemble est développé à la fois. Dans un premier temps un logiciel noyau est développés, puis successivement, les incréments sont développés et intégrés. Chaque incrément est développé selon l'un des modèles précédents. Dans ce modèle, les intégrations sont progressives et il peut y avoir des livraisons et des mises en service après chaque intégration d'incrément. Le modèle incrémental présente comme avantages :
  - chaque développement est moins complexe ;
  - les intégrations sont progressives ;
  - possibilité de livraisons et de mises en service après chaque incrément ;
  - meilleur lissage du temps et de l'effort de développement à cause de la possibilité de recouvrement des différentes phases. l'effort est constant dans le temps par opposition au pic pour spécifications détaillées pour les modèles en cascade ou en V.
- Le modèle incrémental présente comme Risques
  - la remise en cause du noyau de départ ;
  - la remise en cause des incréments précédents;
  - ou encore l'impossibilité d'intégrer un nouvel incrément

Le processus de développement du logiciel

Le modèle incrémental



Exercice

- Pour la peinture des murs d’une pièce, on considère :
  - (1) les tâches suivantes : choisir la couleur, acheter la peinture, nettoyer les murs, préparer la peinture et peindre les murs ;
  - (2) les artefacts suivants : choix de la couleur, pots de peinture achetés, murs propres, peinture mélangée, murs peints.

Dessiner un modèle de processus

Exercice

- Une entreprise de production logiciel adopte un processus de développement logiciel qui consiste à enchainer les différentes phases de développement : étude de faisabilité, spécification, conception, implémentation, tests et livraison. Les retours en arrière entre ces différentes phases ne sont pas planifiés mais si des erreurs sont détectées pendant les tests, il est possible que l'équipe de développement réadapte la conception et/ou l'implémentation du logiciel. Le succès des projets de développement logiciel de cette entreprise est garanti seulement s'il s'agit de reproduire un projet déjà réalisé.
- Déterminez le modèle de cycle de vie utilisé par cette entreprise.
- Indiquer la ou les phases où est produit chacun des documents suivants :
  - Manuel d'utilisation
  - conception architecturale,
  - spécification des modules,
  - code source,
  - cahier de charges,
  - plan de test,
  - estimation des couts,
  - calendrier du projet,
  - documentation.

UML

- Pour faire face à la complexité croissante des systèmes d'information, de nouvelles méthodes et outils ont été créées. La principale avancée des quinze dernières années réside dans la programmation orientée objet (P.O.O.).
- Face à ce nouveau mode de programmation, les méthodes de modélisation classique (telle MERISE) ont rapidement montré certaines limites et ont dû s'adapter (cf. MERISE/2). De très nombreuses méthodes ont également vu le jour comme Booch, OMT …
- Issu du terrain et fruit d'un travail d'experts reconnus, UML est le résultat d'un large consensus. De très nombreux acteurs industriels de renom ont adopté UML et participent à son développement. En l'espace d'une poignée d'années seulement, UML est devenu un standard incontournable.

UML

- Il y donc déjà longtemps que l'approche orientée objet est devenue une réalité. Les concepts de base de l'approche objet sont stables et largement éprouvés. C'est un réflexe quasi-automatique dès lors qu'on cherche à concevoir des logiciels complexes qui doivent "résister" à des évolutions incessantes.
- Toutefois, l'approche objet comporte des défauts:
  - elle est moins intuitive que l'approche fonctionnelle.
  - malgré les apparences, il est plus naturel pour l'esprit humain de décomposer un problème informatique sous forme d'une hiérarchie de fonctions atomiques et de données, qu'en termes d'objets et d'interaction entre ces objets. Or, l'approche objet ne dicte pas comment modéliser la structure d'un système de manière pertinente. Comment dès lors mener une analyse qui respecte les concepts objet ? La dérive fonctionnelle de la conception est un risque...
  - l'application des concepts objet nécessite une très grande rigueur : le vocabulaire précis est un facteur d'échec important dans la mise en oeuvre d'une approche objet (risques d'ambiguïtés et d'incompréhensions). Beaucoup de développeurs (même expérimentés) ne pensent souvent objet qu'à travers un langage de programmation. Or, les langages orientés objet ne sont que des outils qui proposent une manière particulière d'implémenter certains concepts objet.
  - Enfin, comment comparer deux solutions de découpe objet d'un système si l'on ne dispose pas d'un moyen de représentation adéquat ? Il est très simple de décrire le résultat d'une analyse fonctionnelle, mais qu'en est-il d'une découpe objet ?
- Pour remédier à ces inconvénients majeurs de l'approche objet, il faut donc :
  - un langage qui permet la POO (Python)
  - une démarche d'analyse et de conception objet

UML

- Il est impossible de donner une représentation graphique complète d'un logiciel, ou de tout autre système complexe, mais il est possible de donner sur un système des vues partielles.
- UML 2.0 comporte ainsi treize types de diagrammes représentant autant de vues distinctes pour représenter des concepts particuliers du système d'information.
- Ces diagrammes, d'une utilité variable selon les cas, ne sont pas nécessairement tous produits à l'occasion d'une modélisation. Les plus utiles pour la maîtrise d'ouvrage sont les diagrammes d'activités, de cas d'utilisation, de classes, d'objets, de séquence et d'états-transitions. Les diagrammes de composants, de déploiement et de communication sont surtout utiles pour la maîtrise d'œuvre à qui ils permettent de formaliser les contraintes de la réalisation et la solution technique

UML

Diagramme de cas d'utilisation

- Le diagramme de cas d'utilisation représente la structure des grandes fonctionnalités nécessaires aux utilisateurs du système. C'est le premier diagramme du modèle UML, celui où s'assure la relation entre l'utilisateur et les objets que le système met en œuvre.

Diagramme de classes

- Le diagramme de classes est généralement considéré comme le plus important dans un développement orienté objet. Il représente l'architecture conceptuelle du système : il décrit les classes que le système utilise, ainsi que leurs liens, que ceux-ci représentent un emboîtage conceptuel (héritage) ou une relation organique (agrégation).

Diagramme d'objets

- Le diagramme d'objets permet d'éclairer un diagramme de classes en l'illustrant par des exemples. Il est, par exemple, utilisé pour vérifier l'adéquation d'un diagramme de classes à différents cas possibles.

UML

Diagramme d'états transition

- Le diagramme d'états-transitions représente la façon dont évoluent les objets appartenant à une même classe. La modélisation du cycle de vie est essentielle pour représenter et mettre en forme la dynamique du système.

Diagramme d'activités

- Le diagramme d'activités n'est autre que la transcription dans UML de la représentation du processus telle qu'elle a été élaborée lors du travail qui a préparé la modélisation : il montre l'enchaînement des activités qui concourent au processus.

Diagramme de séquence et de communication

- Le diagramme de séquence représente la succession chronologique des opérations réalisées par un acteur. Il indique les objets que l'acteur va manipuler et les opérations qui font passer d'un objet à l'autre. On peut représenter les mêmes opérations par un diagramme de communication, graphe dont les nœuds sont des objets et les arcs (numérotés selon la chronologie) les échanges entre objets. En fait, diagramme de séquence et diagramme de communication sont deux vues différentes mais logiquement équivalentes (on peut construire l'une à partir de l'autre) d'une même chronologie. Ce sont des diagrammes d'interaction.

Diagramme des cas d’utilisation

- Le but de ce ces diagrammes est de recueillir, d’analyser et d’organiser les besoins, et de recenser les grandes fonctionnalités d’un système. Il s’agit 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.

- 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 rôle inscrit dessous.

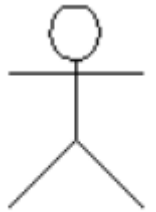
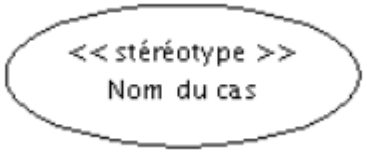


Diagramme des cas d'utilisation

- 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.



- Dans le cas où l'on désire présenter les attributs ou les opérations du cas d'utilisation, il est préférable de le représenter sous la forme d'un classeur stéréotypé « use case ». Nous reviendrons sur les notions d'attributs ou d'opération lorsque nous aborderons les diagrammes de classes et d'objets

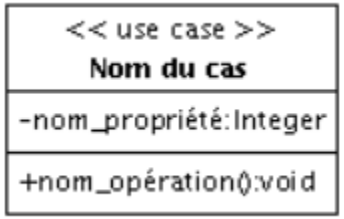


Diagramme des cas d'utilisation

- 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.
- Une **relation d'association** est chemin de communication entre un acteur et un cas d'utilisation et est représenté un trait continu

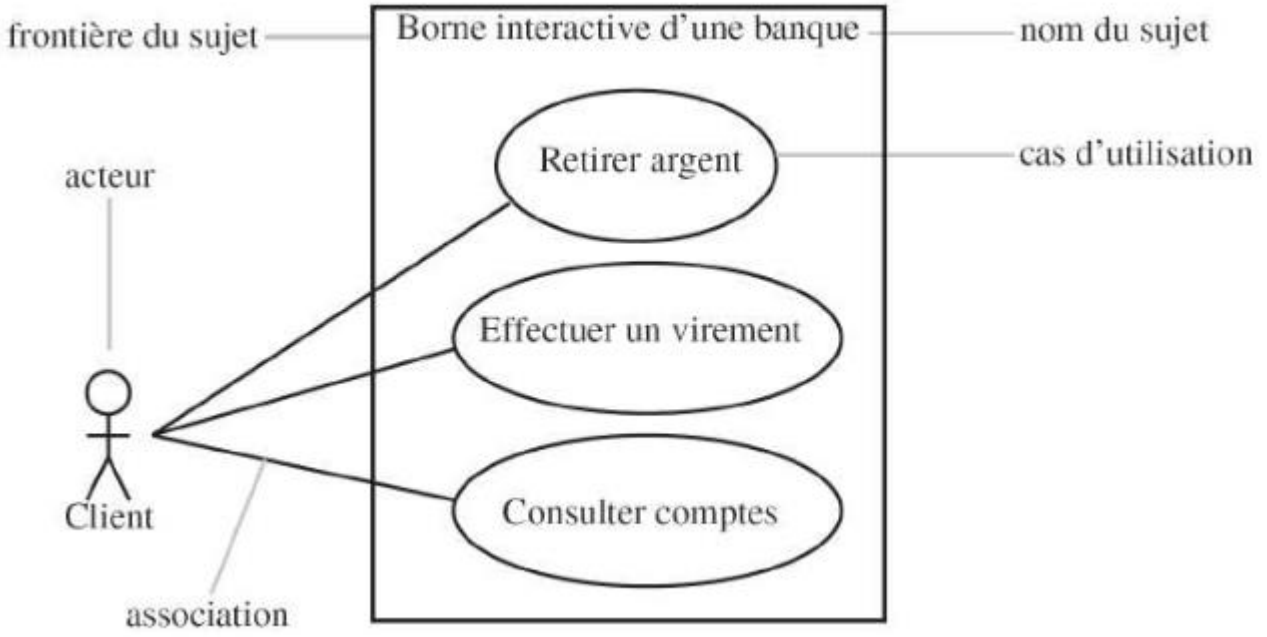
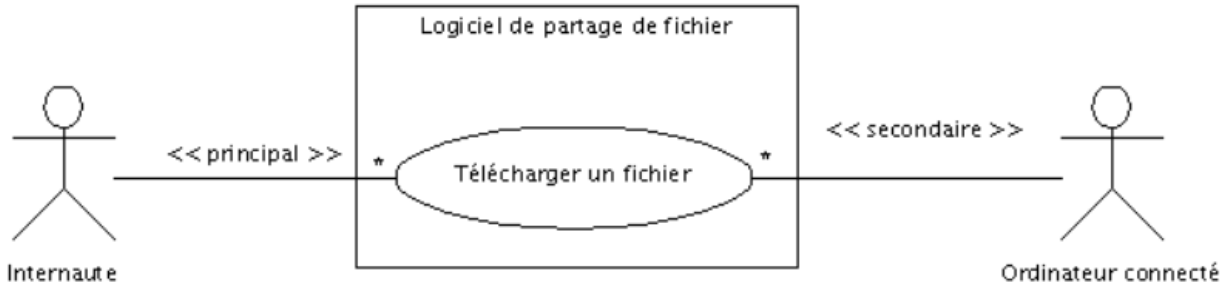


Diagramme des cas d'utilisation

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.



Multiplicité

- Lorsqu'un acteur peut interagir plusieurs fois avec un cas d'utilisation, il est possible d'ajouter une multiplicité sur l'association du côté du cas d'utilisation. Le symbole * signifie plusieurs, exactement n s'écrit tout simplement n, n..m signifie entre n et m, etc. Préciser une multiplicité sur une relation n'implique pas nécessairement que les cas sont utilisés en même temps. La notion de multiplicité n'est pas propre au diagramme de cas d'utilisation.

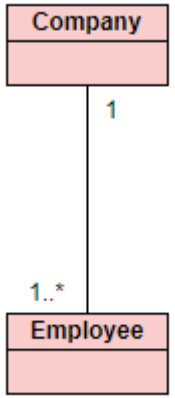


Diagramme des cas d'utilisation

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. 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 un flèche avec un trait pleins dont la pointe est un triangle fermé désignant le cas le plus général

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 ». 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.

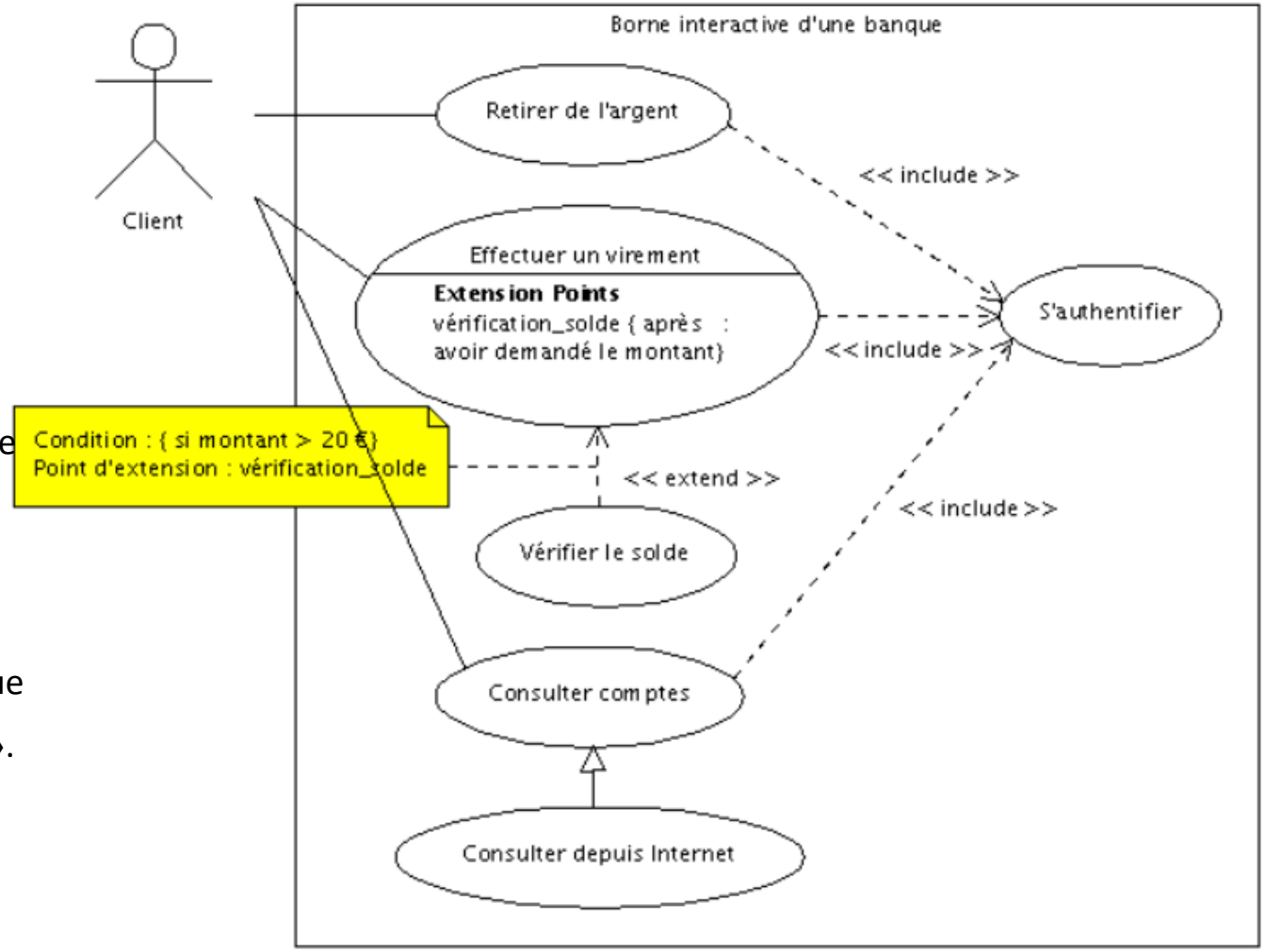


Diagramme des cas d'utilisation

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.

Relation de généralisation

- Un cas A est une généralisation d'un cas B si B est un cas particulier de A. 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.

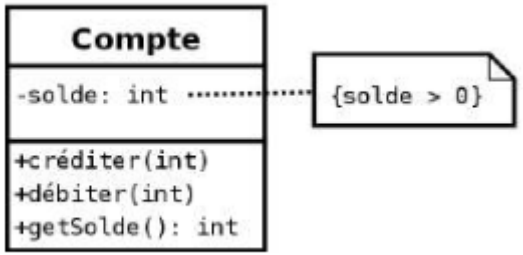
Relations entre acteurs

- 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).

Diagramme des cas d'utilisation

Note

- Une note contient une information textuelle comme un commentaire, un corps de méthode ou une contrainte. Graphiquement, elle est représentée par un rectangle dont l'angle supérieur droit est plié. Le texte contenu dans le rectangle n'est pas contraint par UML. Une note n'indique pas explicitement le type d'élément qu'elle contient, toute l'intelligibilité d'une note doit être contenu dans le texte même. On peut relier une note à l'élément qu'elle décrit grâce à une ligne en pointillés. Si elle décrit plusieurs éléments, on dessine une ligne vers chacun d'entre eux.



Stéréotype

- Un stéréotype est une annotation s'appliquant sur un élément de modèle. Il n'a pas de définition formelle, mais permet de mieux caractériser des variétés d'un même concept. Il permet donc d'adapter le langage à des situations particulières. Il est représenté par une chaînes de caractères entre guillemets (« ») dans, ou à proximité du symbole de l'élément de modèle de base.

Diagramme des cas d'utilisation

Comment identifier les acteurs ?

- Les acteurs d'un système sont les entités externes à ce système qui interagissent (saisie de données, réception d'information, . . .) avec lui. Il faut faire attention à ne pas confondre acteurs et utilisateurs (utilisateur avec le sens de la personne physique qui va appuyer sur un bouton) d'un système. D'une part parce que les acteurs inclus les utilisateurs humains mais aussi les autres systèmes informatiques ou hardware qui vont communiquer avec le système. D'autre part parce que un acteur englobe tout une classe d'utilisateur. Chaque acteur doit être nommé selon son rôle car un acteur représente un ensemble cohérent de rôles joués vis-à-vis du système. Pour trouver les acteurs d'un système, il faut identifier quels sont les différents rôles que vont devoir jouer ses utilisateurs (ex : responsable clientèle, responsable d'agence, administrateur, approbateur, . . .) et les autres systèmes.

Comment recenser les cas d'utilisation ?

- Chaque cas d'utilisation correspond à une fonction métier du système, selon le point de vue d'un de ses acteurs. Aussi, pour identifier les cas d'utilisation, il faut se placer du point de vue de chaque acteur et déterminer comment et surtout pourquoi il se sert du système. Nommez les cas d'utilisation avec un verbe à l'infinitif suivi d'un complément en vous plaçant du point de vue de l'acteur et non pas de celui du système. Par exemple, un distributeur de billets aura probablement un cas d'utilisation Retirer de l'argent et non pas Distribuer de l'argent. Il faut être vigilant pour ne pas retomber dans une décomposition fonctionnelle descendante hiérarchique. Un nombre trop important de cas d'utilisation est en général le symptôme de ce type d'erreur.
- Dans tous les cas, il faut bien garder à l'esprit qu'il n'y a pas de notion temporelle dans un diagramme de cas d'utilisation.

Exercice

- Un client a un compte dans une banque. Un distributeur automatique de billets permet de :
  - Consulter le solde de son compte
  - Déposer de l’argent sur son compte
  - Retirer de l’argent
- Un technicien vient régulièrement assurer la maintenance du distributeur. La banque le sollicite également en cas de panne.
- Donnez un diagramme de cas d'utilisation pour le distributeur de billets.

Exercice

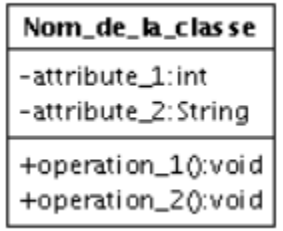
- Le déroulement normal d'utilisation d'une caisse enregistreuse est le suivant :
  - 1. Un client arrive à la caisse avec des articles
  - 2. Le caissier enregistre le numéro d'identification de chaque article, ainsi que la quantité si celle-ci est supérieure à 1
  - 5. La caisse affiche le prix de chaque article et son libellé
  - 6. Lorsque tous les articles ont été enregistrés, le caissier signale la fin de la vente
  - 7. La caisse affiche le total des achats
  - 8. Le client choisit son mode de paiement :
    - Liquide : le caissier encaisse l'argent et la caisse indique le montant éventuel à rendre au client
    - Chèque : le caissier note l'identité du client et la caisse enregistre le montant sur le chèque
    - Carte de crédit : un terminal bancaire fait partie de la caisse, il transmet la demande à un centre d'autorisation multi-banques
  - 9. La caisse enregistre la vente et imprime un ticket
  - 10. Le caissier transmet le ticket imprimé au client
  - 11. Un client peut présenter des coupons de réduction avant le paiement. Lorsque le paiement est terminé, la caisse transmet les informations relatives aux articles vendus au système de gestion des stocks.
  - 12. Tous les matins, le responsable du magasin initialise les caisses pour la journée.
- Donnez un diagramme de cas d'utilisation pour la caisse enregistreuse

Diagramme de classes

- 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. Les principaux éléments de cette vue statique sont les classes et leurs relations : association, généralisation et plusieurs types de dépendances, telles que la réalisation et l'utilisation.
- Une classe est la description formelle d'un ensemble d'objets ayant une sémantique et des propriétés communes. Dit autrement, un objet est une instance d'une classe. C'est une entité discrète dotée d'une identité, d'un état et d'un comportement que l'on peut invoquer. Les objets sont des éléments individuels d'un système en cours d'exécution.

Diagramme de classes

- Une classe est représentée par un rectangle divisé en trois à cinq compartiments :
  - Le nom
  - Les attributs
  - Les opérations
  - Les responsabilités de la classe (les tâches devant être assurées en attente d'informations)
  - Les exceptions



Les attributs

- L'encapsulation consiste à rassembler les données et les méthodes au sein d'une structure en cachant l'implémentation de l'objet, c'est-à-dire en empêchant l'accès aux données par un autre moyen que les services proposés. Ces services accessibles aux utilisateurs de l'objet définissent ce que l'on appel l'interface de l'objet (sa vue externe). L'encapsulation permet donc de garantir l'intégrité des données contenues dans l'objet. L'encapsulation permet de définir des niveaux de visibilité des éléments d'un conteneur :
  - public ou **+** : tout élément qui peut voir le conteneur peut également voir l'élément indiqué.
  - protected ou **#** : seul un élément situé dans le conteneur ou un de ses descendants peut voir l'élément indiqué.
  - private ou **-** : seul un élément situé dans le conteneur peut voir l'élément.
  - package ou ~ ou rien : seul un élément déclaré dans le même paquetage peut voir l'élément
- La multiplicité d'un attribut précise le nombre de valeurs que l'attribut peut contenir. Lorsqu'un multiplicité supérieure à 1 est précisée, il est possible d'ajouter une contrainte (<contrainte>) pour préciser si les valeurs sont ordonnées ({ordered}) ou pas ({list}).
- Par défaut, chaque instance d'une classe possède sa propre copie des attributs de la classe. Les valeurs des attributs peuvent donc différer d'un objet à un autre. Cependant, il est parfois nécessaire de définir un attribut de classe (static en Java ou en C++) qui garde une valeur unique et partagée par toutes les instances de la classe. Les instances ont accès à cet attribut mais n'en possèdent pas une copie. Un attribut de classe n'est donc pas une propriété d'une instance mais une propriété de la classe et l'accès à cet attribut ne nécessite pas l'existence d'une instance. Graphiquement, un attribut de classe est souligné.
- Les attributs dérivés peuvent être calculés à partir d'autres attributs et de formules de calcul. Lors de la conception, un attribut dérivé peut être utilisé comme marqueur jusqu'à ce que vous puissiez déterminer les règles à lui appliquer. Les attributs dérivés sont symbolisés par l'ajout d'un « / » devant leur nom

Diagramme de classes

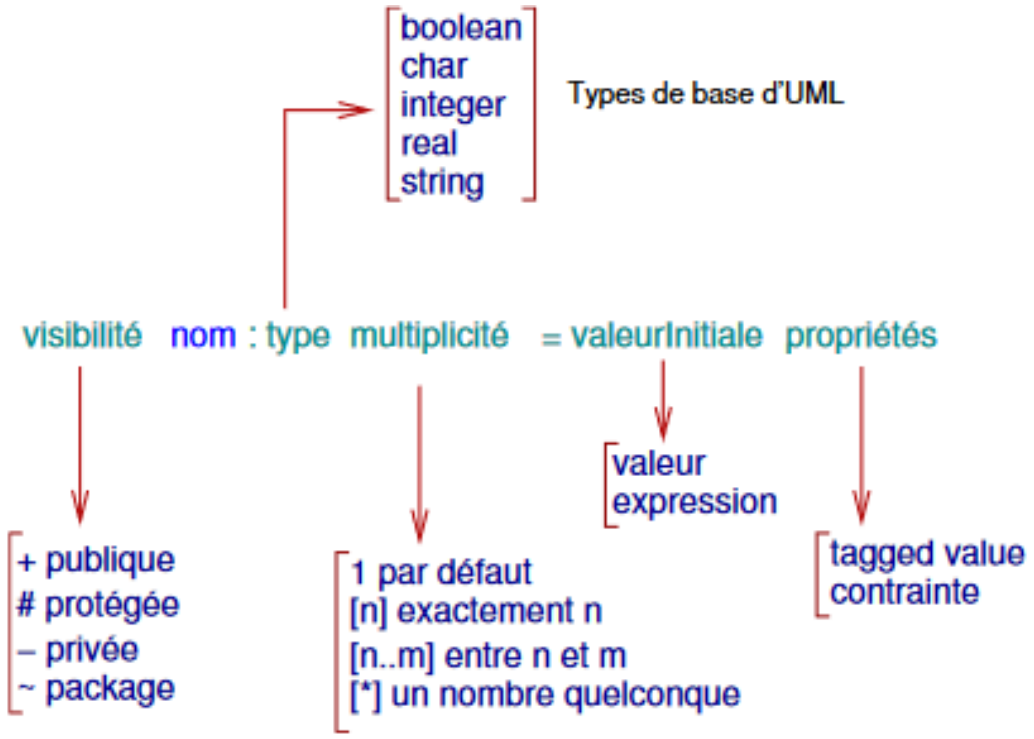


Diagramme de classes

Les méthodes

- Dans une classe, une opération (même nom et même types de paramètres) doit être unique. Quand le nom d'une opération apparaît plusieurs fois avec des paramètres différents, on dit que l'opération est surchargée. En revanche, il est impossible que deux opérations ne se distinguent que par leur valeur retournée.
- La direction peut être ajoutée. Elle peut prendre l'une des valeurs suivantes :
  - in : Paramètre d'entrée passé par valeur. Les modifications du paramètre ne sont pas disponibles pour l'appelant. C'est le comportement par défaut.
  - out : Paramètre de sortie uniquement. Il n'y a pas de valeur d'entrée et la valeur finale est disponible pour l'appelant.
  - inout : Paramètre d'entrée/sortie. La valeur finale est disponible pour l'appelant.
- Comme pour les attributs de classe, il est possible de déclarer des méthodes de classe. Une méthode de classe ne peut manipuler que des attributs de classe et ses propres paramètres. Cette méthode n'a pas accès aux attributs de la classe. L'accès à une méthode de classe ne nécessite pas l'existence d'une instance de cette classe. Graphiquement, une méthode de classe est soulignée.
- Une méthode est dite abstraite lorsqu'on connaît son entête mais pas la manière dont elle peut être réalisée (i.e. on connaît sa déclaration mais pas sa définition). Une classe est dite abstraite lorsqu'elle définit au moins une méthode abstraite ou lorsqu'une classe parent contient une méthode abstraite non encore réalisée. On ne peut instancier une classe abstraite : elle est vouée à se spécialiser. Une classe abstraite peut très bien contenir des méthodes concrètes. Une classe abstraite pure ne comporte que des méthodes abstraites. En programmation orientée objet, une telle classe est appelée une interface.

Diagramme de classes

Généralisation et héritage

- La généralisation décrit une relation entre une classe générale (classe de base ou classe parent) et une classe spécialisée (sous-classe). La classe spécialisée est intégralement cohérente avec la classe de base, mais comporte des informations supplémentaires (attributs, opérations, associations). Un objet de la classe spécialisée peut être utilisé partout où un objet de la classe de base est autorisé. En POO on parle d'héritage.
- Le symbole utilisé pour la relation d'héritage ou de généralisation est une flèche avec un trait plein dont la pointe est un triangle fermé désignant le cas le plus général.
- Les propriétés principales de l'héritage sont :
  - La classe enfant possède toutes les propriétés des ses classes parents, mais elle ne peut accéder aux propriétés privées de celle-ci.
  - Une classe enfant peut redéfinir (même signature) une ou plusieurs méthodes de la classe parent. Sauf indication contraire, un objet utilise les opérations les plus spécialisées dans la hiérarchie des classes.
  - Toutes les associations de la classe parent s'appliquent aux classes dérivées.
  - Une instance d'une classe peut être utilisée partout où une instance de sa classe parent est attendue.
  - Une classe peut avoir plusieurs parents, on parle alors d'héritage multiple.

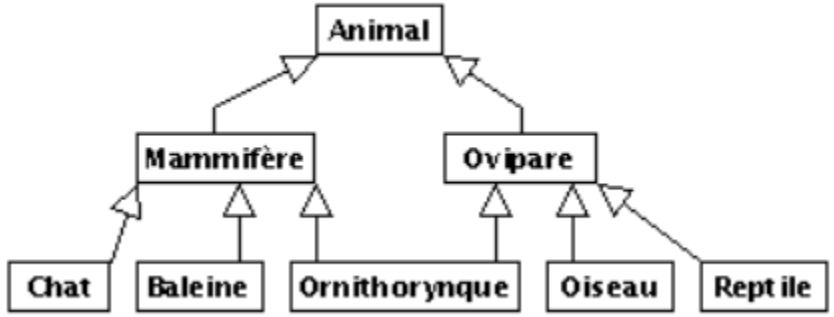
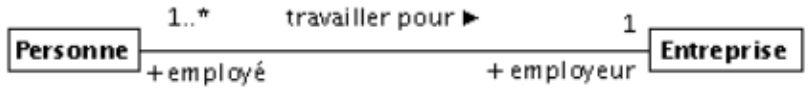
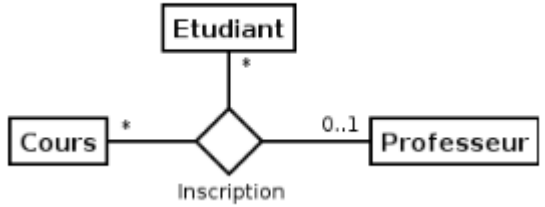


Diagramme de classes

Association

- Une association est une relation entre deux classes ou plus qui décrit les connexions structurelle entre leurs instances. Une association est matérialisée par un trait plein entre les classes associées.
- Une association n-aire lie plus de deux classes. On représente une association n-aire par un grand losange avec un chemin partant vers chaque classe participante. Le nom de l'association, le cas échéant, apparaît à proximité du losange. La ligne pointillé d'une classe-association peut être reliée au losange par une ligne discontinue pour représenter une association n-aire dotée d'attributs, d'opérations ou d'associations.
- La navigabilité indique s'il est possible de traverser une association. On représente graphiquement la navigabilité par une flèche du côté de la terminaison navigable et on empêche la navigabilité par une croix du côté de la terminaison non navigable. Par défaut, une association est navigable dans les deux sens.
- Remarque : Un attribut est une association dégénérée dans laquelle une terminaison d'association est détenue par une classe





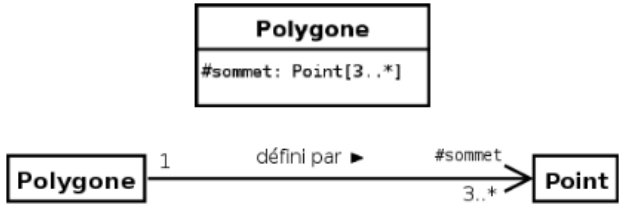
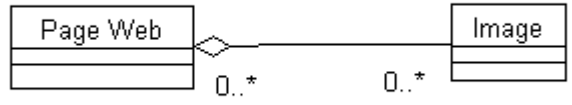


Diagramme de classes

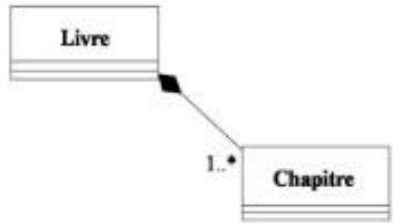
Agrégation

- L'agrégation est une association avec relation de subordination, représentée par un trait reliant les deux classes et dont l'origine se distingue de l'autre extrémité (la classe subordonnée) par un losange vide. Une des classes regroupe d'autres classes.



Composition

- La composition est une agrégation avec cycle de vie dépendant : la classe composante est détruite lorsque la classe composée (ou classe composite) disparaît. L'origine de cette association est représentée par un losange plein.



Dépendance

- La dépendance implique qu'une ou plusieurs méthodes reçoivent un objet d'un type d'une autre classe. Il n'y a pas de liaison en ce qui concerne la destruction d'objets mais une dépendance est quand même là. Elle est symbolisée par une flèche en pointillés, dont son extrémité possède trois traits qui se coupent en un même point.

Diagramme de classes

Multiplicité

- La multiplicité associée à une terminaison d'association, d'agrégation ou de composition déclare le nombre d'objets susceptibles d'occuper la position définie par la terminaison d'association. Voici quelques exemples de multiplicité :
  - exactement un : 1 ou 1..1
  - plusieurs : $*$ ou 0..$*$
  - au moins un : 1..$*$
  - de un à six : 1..6
- Dans une association binaire, la multiplicité sur la terminaison cible contraint le nombre d'objets de la classe cible pouvant être associés à un seul objet donné de la classe source (la classe de l'autre terminaison de l'association). Dans une association n-aire, la multiplicité apparaissant sur le lien de chaque classe s'applique sur une instance de chacune des classes, à l'exclusion de la classe-association et de la classe considérée. Par exemple, si on prend une association ternaire entre les classes (A, B, C), la multiplicité de la terminaison C indique le nombre d'objets C qui peuvent apparaître dans l'association avec une paire particulière d'objets A et B.

Diagramme de classes

Interfaces

- Les classes permettent de définir en même temps un objet et son interface. Ici, nous ne définissons que des éléments d'interface. Il peut s'agir de l'interface complète d'un objet, ou simplement d'une partie d'interface qui sera commune à plusieurs objets. Le rôle de ce classeur, stéréotypé « interface », est de regrouper un ensemble de propriétés et d'opérations assurant un service cohérent.
- Une interface est représentée comme une classe excepté l'absence du mot-clef abstract (car l'interface et toutes ses méthodes sont, par définition, abstraites) et l'ajout du stéréotype « interface ». Une interface doit être réalisée par au moins une classe. Graphiquement, cela est représenté par un trait discontinu terminé par une flèche triangulaire et le stéréotype « realize ». Une classe (classe cliente de l'interface) peut dépendre d'une interface (interface requise). On représente cela par une relation de dépendance et le stéréotype « use ».

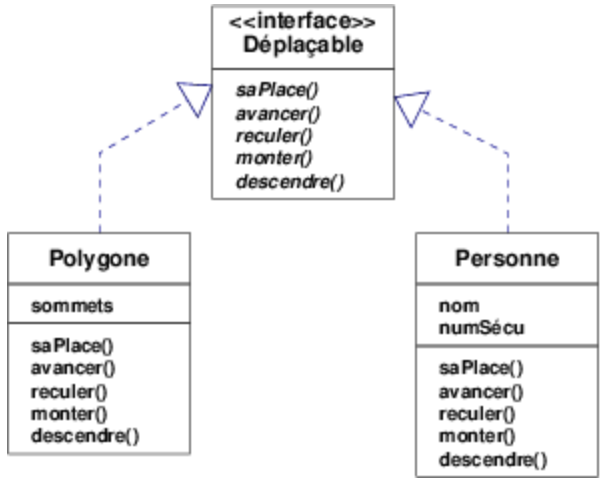


Diagramme de classes

Construire un diagramme de classes

- Il y a au moins trois points de vue qui guident la modélisation. En fonction du point de vue adopté, vous obtiendrez des modèles différents.
  - Le point de vue spécification met l'accent sur les interfaces des classes plutôt que sur leurs contenus.
  - Le point de vue conceptuel capture les concepts du domaine et les liens qui les lient. Il s'intéresse peu ou prou à la manière éventuelle d'implémenter ces concepts et relations et aux langages d'implémentation.
  - Le point de vue implémentation, le plus courant, détaille le contenu et l'implémentation de chaque classe.
- Une démarche couramment utilisée pour bâtir un diagramme de classes consiste à :
  - Trouver les classes du domaine étudié. Cette étape empirique se fait généralement en collaboration avec un expert du domaine. Les classes correspondent généralement à des concepts ou des substantifs du domaine.
  - Trouver les associations entre classes. Les associations correspondent souvent à des verbes, ou des constructions verbales, mettant en relation plusieurs classes, comme « est composé de », « pilote », « travaille pour ». Attention, méfiez vous de certains attributs qui sont en réalité des relations entre classes.
  - Trouver les attributs des classes. Les attributs correspondent souvent à des substantifs, ou des groupes nominaux, tels que « la masse d'une voiture » ou « le montant d'une transaction ». Les adjectifs et les valeurs correspondent souvent à des valeurs d'attributs. Vous pouvez ajouter des attributs à toutes les étapes du cycle de vie d'un projet (implémentation comprise). N'espérez pas trouver tous les attributs dès la construction du diagramme de classes.
  - Organiser et simplifier le modèle en éliminant les classes redondantes et en utilisant l'héritage.
  - Vérifier les chemins d'accès aux classes.
  - Itérer et raffiner le modèle. Un modèle est rarement correct dès sa première construction. La modélisation objet est un processus non pas linéaire mais itératif.
- On privilégie la notation des noms en CamelCase (pour du Python) :
  - les classes commencent par une Majuscule,
  - les objets commencent par une minuscule,
  - les mots sont accolés et écrits en minuscule, sauf la première lettre de chaque mot qui est majuscule.

Exercice

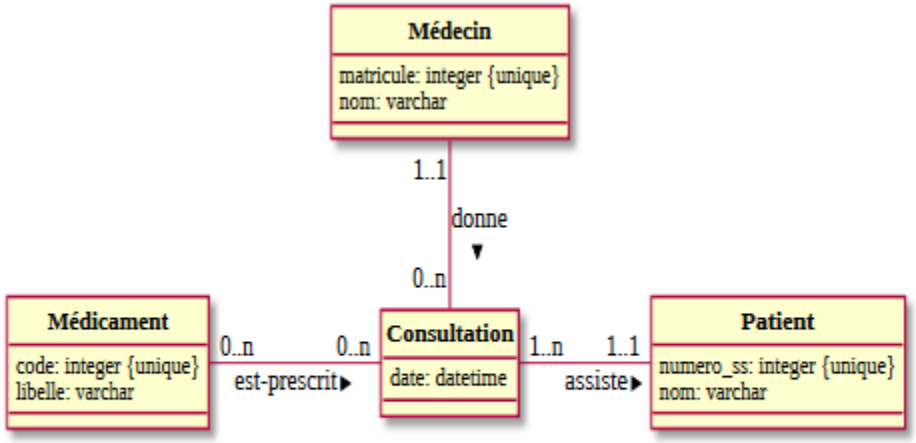


Image 1

A Un patient peut effectuer plusieurs visites.

B Tous les patients ont effectué au moins une consultation.

C Un médecin peut recevoir plusieurs patients pendant la même consultation.

D Un médecin peut prescrire plusieurs médicaments lors d'une même consultation.

E Deux médecins différents peuvent prescrire le même médicament.

Exercice

- Une personne est caractérisée par son nom, son prénom, son sexe et son âge. Les objets de classe Personne doivent pouvoir calculer leurs revenus et leurs charges. Les attributs de la classe sont privés ; le nom, le prénom ainsi que l'âge de la personne doivent être accessibles par des opérations publiques.
- Question : Donnez une représentation UML de la classe Personne, en remplissant tous les compartiments adéquats.

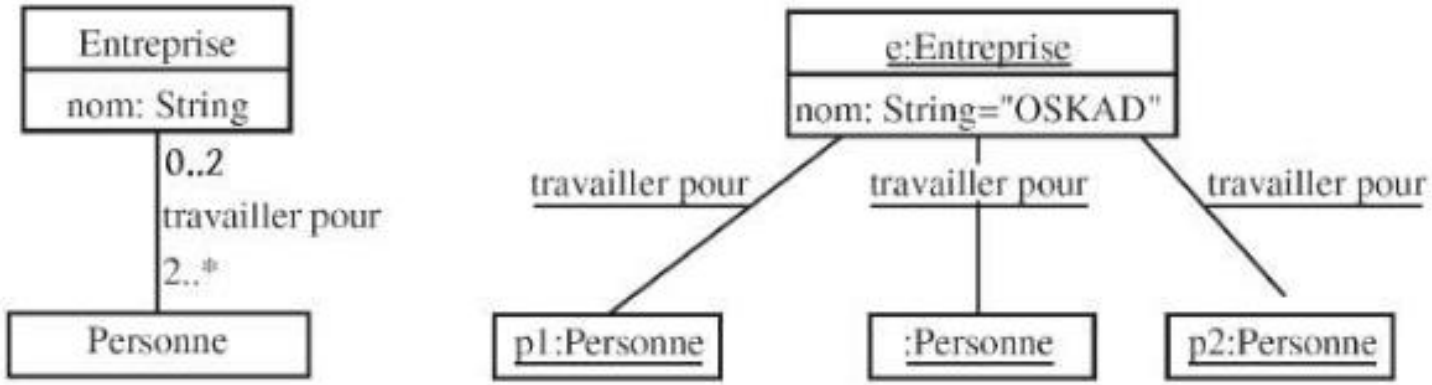
- Deux types de revenus sont envisagés : d'une part le salaire et d'autre part toutes les autres sources de revenus. Les deux revenus sont représentés par des nombres réels. Pour calculer les charges globales, on applique un coefficient fixe de 20% sur les salaires et un coefficient de 15% sur les autres revenus.
- Question : Enrichissez la représentation précédente pour prendre en compte ces nouveaux éléments.

- Un objet de la classe Personne peut être créé à partir du nom et de la date de naissance. Il est possible de changer le prénom d'une personne. Par ailleurs, le calcul des charges ne se fait pas de la même manière lorsque la personne décède.
- Question : Enrichissez encore la représentation précédente pour prendre en compte ces nouveaux éléments.

Diagramme d’objets

- Un diagramme d’objets représente des objets (i.e. instances de classes) et leurs liens (i.e. instances de relations) pour donner une vue de l’état du système à un instant donné. Un diagramme d’objets permet, selon les situations, d’illustrer le modèle de classes (en montrant un exemple qui explique le modèle), de préciser certains aspects du système (en mettant en évidence des détails imperceptibles dans le diagramme de classes), d’exprimer une exception (en modélisant des cas particuliers, des connaissances non généralisables . . .), ou de prendre une image (snapshot) d’un système à un moment donné. Le diagramme de classes modélise les règles et le diagramme d’objets modélise des faits. Un diagramme d’objets ne montre pas l’évolution du système dans le temps. Pour représenter une interaction, il faut utiliser un diagramme de communication.
- Graphiquement, un objet se représente comme une classe. Cependant, le compartiment des opérations n’est pas utile. De plus, le nom de la classe dont l’objet est une instance est précédé d’un « : » et est souligné. Pour différencier les objets d’une même classe, leur identifiant peut être ajouté devant le nom de la classe. Enfin les attributs reçoivent des valeurs. Quand certaines valeurs d’attribut d’un objet ne sont pas renseignées, on dit que l’objet est partiellement défini.
- Dans un diagrammes d’objets, les relations du diagramme de classes deviennent des liens. Graphiquement, un lien se représente comme une relation, mais, s’il y a un nom, il est souligné. Naturellement, on ne représente pas les multiplicités.

Diagramme d’objets



Exercice

- Pour chacun des énoncés suivants, donnez un diagramme des classes :
  - Tout écrivain a écrit au moins une œuvre
  - Les personnes peuvent être associées à des universités en tant qu'étudiants aussi bien qu'en tant que professeurs.
  - Un rectangle a deux sommets qui sont des points. On construit un rectangle à partir des coordonnées de deux points. Il est possible de calculer sa surface et son périmètre, ou encore de le translater.
  - Les cinémas sont composés de plusieurs salles. Les films sont projetés dans des salles. Les projections correspondantes ont lieu à chacune à une heure déterminée.
  - Tous les jours, le facteur distribue des recommandés dans une zone géographique qui lui est affectée. Les habitants sont aussi associés à une zone géographique. Les recommandés sont de deux sortes : lettres ou colis. Comme plusieurs facteurs peuvent intervenir sur la même zone, on souhaite, pour chaque recommandé, le facteur qui l'a distribué, en plus du destinataire.

Exercice

- Un hôtel est composé d'au moins deux chambres. Chaque chambre dispose d'une salle d'eau : douche ou bien baignoire. Un hôtel héberge des personnes. Il peut employer du personnel et il est impérativement dirigé par un directeur. On ne connaît que le nom et le prénom des employés, des directeurs et des occupants. Certaines personnes sont des enfants et d'autres des adultes (faire travailler des enfants est interdit). Un hôtel a les caractéristiques suivantes : une adresse, un nombre de pièces et une catégorie. Une chambre est caractérisée par le nombre et de lits qu'elle contient, son prix et son numéro. On veut pouvoir savoir qui occupe quelle chambre à quelle date. Pour chaque jour de l'année, on veut pouvoir calculer le loyer de chaque chambre en fonction de son prix et de son occupation (le loyer est nul si la chambre est inoccupée). La somme de ces loyers permet de calculer le chiffre d'affaires de l'hôtel entre deux dates.
- Question : Donnez une diagramme de classes pour modéliser le problème de l'hôtel.

Exercice

- Une banque compte plusieurs agences réparties sur le territoire français. Une banque est caractérisée par le nom de son directeur général, son capital global, son propre nom et de l'adresse de son siège social. Le directeur général est identifié par son nom, son prénom et son revenu. Une agence a un numéro d'agence et une adresse. Chaque agence emploie plusieurs employés, qui se caractérisent par leurs nom, prénom et date d'embauche. Les employés peuvent demander leur mutation d'une agence à une autre, mais un employé ne peut travailler que dans une seule agence. Les employés d'une agence ne font que gérer des clients. Un client ne peut avoir des comptes que dans une seule agence de la banque. Chaque nouveau client se voit systématiquement attribuer un employé de l'agence (conseiller). Les clients ont un nom, un prénom et une adresse. Les comptes sont de nature différente selon qu'ils soient rémunérés ou non (comptes courants). Les comptes rémunérés ont un taux d'intérêt et rapportent des intérêts versés annuellement. Une première lecture de l'énoncé permettrait de faire apparaître les classes suivantes, avec leurs propriétés :
- Analysez ces classes et utilisez la généralisation pour factoriser au mieux la description des propriétés.
- Une relation particulière lie l'agence, le client, l'employé et le compte. De quelle relation s'agit-il ? Donnez un diagramme de classes pour la modéliser
- Donnez le diagramme de classes en n'utilisant que le nom des classes et ajoutez toutes les décorations adéquates aux associations.

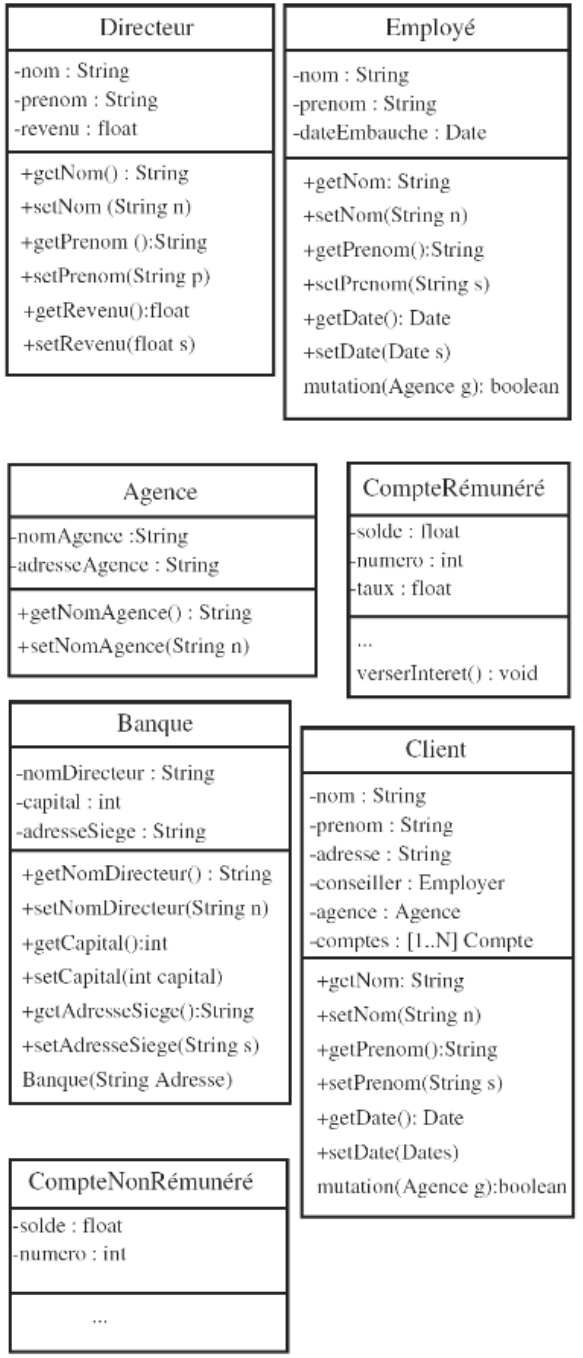


Diagramme d'états-transition

- Le diagramme d'états-transitions (*state machine diagram*) d'UML décrit le comportement interne d'un objet à l'aide d'un automate à états finis. Il présente les séquences possibles d'états et d'actions qu'une instance de classe peut traiter au cours de son cycle de vie en réaction à des événements discrets (de type signaux, invocations de méthode).
- Le diagramme d'états-transitions est le seul diagramme, de la norme UML, à offrir une vision complète et non ambiguë de l'ensemble des comportements de l'élément auquel il est attaché. En effet, un diagramme d'interaction n'offre qu'une vue partielle correspondant à un scénario sans spécifier comment les différents scénarii interagissent entre eux.
- La vision globale du système n'apparaît pas sur ce type de diagramme puisqu'ils ne s'intéressent qu'à un seul élément du système indépendamment de son environnement. Concrètement, un diagramme d'états-transitions est un graphe qui représente un automate à états finis, c'est-à-dire une machine dont le comportement des sorties ne dépend pas seulement de l'état de ses entrées, mais aussi d'un historique des sollicitations passées.
- Un diagramme d'états-transitions rassemble et organise les états et les transitions d'un classeur donné. Bien entendu, le modèle dynamique du système comprend plusieurs diagrammes d'états-transitions. Il est souhaitable de construire un diagramme d'états-transitions pour chaque classeur (qui, le plus souvent, est une classe) possédant un comportement dynamique important. Un diagramme d'états-transitions ne peut être associé qu'à un seul classeur. Tous les automates à états finis des diagrammes d'états-transitions d'un système s'exécutent concurremment et peuvent donc changer d'état de façon indépendante.

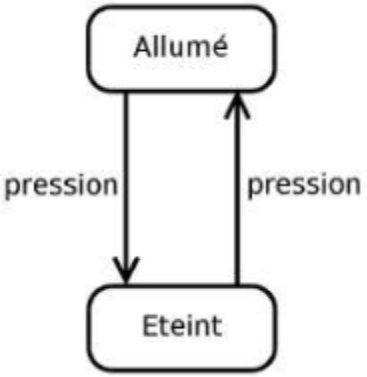


Diagramme d’états-transition

- Un état, que l’on peut qualifier informellement d’élémentaire, se représente graphiquement dans un diagrammes d’états-transitions par un rectangles aux coins arrondi.
- L’état initial est un pseudo état qui indique l’état de départ, par défaut, lorsque le diagramme d’états-transitions, ou l’état enveloppant, est invoqué. Lorsqu’un objet est créé, il entre dans l’état initial.
- L’état final est un pseudo état qui indique que le diagramme d’états-transitions, ou l’état enveloppant, est terminé

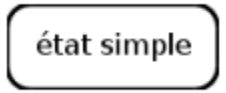


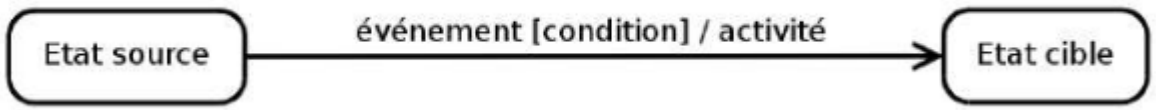
Diagramme d'états-transition

- Un événement est quelque chose qui se produit pendant l'exécution d'un système et qui mérite d'être modélisé. Les diagrammes d'états-transitions permettent justement de spécifier les réactions d'une partie du système à des événements discrets. Un événement se produit à un instant précis et est dépourvu de durée. Quand un événement est reçu, une transition peut être déclenchée et faire basculer l'objet dans un nouvel état. On peut diviser les événements en plusieurs types explicites et implicites :
  - Signal : Un signal est un type de classeur destiné explicitement à véhiculer une communication asynchrone à sens unique entre deux objets. L'objet expéditeur crée et initialise explicitement une instance de signal et l'envoi à un objet explicite ou à tout un groupe d'objets. Il n'attend pas que le destinataire traite le signal pour poursuivre son déroulement. La réception d'un signal est un événement pour l'objet destinataire. Un même objet peut être à la fois expéditeur et destinataire. Les signaux sont déclarés par le stéréotype « signal »
  - Appel : Un événement d'appel (call) représente la réception de l'appel d'une opération par un objet. Les paramètres de l'opération sont ceux de l'événement d'appel. La syntaxe d'un événement d'appel est la même que celle d'un signal. Par contre, les événements d'appel sont des méthodes déclarées au niveau du diagramme de classes.
  - Changement : Un événement de changement (change) est généré par la satisfaction d'une
  - expression booléenne sur des valeurs d'attributs. Il s'agit d'une manière déclarative d'attendre qu'une condition soit satisfaite.
  - Temporel : Les événements temporels sont générés par le passage du temps. Ils sont spécifiés soit de manière absolue (date précise), soit de manière relative (temps écoulé). Par défaut, le temps commence à s'écouler dès l'entrée dans l'état courant.

Diagramme d’états-transition

- Une transition définit la réponse d’un objet à l’occurrence d’un événement. Elle lie, généralement, deux états E1 et E2 et indique qu’un objet dans un état E1 peut entrer dans l’état E2 et exécuter certaines activités, si un événement déclencheur se produit et que la condition de garde est vérifiée. Le même événement peut être le déclencheur de plusieurs transitions quittant un même état. Chaque transition avec le même événement doit avoir une condition de garde différente. En effet, une seule transition peut se déclencher dans un même flot d’exécution. Si deux transitions sont activées en même temps par un même événement, une seule se déclenche et le choix n’est pas prévisible.
- Une transition peut avoir une condition de garde (spécifiée par ’[’ <garde> ’]’ dans la syntaxe). Il s’agit d’une expression logique sur les attributs de l’objet, associé au diagramme d’états-transitions, ainsi que sur les paramètres de l’événement déclencheur. La condition de garde est évaluée uniquement lorsque l’événement déclencheur se produit. Si l’expression est fausse à ce moment là, la transition ne se déclenche pas, si elle est vraie, la transition se déclenche et ses effets se produisent.
- Lorsqu’une transition se déclenche (on parle également de tir d’une transition), son effet (spécifié par ’/’ <activité> dans la syntaxe) s’exécute. Lorsque l’exécution de l’effet est terminée, l’état cible de la transition devient actif.
- La ransition est en général externe. Il s’agit du type de transition le plus répandu. Elle est représentée par une flèche allant de l’état source vers l’état cible.
- Une transition dépourvue d’événement déclencheur explicite se déclenche à la fin de l’activité contenue dans l’état source (y compris les état imbriqués). Elle peut contenir une condition de garde qui est évaluée au moment où l’activité contenue dans l’état s’achève, et non pas ensuite. C’est une transitiond’achèvement.
- Les règles de déclenchement d’une transition interne sont les mêmes que pour une transition externe excepté qu’une transition interne ne possède pas d’état cible et que l’état actif reste le même à la suite de son déclenchement. La syntaxe d’une transition interne reste la même que celle d’une transition classique. Par contre, les transitions internes ne sont pas représentées par des arcs mais sont spécifiées dans un compartiment de leur état associé

Diagramme d’états-transition



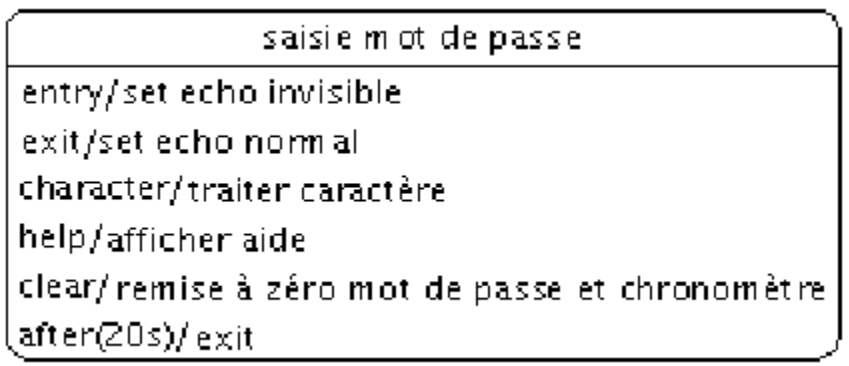
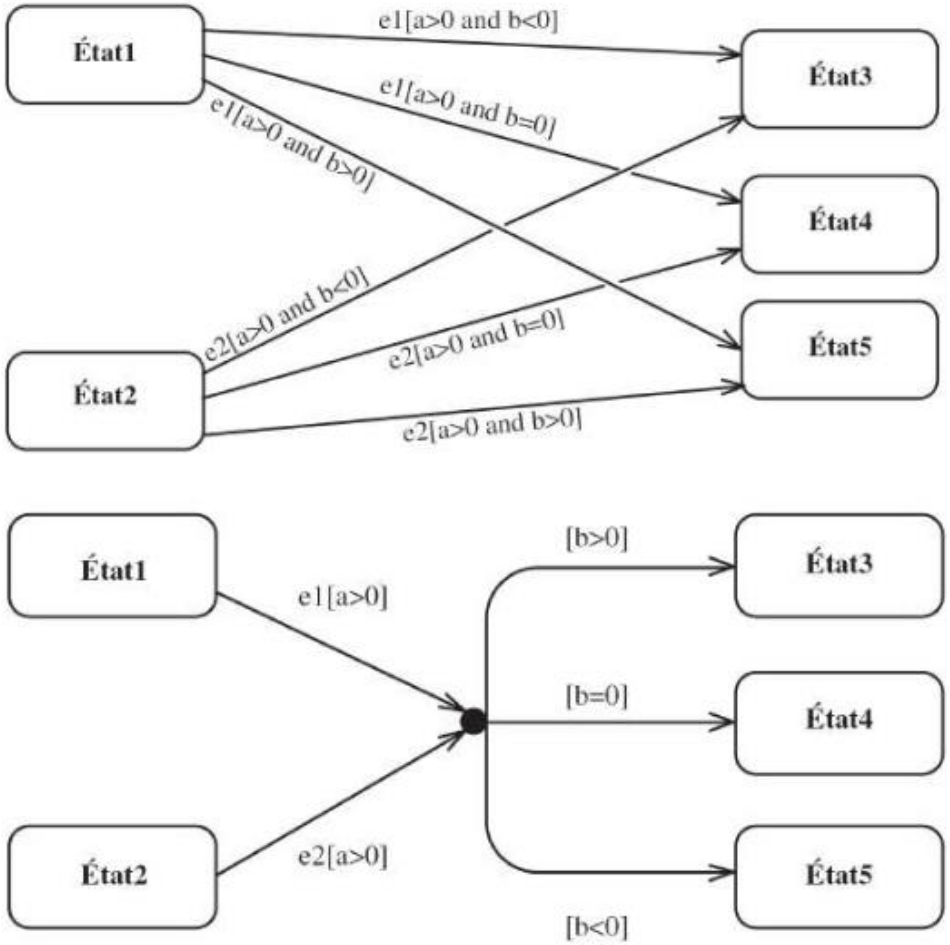


Diagramme d’états-transition

- Il est possible de représenter des alternatives pour le franchissement d’une transition. On utilise pour cela des pseudo-états particuliers : les points de jonction (représentés par un petit cercle plein) et les points de décision (représenté par un losange).
- Les points de jonction sont un artefact graphique (un pseudo-état en l’occurrence) qui permet de partager des segments de transition, l’objectif étant d’aboutir à une notation plus compacte ou plus lisible des chemins alternatifs. Un point de jonction peut avoir plusieurs segments de transition entrante et plusieurs segments de transition sortante. Par contre, il ne peut avoir d’activité interne ni des transitions sortantes dotées de déclencheurs d’événements. Il ne s’agit pas d’un état qui peut être actif au cours d’un laps de temps fini. Lorsqu’un chemin passant par un point de jonction est emprunté (donc lorsque la transition associée est déclenchée) toutes les gardes le long de ce chemin doivent s’évaluer à vrai dès le franchissement du premier segment.
- Un point de décision possède une entrée et au moins deux sorties. Contrairement à un point de jonction, les gardes situées après le point de décision sont évaluées au moment où il est atteint. Cela permet de baser le choix sur des résultats obtenus en franchissant le segment avant le point de choix. Si, quand le point de décision est atteint, aucun segment en aval n’est franchissable, c’est que le modèle est mal formé.

Diagramme d’états-transition



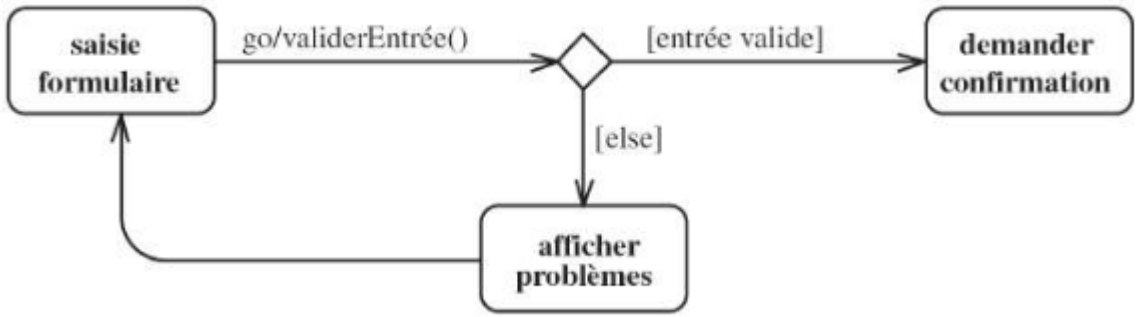


Diagramme d’états-transition

Etat composite

- Un état simple ne possède pas de sous-structure mais uniquement, le cas échéant, un jeu de transitions internes. Un état composite est un état décomposé en régions contenant chacune un ou plusieurs sous-états. Implicitement, tout diagramme d’états-transitions est contenu dans un état externe qui n’est usuellement pas représenté. Cela apporte une plus grande homogénéité dans la description : tout diagramme d’états-transitions est implicitement un état composite. L’utilisation d’états composites permet de développer une spécification par raffinements. Il n’est pas nécessaire de représenter les sous-états à chaque utilisation de l’état englobant.

Etat historique

- Un état historique, également qualifié d’état historique plat, est un pseudo-état qui mémorise le dernier sous-état actif d’un état composite. Graphiquement, il est représenté par un cercle contenant un H. Une transition ayant pour cible l’état historique est équivalente à une transition qui a pour cible le dernier état visité de l’état englobant. Un état historique peut avoir une transition sortante non étiquetée indiquant l’état à exécuter si la région n’a pas encore été visitée. Il est également possible de définir un état historique profond représenté graphiquement par un cercle contenant un H*. Cet état historique profond permet d’atteindre le dernier état visité dans la région, quel que soit sont niveau d’imbrication, alors que le l’état historique plat limite l’accès aux états de son niveau d’imbrication.

Diagramme d’états-transition

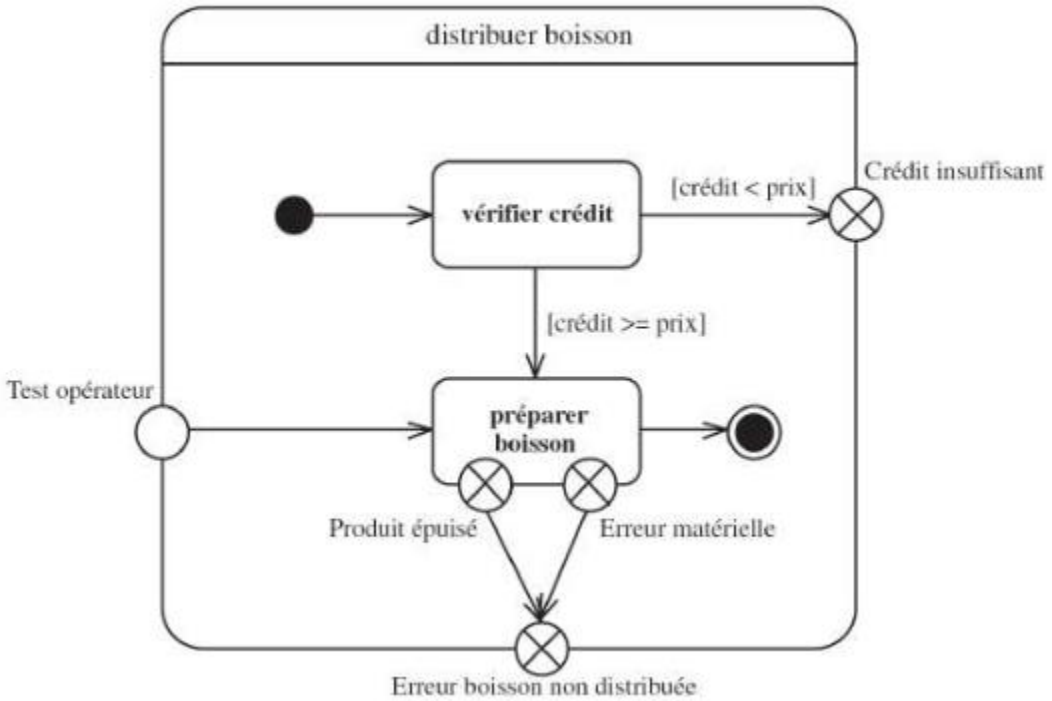
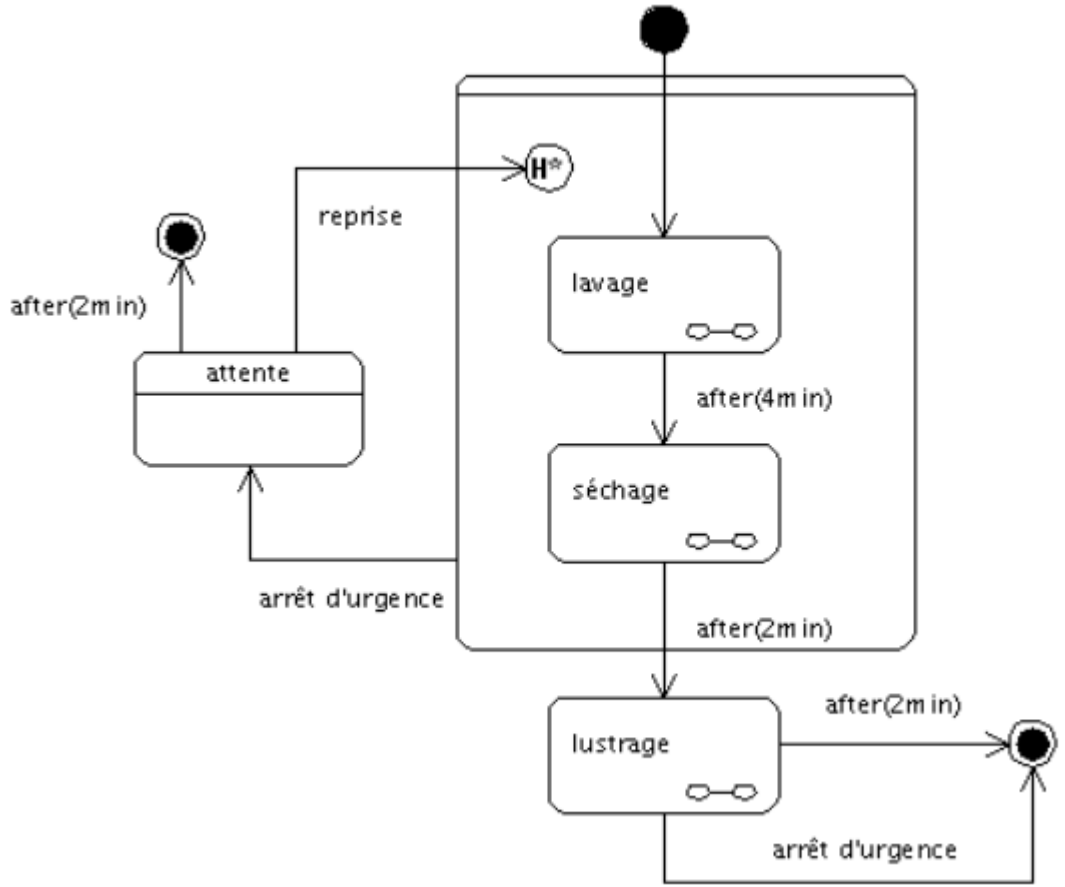


Diagramme d’états-transition



Exercice

- Considérons une classe Partie dont la responsabilité est de gérer le déroulement d'une partie de jeu d'échecs. Cette classe peut être dans deux états :
    - le tour des blancs
    - le tour des noirs.

Les événements à prendre en considération sont

- un déplacement de pièces de la part du joueur noir
- un déplacement de pièces de la part du joueur blanc
- la demande de prise en compte d'un échec et mat par un joueur. S'il est validé par la classe partie,un échec et mat assure la victoire du dernier joueur. Dans ce cas, une activité « noirsGagnants » ou « blancsGagnants » selon le cas est déclenchée (appel de méthode).
- la demande de prise en compte d'un pat qui mène aussi à une fin de partie, avec une égalité. Dans ce cas, une activité « égalité » est déclenchée.

- Donner le diagramme d'états/transitions associé à la classe Partie.

Diagramme d’activités

- Les diagrammes d’activités permettent de mettre l’accent sur les traitements. Ils sont donc particulièrement adaptés à la modélisation du cheminement de flots de contrôle et de flots de données. Ils permettent ainsi de représenter graphiquement le comportement d’une méthode ou le déroulement d’un cas d’utilisation.
- Les diagrammes d’activités sont relativement proches des diagrammes d’états-transitions dans leur présentation, mais leur interprétation est sensiblement différente. Les diagrammes d’états-transitions sont orientés vers des systèmes réactifs, mais ils ne donnent pas une vision satisfaisante d’un traitement faisant intervenir plusieurs classeurs et doivent être complétés, par exemple, par des diagrammes de séquence. Au contraire, les diagrammes d’activités ne sont pas spécifiquement rattachés à un classeur particulier. Ils permettent de spécifier des traitements a priori séquentiels et offrent une vision très proche de celle des langages de programmation impératifs.
- Une **action** est le plus petit traitement qui puisse être exprimé en UML. Une action a une incidence sur l’état du système ou en extrait une information. Les actions sont des étapes discrètes à partir desquelles se construisent les comportements. La notion d’action est à rapprocher de la notion d’instruction élémentaire d’un langage de programmation.
- Une **activité** définit un comportement décrit par un séquencement organisé d’unités dont les éléments simples sont les actions. Le flot d’exécution est modélisé par des nœuds reliés par des arcs (transitions). Le flot de contrôle reste dans l’activité jusqu’à ce que les traitements soient terminés. Une activité est un comportement (behavior en anglais) et à ce titre peut être associée à des paramètres

Diagramme d'activités

- Les actions les plus courantes de la norme UML sont les suivantes :
  - Action appeler (call operation) – L'action call operation correspond à l'invocation d'une opération sur un objet de manière synchrone ou asynchrone. Lorsque l'action est exécutée, les paramètres sont transmis à l'objet cible. Si l'appel est asynchrone, l'action est terminée et les éventuelles valeurs de retour seront ignorées. Si l'appel est synchrone, l'appelant est bloqué pendant l'exécution de l'opération et, le cas échéant, les valeurs de retour pourront être réceptionnées.
  - Action comportement (call behavior) – L'action call behavior est une variante de l'action call operation car elle invoque directement une activité plutôt qu'une opération.
  - Action envoyer (send) – Cette action crée un message et le transmet à un objet cible, où elle peut déclencher un comportement. Il s'agit d'un appel asynchrone (i.e. qui ne bloque pas l'objet appelant) bien adapté à l'envoi de signaux (send signal).
  - Action accepter événement (accept event) – L'exécution de cette action bloque l'exécution en cours jusqu'à la réception du type d'événement spécifié, qui généralement est un signal. Cette action est utilisée pour la réception de signaux asynchrones.
  - Action accepter appel (accept call) – Il s'agit d'une variante de l'action accept event pour les appels synchrones.
  - Action répondre (reply) – Cette action permet de transmettre un message en réponse à la réception d'une action de type accept call.
  - Action créer (create) – Cette action permet d'instancier un objet.
  - Action détruire (destroy) – Cette action permet de détruire un objet.
  - Action lever exception (raise exception) – Cette action permet de lever explicitement une exception

Diagramme d’activités

- Un nœud d’activité est un type d’élément abstrait permettant de représenter les étapes le long du flot d’une activité.
  - Nœud d’action
  - Nœud objet
  - Nœud de décision
  - Nœud de bifurcation ou d’union
  - Nœud initial (il peut y en avoir plusieurs)
  - Nœud final
  - Nœud final de flot

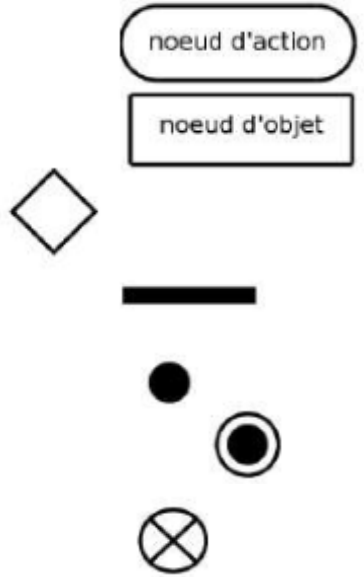
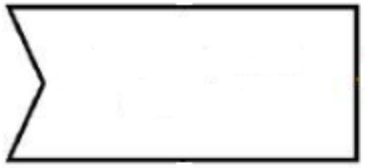
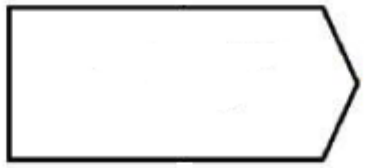


Diagramme d'activités

- Un nœud d'action est l'unité fondamentale de fonctionnalité exécutable dans une activité. L'exécution d'une action représente une transformation ou un calcul quelconque dans le système modélisé. Les actions sont généralement liées à des opérations qui sont directement invoquées. Un nœud d'action doit avoir au moins un arc entrant. Graphiquement, un nœud d'action est représenté par un rectangle aux coins arrondis qui contient sa description textuelle. Cette description textuelle peut aller d'un simple nom à une suite d'actions réalisées par l'activité. UML n'impose aucune syntaxe pour cette description textuelle, on peut donc utiliser une syntaxe proche de celle d'un langage de programmation particulier ou du pseudo-code.
- Certaines actions de communication ont une notation spéciale.



Accept event



Send signal

Accept
time event

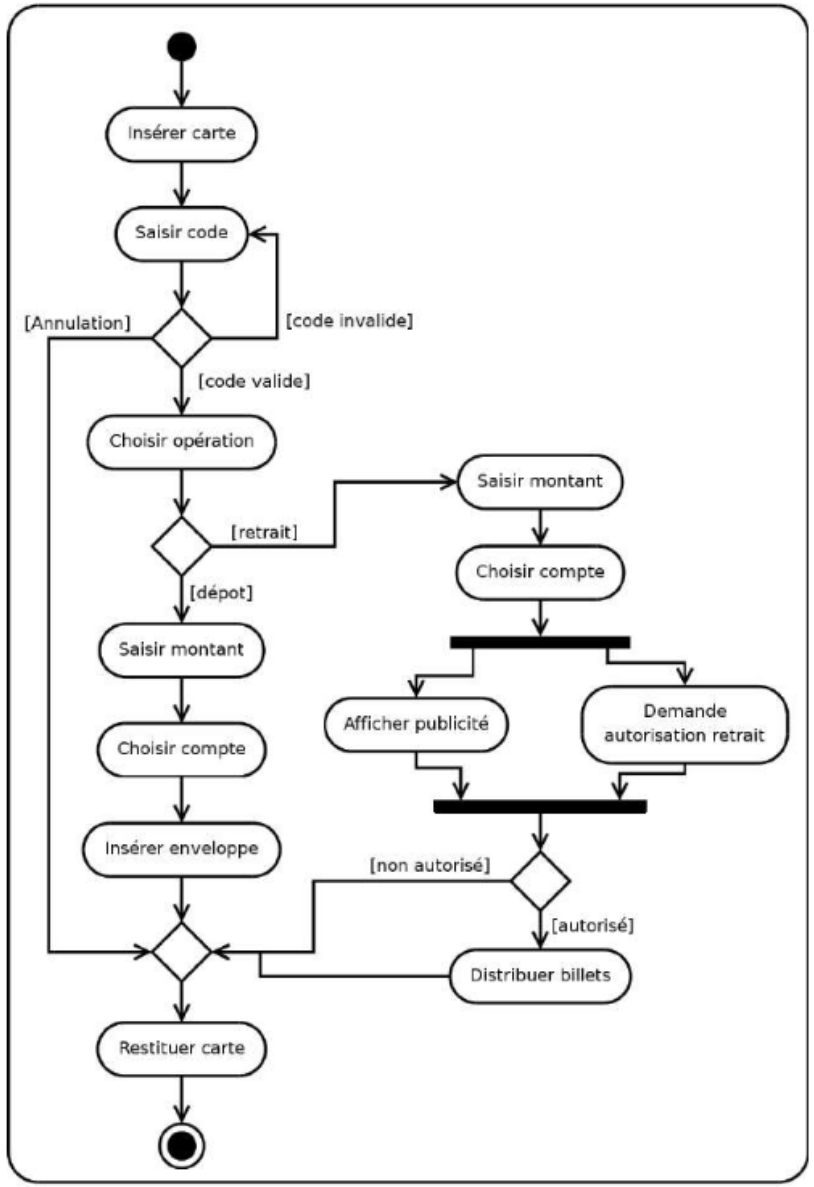


Diagramme d’activités

- Le passage d’une activité vers une autre est matérialisé par une transition. Graphiquement les transitions sont représentées par des flèches en traits pleins qui connectent les activités entre elles. Elles sont déclenchées dès que l’activité source est terminée et provoquent automatiquement et immédiatement le début de la prochaine activité à déclencher (l’activité cible). Contrairement aux activités, les transitions sont franchies de manière atomique, en principe sans durée perceptible. Les transitions spécifient l’enchaînement des traitements et définissent le flot de contrôle.

Diagramme d’activités

Nœuds de contrôle

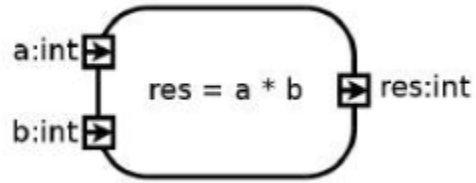
- Nœud initial : C’est le nœud à partir duquel le flot débute lorsque l’activité enveloppante est invoquée. Une activité peut avoir plusieurs nœuds initiaux. Un nœud initial possède un arc sortant et pas d’arc entrant.
- Nœud de fin d’activité : lorsque l’un des arcs d’un nœud de fin d’activité est activé (i.e. lorsqu’un flot d’exécution atteint un nœud de fin d’activité), l’exécution de l’activité enveloppante s’achève et tout nœud ou flot actif au sein de l’activité enveloppante est abandonné. Si l’activité a été invoquée par un appel synchrone, un message (reply) contenant les valeurs sortantes est transmis en retour à l’appelant. Graphiquement, un nœud de fin d’activité est représenté par un cercle vide contenant un petit cercle plein.
- Nœud de fin de flot : Lorsqu’un flot d’exécution atteint un nœud de fin de flot, le flot en question est terminé, mais cette fin de flot n’a aucune incidence sur les autres flots actifs de l’activité enveloppante. Graphiquement, un nœud de fin de flot est représenté par un cercle vide barré d’un X. Les nœuds de fin de flot sont particuliers et à utiliser avec parcimonie.

Diagramme d’activités

Nœuds de contrôle

- Nœud de décision : Un nœud de décision est un nœud de contrôle qui permet de faire un choix entre plusieurs flots sortants. Il possède un arc entrant et plusieurs arcs sortants. Ces derniers sont généralement accompagnés de conditions de garde pour conditionner le choix. Si, quand le nœud de décision est atteint, aucun arc en aval n’est franchissable (i.e. aucune condition de garde n’est vraie), c’est que le modèle est mal formé. L’utilisation d’une garde [else] est recommandée après un nœud de décision car elle garantit un modèle bien formé. En effet, la condition de garde [else] est validée si et seulement si toutes les autres gardes des transitions ayant la même source sont fausses. Dans le cas où plusieurs arcs sont franchissables (i.e. plusieurs conditions de garde sont vraies), seul l’un d’entre eux est retenu et ce choix est non déterministe. Graphiquement, on représente un nœud de décision par un losange.
- Nœud de fusion : Un nœud de fusion est un nœud de contrôle qui rassemble plusieurs flots alternatifs entrants en un seul flot sortant. Il n’est pas utilisé pour synchroniser des flots concurrents (c’est le rôle du nœud d’union) mais pour accepter un flot parmi plusieurs. Graphiquement, on représente un nœud de fusion, comme un nœud de décision, par un losange. Graphiquement, il est possible de fusionner un nœud de fusion et un nœud de décision, et donc d’avoir un losange possédant plusieurs arcs entrants et sortants. Il est également possible de fusionner un nœud de décision ou de fusion avec un autre nœud, comme un nœud de fin de flot ou avec une activité.
- Nœud de bifurcation : Un nœud de bifurcation, également appelé nœud de débranchement est un nœud de contrôle qui sépare un flot en plusieurs flots concurrents. Un tel nœud possède donc un arc entrant et plusieurs arcs sortants. On apparie généralement un nœud de bifurcation avec un nœud d’union pour équilibrer la concurrence. Graphiquement, on représente un nœud de bifurcation par un trait plein.
- Nœud d’union : Un nœud d’union, également appelé nœud de jointure est un nœud de contrôle qui synchronise des flots multiples. Un tel nœud possède donc plusieurs arcs entrants et un seul arc sortant. Lorsque tous les arcs entrants sont activés, l’arc sortant l’est également. Graphiquement, on représente un nœud de union, comme un nœud de bifurcation, par un trait plein

Diagramme d'activités



Nœuds de contrôle

- Nœud d'objet : Un nœud d'objet permet de définir un flot d'objet (i.e. un flot de données) dans un diagramme d'activités. Ce nœud représente l'existence d'un objet généré par une action dans une activité et utilisé par d'autres actions.
- Pour spécifier les valeurs passées en argument à une activité et les valeurs de retour, on utilise des nœuds d'objets appelés pin (pin en anglais) d'entrée ou de sortie. L'activité ne peut débuter que si l'on affecte une valeur à chacun de ses pins d'entrée. Quand l'activité se termine, une valeur doit être affectée à chacun de ses pins de sortie. Les valeurs sont passées par copie : une modification des valeurs d'entrée au cours du traitement de l'action n'est visible qu'à l'intérieur de l'activité. Graphiquement, un pin est représenté par un petit carré attaché à la bordure d'une activité . Il est typé et éventuellement nommé. Il peut contenir des flèches indiquant sa direction (entrée ou sortie) si l'activité ne permet pas de le déterminer de manière univoque.
- Un flot d'objets permet de passer des données d'une activité à une autre. Un arc reliant un pin de sortie à un pin d'entrée est, par définition même des pins, un flot d'objets. Dans cette configuration, le type du pin récepteur doit être identique ou parent (au sens de la relation de généralisation) du type du pin émetteur. Il existe une autre représentation possible d'un flot d'objets, plus axée sur les données proprement dites car elle fait intervenir un nœud d'objet détaché d'une activité particulière. Graphiquement, un tel nœud d'objet est représenté par un rectangle dans lequel est mentionné le type de l'objet (souligné). Des arcs viennent ensuite relier ce nœud d'objet à des activités sources et cibles. Le nom d'un état, ou d'une liste d'états, de l'objet peut être précisé entre crochets après ou sous le type de l'objet. On peut également préciser des contraintes entre accolades, soit à l'intérieur, soit en dessous du rectangle du nœud d'objet. Un flot d'objets peut porter une étiquette stéréotypée mentionnant deux comportements particuliers :
  - «transformation» indique une interprétation particulière de la donnée véhiculée par le flot ;
  - «selection» indique l'ordre dans lequel les objets sont choisis dans le nœud pour le quitter

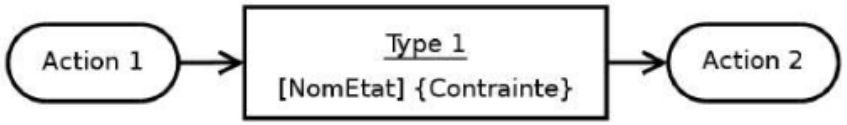
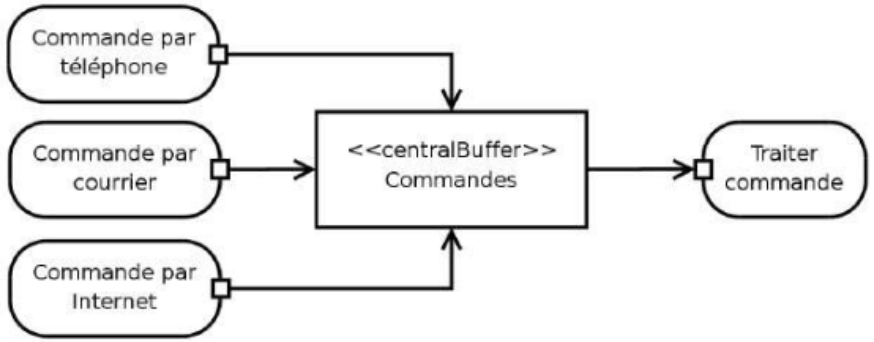


Diagramme d'activités

Nœuds de contrôle

- Nœud tampon central : Un nœud tampon central est un nœud d'objet qui accepte les entrées de plusieurs nœuds d'objets ou produit des sorties vers plusieurs nœuds d'objets. Les flots en provenance d'un nœud tampon central ne sont donc pas directement connectés à des actions. Ce nœud modélise donc un tampon traditionnel qui peut contenir des valeurs en provenance de diverses sources et livrer des valeurs vers différentes destinations. Graphiquement, un nœud tampon central est représenté comme un nœud d'objet détaché stéréotypé «centralBuffer».
- Noeud de stockage de données : Un nœud de stockage des données est un nœud tampon central particulier qui assure la persistance des données. Lorsqu'une information est sélectionnée par un flux sortant, l'information est dupliquée et ne disparaît pas du nœud de stockage des données comme ce serait le cas dans un nœud tampon central. Lorsqu'un flux entrant véhicule une donnée déjà stockée par le nœud de stockage des données, cette dernière est écrasée par la nouvelle. Graphiquement, un nœud tampon central est représenté comme un nœud d'objet détaché stéréotypé «datastore».



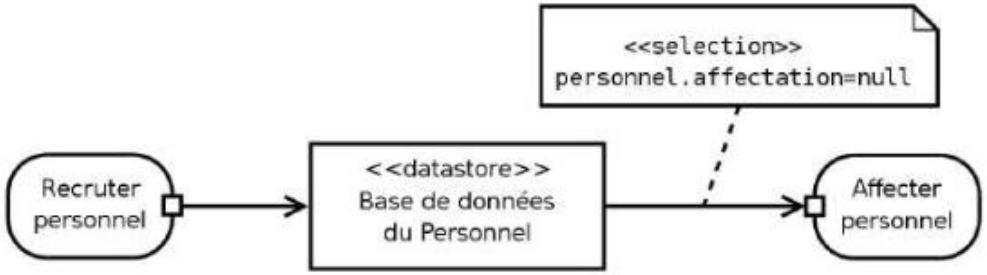


Diagramme d’activités

Partitions

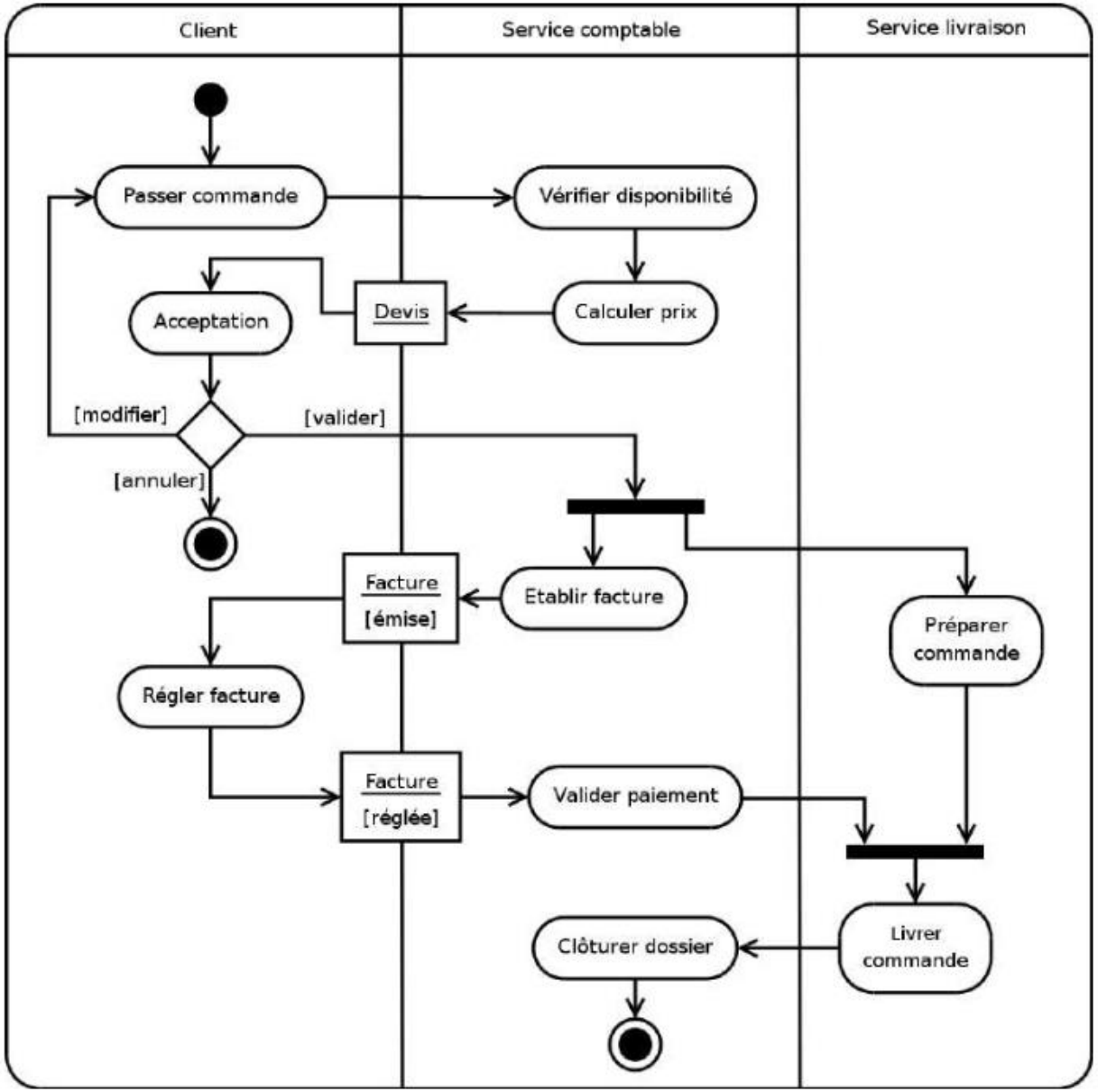


Diagramme d’activités

- Un nœud d’activité est un type d’élément abstrait permettant de représenter les étapes le long du flot d’une activité.
  - Nœud d’action
  - Nœud objet
  - Nœud de décision
  - Nœud de bifurcation ou d’union
  - Nœud initial (il peut y en avoir plusieurs)
  - Nœud final
  - Nœud final de flot

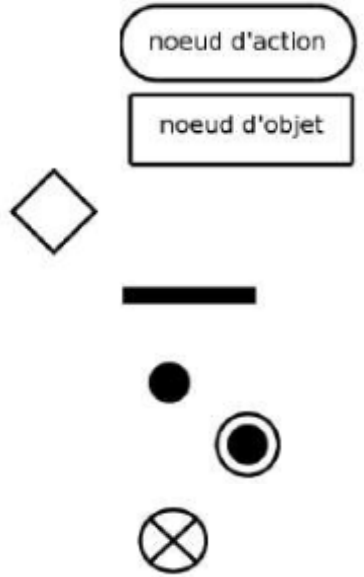
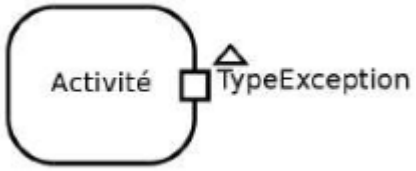
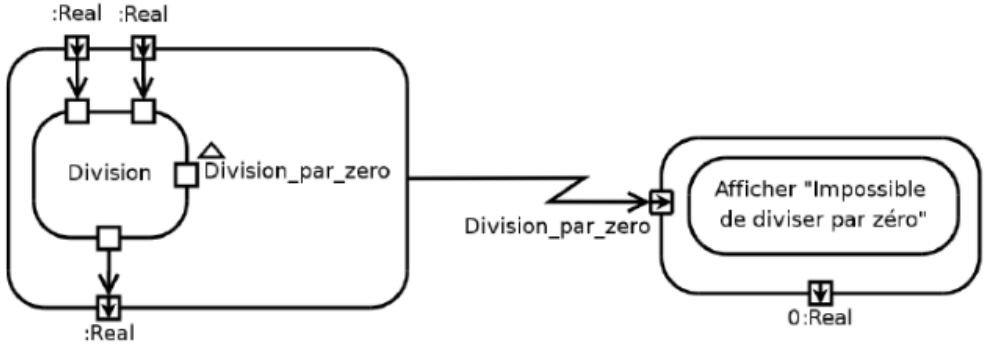


Diagramme d'activités

Exceptions

- Une exception est générée quand une situation anormale entrave le déroulement nominal d'une tâche. Elle peut être générée automatiquement pour signaler une erreur d'exécution (débordement d'indice de tableau, division par zéro, . . .), ou être soulevée explicitement par une action (RaiseException) pour signaler une situation problématique qui n'est pas prise en charge par la séquence de traitement normale. Graphiquement, on peut représenter le fait qu'une activité peut soulever une exception comme un pin de sortie orné d'un petit triangle et en précisant le type de l'exception à proximité du pin de sortie.
- Un gestionnaire d'exception est une activité possédant un pin d'entrée du type de l'exception qu'il gère et lié à l'activité qu'il protège par un arc en zigzag ou un arc classique orné d'une petite flèche en zigzag. Le gestionnaire d'exception doit avoir les mêmes pins de sortie que le bloc qu'il protège. Les exceptions sont des classeurs et, à ce titre, peuvent posséder des propriétés comme des attributs ou des opérations. Il est également possible d'utiliser la relation d'héritage sur les exceptions. Un gestionnaire d'exception spécifie toujours le type des exceptions qu'il peut traiter, toute exception dérivant de ce type est donc également prise en charge. Lorsqu'une exception survient, l'exécution de l'activité en cours est abandonnée sans générer de valeur de sortie. Le mécanisme d'exécution recherche alors un gestionnaire d'exception susceptible de traiter l'exception levée ou une de ses classes parentes. Si l'activité qui a levé l'exception n'est pas protégée de cette exception, l'exception est propagée à l'activité englobante. L'exécution de cette dernière est abandonnée, ses valeurs de sortie ne sont pas générées et un gestionnaire d'exception est recherché à son niveau. Dans la plupart des langages orientés objet, une exception qui se propage jusqu'à la racine du programme implique son arrêt. Quand un gestionnaire d'exception adapté a été trouvé et que son exécution se termine, l'exécution se poursuit comme si l'activité protégée s'était terminée normalement, les valeurs de sortie fournies par le gestionnaire remplaçant celle que l'activité protégée aurait dû produire.





Exercice

- Les chaînes de caractères du langage C sont codées comme un tableau de caractères non nuls, terminé par un caractère spécial. Par exemple, la chaîne s=‘hello !’ est codée comme suit :

s[0]	s[1]	s[2]	s[3]	s[4]	s[5]	s[6]
`h'	'e'	'l'	'l'	'o'	'!'	'\0'

- Question : Décrivez une activité implémentant la fonction strlen, qui prend en entrée un tableau de caractères et rend un entier correspondant à la taille de la chaîne.

Une implémentation possible pourraît être :

```
int strlen( char s[] )
{
int i ;
i = 0 ;
while( s[i] != '\0' )
++i ;
return( i ) ;
}
```

Exercice

- Au jeu d'échecs, la promotion a lieu lorsqu'un pion atteint le bout de l'échiquier. Dans le système étudié, la promotion est assurée par les trois classes : Joueur, Partie et Pion. Les cas qui mènent à d'autres déplacements sont ignorés (les représenter par un état final). Le joueur commence par saisir les cases du déplacement et finalement, s'il y a lieu, choisit la pièce qui vient remplacer le pion arrivé à destination. Le pion vérifie que le déplacement demandé est conforme à ses possibilités et la partie contrôle le déroulement global de la partie. C'est elle qui a une connaissance complète de l'échiquier. Ainsi, il est de sa responsabilité de vérifier que la case d'arrivée demandée est une case valide et si elle donne lieu à une promotion. Au moment où le pion vérifie la validité du coup, la partie effectue des vérifications concernant les règles générales (possibilité de jouer si le joueur est en échec...).

- Question : Représenter par un diagramme d'activité avec couloirs la promotion d'un pion en une pièce au choix dans le jeu d'échecs.

Diagramme d’interactions

- Un objet interagit pour implémenter un comportement. On peut décrire cette interaction de deux manières complémentaires : l’une est centrée sur des objets individuels (diagramme d’états-transitions) et l’autre sur une collection d’objets qui coopèrent (diagrammes d’interaction). La spécification d’un diagramme d’états-transitions est précise et conduit immédiatement au code. Elle ne permet pas pour autant d’expliquer le fonctionnement global d’un système, car elle se concentre sur un seul objet à la fois. La vue de l’ensemble des interactions offre une vue plus holistique du comportement d’un jeu d’objets.
- Les diagrammes d’interaction permettent d’établir un lien entre les diagrammes de cas d’utilisation et les diagrammes de classes : ils montrent comment des objets (i.e. des instances de classes) communiquent pour réaliser une certaine fonctionnalité. Ils apportent un aspect dynamique à la modélisation du système. Le modélisateur doit pouvoir focaliser son attention sur un sous-ensemble d’éléments du système et étudier leur façon d’interagir pour décrire un comportement particulier du système. UML permet de décrire un comportement limité à un contexte précis de deux façons : dans le cadre d’un classeur structuré ou dans celui d’une collaboration.

Diagramme d’interactions

Classeur structuré

- Les classes découvertes au moment de l’analyse (dans le diagramme de classes) ne sont pas assez détaillées pour pouvoir être implémentées par des développeurs. UML propose de partir des classeurs découverts au moment de l’analyse (classes, sous-systèmes, cas d’utilisation, . . .) et de les décomposer en éléments suffisamment fins pour permettre leur implémentation.
- Les classeurs ainsi décomposés s’appellent des classeurs structurés. Graphiquement, un classeur structuré se représente par un rectangle en trait plein comprenant deux compartiments. Le compartiment supérieur contient le nom du classeur et le compartiment inférieur montre les objets internes reliées par des connecteurs.

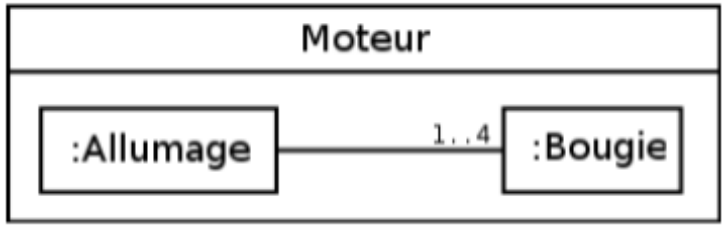
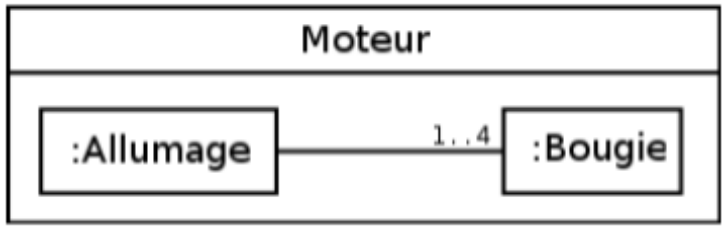


Diagramme d'interactions

Classeur structuré

- Les classes découvertes au moment de l'analyse (dans le diagramme de classes) ne sont pas assez détaillées pour pouvoir être implémentées par des développeurs. UML propose de partir des classeurs découverts au moment de l'analyse (classes, sous-systèmes, cas d'utilisation, . . .) et de les décomposer en éléments suffisamment fins pour permettre leur implémentation.
- Les classeurs ainsi décomposés s'appellent des classeurs structurés. Graphiquement, un classeur structuré se représente par un rectangle en trait plein comprenant deux compartiments. Le compartiment supérieur contient le nom du classeur et le compartiment inférieur montre les objets internes reliées par des connecteurs.



Collaborations

- Une collaboration montre des instances qui collaborent dans un contexte donné pour mettre en œuvre une fonctionnalité d'un système. Graphiquement, une collaboration se représente par une ellipse en trait pointillé comprenant deux compartiments. Le compartiment supérieur contient le nom de la collaboration et le compartiment inférieur montre les participants à la collaboration

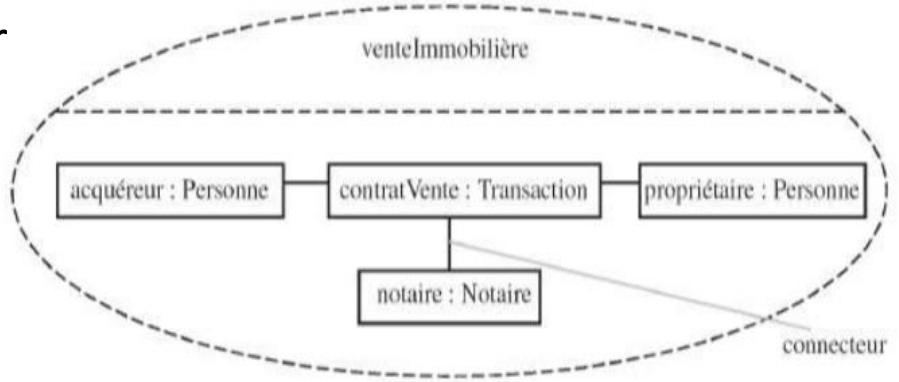


Diagramme d'interactions

- Une interaction montre le comportement d'un classeur structuré ou d'une collaboration en se focalisant sur l'échange d'informations entre les éléments du classeur ou de la collaboration. Une interaction contient un jeu de ligne de vie. Chaque ligne de vie correspond à une partie interne d'un classeur ou de la collaboration et représente une instance ou un jeu d'instances sur une période donnée. L'interaction décrit donc l'activité interne des éléments du classeur ou de la collaboration, appelés lignes de vie, et des messages qu'ils échangent.
- UML propose principalement deux diagrammes pour illustrer une interaction : le diagramme de communication et celui de séquence. Une même interaction peut être présentée aussi bien par l'un que par l'autre.
- Remarque : il existe un troisième diagramme, le diagramme de timing. Son usage est limité à la modélisation des systèmes qui s'exécutent sous de fortes contraintes de temps, comme les systèmes temps réel, il ne sera pas abordé dans ce cours.

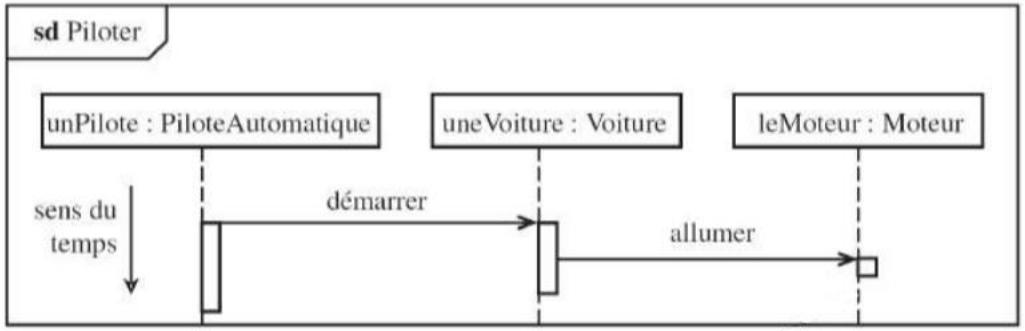


Diagramme d’interactions

Diagramme de communication

- Le diagramme de communication rend compte de l’organisation spatiale des participants à l’interaction, il est souvent utilisé pour illustrer un cas d’utilisation ou pour décrire une opération.
- Les lignes de vie sont représentées par des rectangles contenant une étiquette dont la syntaxe est : [<nom_du_rôle>] : [<Nom_du_type>]. Au moins un des deux noms doit être spécifié dans l’étiquette, les deux points (:) sont, quand à eux, obligatoires.
- Les relations entre les lignes de vie sont appelées connecteurs et se représentent par un trait plein reliant deux lignes de vies et dont les extrémités peuvent être ornées de multiplicités.
- Dans un diagramme de communication, les messages sont généralement ordonnés selon un numéro de séquence croissant. Un message est, habituellement, spécifié sous la forme suivante :

  [’[’cond’]’ [séq] [*[||] ’[’iter’]’] :] [r :=] msg([par])

  - **cond** est une condition sous forme d’expression booléenne entre crochets.
  - **séq** est le numéro de séquence du message. On numérote les messages par envoi et sous-envoi désignés par des chiffres séparés par des points : ainsi l’envoi du message 1.4.3 est antérieur à celui du message 1.4.4 mais postérieur à celui du message 1.3.5. La simultanéité d’un envoi est désignée par une lettre : les messages 1.6.a et 1.6.b sont envoyés en même temps.
  - **iter** spécifie (en langage naturel, entre crochets) l’envoi séquentiel (ou en parallèle, avec ||). On peut omettre cette spécification et ne garder que le caractère "*" pour désigner un message récurrent envoyé un certain nombre de fois.
  - **r** est la valeur de retour du message, qui sera par exemple transmise en paramètre à un autre message.
  - **msg** est le nom du message.
  - **par** désigne les paramètres (optionnels) du message.
- Cette syntaxe un peu complexe permet de préciser parfaitement l’ordonnancement et la synchronisation des messages entre les objets du diagramme de communication. La direction d’un message est spécifiée par une flèche pointant vers l’un ou l’autre des objets de l’interaction, reliés par ailleurs avec un connecteur.

Diagramme d’interactions

Diagramme de séquences

- 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.
- Une ligne de vie se représente par un rectangle, auquel est accroché une ligne verticale pointillée, contenant une étiquette dont la syntaxe est : [<nom_du_rôle>] : [<Nom_du_type>]. Au moins un des deux noms doit être spécifié dans l’étiquette, les deux points (:) sont, quand à eux, obligatoires.
- Un message définit une communication particulière entre des lignes de vie. Plusieurs types de messages existent, les plus commun 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 serra traité par le destinataire. Un signal est, par définition, un message asynchrone.
- 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.
- 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
  - 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 (figure 7.7).
  - 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

Diagramme d’interactions

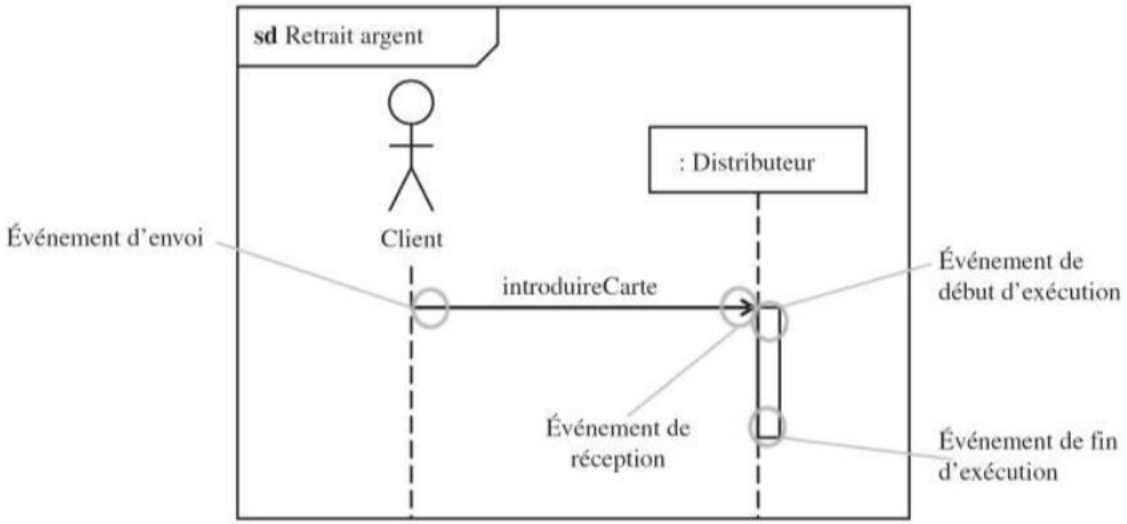
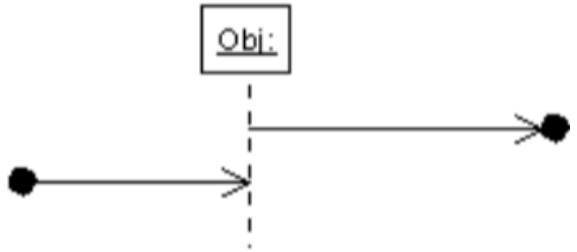


Diagramme d'interactions

Diagramme de séquences

- UML permet de séparer clairement l'envoi du message, sa réception, ainsi que le début de l'exécution de la réaction et sa fin. 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. La syntaxe de ces messages est la même que pour un diagramme de communication excepté deux points :
  - la direction du message est directement spécifiée par la direction de la flèche qui matérialise le message, et non par une flèche supplémentaire au dessus du connecteur reliant les objets comme c'est le cas dans un diagramme de communication ;
  - les numéros de séquence sont généralement omis puisque l'ordre relatif des messages est déjà matérialisé par l'axe vertical qui représente l'écoulement du temps.
- La syntaxe de réponse à un message est la suivante, où message représente le message d'envoi :

  [<attribut> =] message [: <valeur_de_retour>]
- Un message complet est tel que les événements d'envoi et de réception sont connus. Il se représente par une simple flèche dirigée de l'émetteur vers le récepteur.
- Un message perdu est tel que l'événement d'envoi est connu, mais pas l'événement de réception. Il se représente par une flèche qui pointe sur une petite boule noire.
- Un message trouvé est tel que l'événement de réception est connu, mais pas l'événement d'émission. Une flèche partant d'une petite boule noire représente un message trouvé.

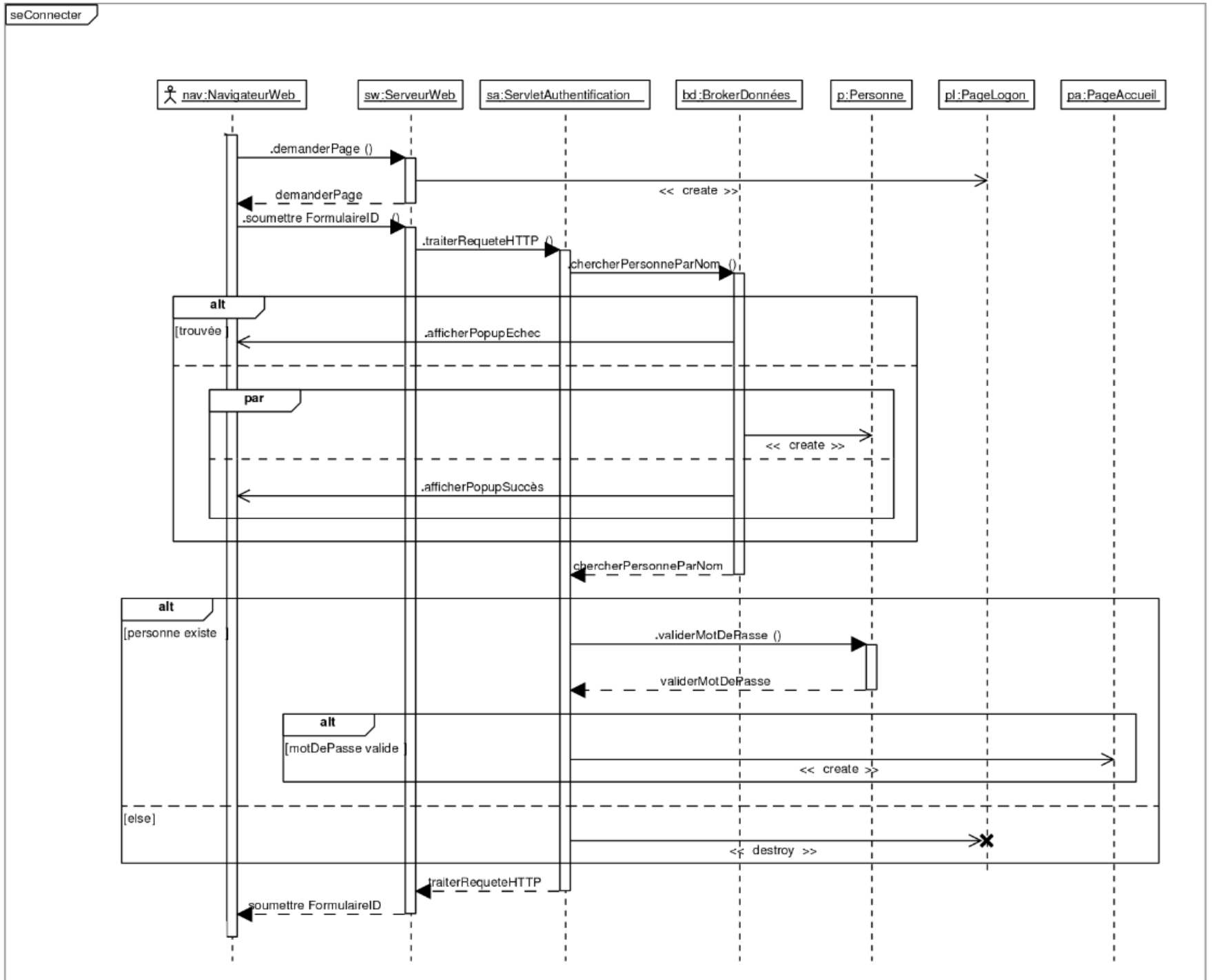


Exercice

- Expliquez la syntaxe des messages suivants extraits d'un diagramme de communication.
  - f
  - y := f(x)
  - 1 : f
  - [x>0] : f
  - *[x>0] : f
  - *[i :=0..10] : f
  - 1 *[i :=0..10] : f

Exercice

- Donnez le diagramme de communication équivalent au diagramme de séquence suivant



Mise en œuvre d'UML

- La problématique que pose la mise en œuvre d'UML est simple : comment passer de l'expression des besoins au code de l'application ? UML n'est qu'un langage de modélisation, ce n'est pas une méthode. En effet, UML ne propose pas une démarche de modélisation explicitant et encadrant toutes les étapes d'un projet, de la compréhension des besoins à la production du code de l'application. Une méthode se doit de définir une séquence d'étapes, partiellement ordonnées, dont l'objectif est de produire un logiciel de qualité qui répond aux besoins des utilisateurs dans des temps et des coûts prévisibles.
- Bien qu'UML ne soit pas une méthode, ses auteurs précisent néanmoins qu'une méthode basée sur l'utilisation UML doit être :
  - Pilotée par les cas d'utilisation : La principale qualité d'un logiciel étant son utilité, c'est-à-dire son adéquation avec les besoins des utilisateurs, toutes les étapes, de la spécification des besoins à la maintenance, doivent être guidées par les cas d'utilisation qui modélisent justement les besoins des utilisateurs.
  - Centrée sur l'architecture : L'architecture est conçue pour satisfaire les besoins exprimés dans les cas d'utilisation, mais aussi pour prendre en compte les évolutions futures et les contraintes de réalisation. La mise en place d'une architecture adaptée conditionne le succès d'un développement. Il est important de la stabiliser le plus tôt possible.
  - Itérative et incrémentale : L'ensemble du problème est décomposé en petites itérations, définies à partir des cas d'utilisation et de l'étude des risques. Les risques majeurs et les cas d'utilisation les plus importants sont traités en priorité. Le développement procède par des itérations qui conduisent à des livraisons incrémentales du système. Nous avons déjà présenté le modèle de cycle de vie par incrément dans la section

Mise en œuvre d'UML

- Nous allons voir ici une méthode simple et générique qui se situe à mi-chemin entre UP (Unified Process), qui constitue un cadre général très complet de processus de développement, et XP (eXtreme Programming) qui est une approche minimaliste centrée sur le code. Cette méthode est issue de celle présentée par Roques (2002) et qui est :
  - conduite par les cas d'utilisation, comme UP, mais bien plus simple ;
  - relativement légère et restreinte, comme XP, mais sans négliger les activités de modélisation en analyse et conception ;
  - fondée sur l'utilisation d'un sous-ensemble nécessaire et suffisant du langage UML (modéliser 80% des problèmes en utilisant 20% d'UML)
- Dans tous les cas, il faut garder à l'esprit qu'une méthode n'est pas une formule magique. Le fait de produire des diagrammes UML selon un ordre établi n'est en aucun cas une garantie de réussite. Une méthode ne sert qu'à canaliser et ordonner les étapes de la modélisation. La valeur n'est pas dans la méthode mais dans les personnes qui la mettent en œuvre.

Mise en œuvre d'UML

1 Identification et représentation des besoins : diagramme de cas d'utilisation

- Les cas d'utilisation sont utilisés tout au long du projet. Dans un premier temps, on les crée pour identifier et modéliser les besoins des utilisateurs. Ces besoins sont déterminés à partir des informations recueillies lors des rencontres entre informaticiens et utilisateurs. Il faut impérativement proscrire toute considération de réalisation lors de cette étape.
- Durant cette étape, on détermine les limites du système, identifie les acteurs et recense les cas d'utilisation. Si l'application est complexe, vous pourrez organiser les cas d'utilisation en paquetages.
- Dans le cadre d'une approche itérative et incrémentale, il faut affecter un degré d'importance et un coefficient de risque à chacun des cas d'utilisation pour définir l'ordre des incréments à réaliser. Les interactions entre les acteurs et le système (au sein des cas d'utilisation) seront explicitées sous forme textuelle et sous forme graphique au moyen de diagrammes de séquence. Les utilisateurs ont souvent beaucoup de difficultés à exprimer clairement et précisément ce qu'ils attendent du système. L'objectif de cette étape et des deux suivantes est justement de les aider à formuler et formaliser ces besoins.

2 Spécification détaillée des besoins : diagrammes de séquence système

- Dans cette étape, on cherche à détailler la description des besoins par la description textuelle des cas d'utilisation et la production de diagrammes de séquence système illustrant cette description textuelle, puisque nous somme toujours dans la spécification des besoins.
- Les scénarii de la description textuelle des cas d'utilisation peuvent être vus comme des instances de cas d'utilisation et sont illustrés par des diagrammes de séquence système. Il faut, au minimum, représenter le scénario nominal de chacun des cas d'utilisation par un diagramme de séquence qui rend compte de l'interaction entre l'acteur, ou les acteurs, et le système. Le système est ici considéré comme un tout et est représenté par une ligne de vie. Chaque acteur est également associé à une ligne de vie.
- Lorsque les scénarii alternatifs d'un cas d'utilisation sont nombreux et importants, l'utilisation d'un diagramme d'états-transitions ou d'activités peut s'avérer préférable à une multitude de diagrammes de séquence

Mise en œuvre d'UML

3 Maquette de l'IHM de l'application (non couvert par UML)

- Une maquette d'IHM (Interface Homme-Machine) est un produit jetable permettant aux utilisateurs d'avoir une vue concrète mais non définitive de la future interface de l'application . La maquette peut très bien consister en un ensemble de dessins produits par un logiciel de présentation ou de dessin. Par la suite, la maquette pourra intégrer des fonctionnalités de navigation permettant à l'utilisateur de tester l'enchaînement des écrans ou des menus, même si les fonctionnalités restent fictives. La maquette doit être développée rapidement afin de provoquer des retours de la part des utilisateurs.

4 Analyse du domaine : modèle du domaine

- La modélisation des besoins par des cas d'utilisation s'apparente à une analyse fonctionnelle classique. L'élaboration du modèle des classes du domaine permet d'opérer une transition vers une véritable modélisation objet. L'analyse du domaine est une étape totalement dissociée de l'analyse des besoins. Elle peut être menée avant, en parallèle ou après cette dernière.
- La phase d'analyse du domaine permet d'élaborer la première version du diagramme de classes appelée modèle du domaine. Ce modèle doit définir les classes qui modélisent les entités ou concepts présents dans le domaine (on utilise aussi le terme de métier) de l'application. Il s'agit donc de produire un modèle des objets du monde réel dans un domaine donné. Ces entités ou concept peuvent être identifiés directement à partir de la connaissance du domaine ou par des entretiens avec des experts du domaine. Il faut absolument utiliser le vocabulaire du métier pour nommer les classes et leurs attributs. Les classes du modèle du domaine ne doivent pas contenir d'opérations, mais seulement des attributs. Les étapes à suivre pour établir ce diagramme sont :
    - identifier les entités ou concepts du domaine ;
    - identifier et ajouter les associations et les attributs ;
    - organiser et simplifier le modèle en éliminant les classes redondantes et en utilisant l'héritage ;
    - le cas échéant, structurer les classes en paquetage selon les principes de cohérence et d'indépendance.
- L'erreur la plus courante lors de la création d'un modèle du domaine consiste à modéliser un concept par un attribut alors que ce dernier devait être modélisé par une classe. Si la seule chose que recouvre un concept est sa valeur, il s'agit simplement d'un attribut. Par contre, si un concept recouvre un ensemble d'informations, alors il s'agit plutôt d'une classe qui possède elle-même plusieurs attributs.

Mise en œuvre d'UML

5 Diagramme de classes participantes

- Le diagramme de classes participantes est particulièrement important puisqu'il effectue la jonction entre, d'une part, les cas d'utilisation, le modèle du domaine et la maquette, et d'autre part, les diagrammes de conception logicielle que sont les diagrammes d'interaction et le diagramme de classes de conception. Il n'est pas souhaitable que les utilisateurs interagissent directement avec les instances des classes du domaine par le biais de l'interface graphique. En effet, le modèle du domaine doit être indépendant des utilisateurs et de l'interface graphique. De même, l'interface graphique du logiciel doit pouvoir évoluer sans répercussion sur le cœur de l'application. C'est le principe fondamental du découpage en couches d'une application. Ainsi, le diagramme de classes participantes modélise trois types de classes d'analyse :
  - Les classes de dialogues – Les classes qui permettent les interactions entre l'IHM et les utilisateurs sont qualifiées de dialogues. Ces classes sont directement issues de l'analyse de la maquette présentée. Il y a au moins un dialogue pour chaque association entre un acteur et un cas d'utilisation du diagramme de cas d'utilisation. En général, les dialogues vivent seulement le temps du déroulement du cas d'utilisation concerné.
  - Les classes de contrôles – Les classes qui modélisent la cinématique de l'application sont appelées contrôles. Elles font la jonction entre les dialogues et les classes métier en permettant au différentes vues de l'application de manipuler des informations détenues par un ou plusieurs objets métier. Elles contiennent les règles applicatives et les isolent à la fois des dialogues et des entités.
  - Les classes entités – Les classes métier, qui proviennent directement du modèle du domaine sont qualifiées d'entités. Ces classes sont généralement persistantes, c'est-à-dire qu'elles survivent à l'exécution d'un cas d'utilisation particulier et qu'elles permettent à des données et des relations d'être stockées dans des fichiers ou des bases de données. Lors de l'implémentation, ces classes peuvent ne pas se concrétiser par des classes mais par des relations, au sens des bases de données relationnelles
- Lors de l'élaboration du diagramme de classes participantes, il faut veiller au respect des règles suivantes :
  - Les entités, qui sont issues du modèle du domaine, ne comportent que des attributs
  - Les entités ne peuvent être en association qu'avec d'autres entités ou avec des contrôles, mais, dans ce dernier cas, avec une contrainte de navigabilité interdisant de traverser une association d'une entité vers un contrôle.
  - Les contrôles ne comportent que des opérations. Ils implémentent la logique applicative (i.e. les fonctionnalités de l'application), et peuvent correspondre à des règles transverses à plusieurs entités. Chaque contrôle est généralement associé à un cas d'utilisation, et vice versa. Mais rien n'empêche de décomposer un cas d'utilisation complexe en plusieurs contrôles.
  - Les contrôles peuvent être associés à tous les types de classes, y compris d'autres contrôles. Dans le cas d'une association entre un dialogue et un contrôle, une contrainte de navigabilité doit interdire de traverser l'association du contrôle vers le dialogue.
  - Les dialogues comportent des attributs et des opérations. Les attributs représentent des informations ou des paramètres saisis par l'utilisateur ou des résultats d'actions. Les opérations réalisent (généralement par délégation aux contrôles) les actions que l'utilisateur demande par le biais de l'IHM.
  - Les dialogues peuvent être en association avec des contrôles ou d'autres dialogues, mais pas directement avec des entités.
  - Il est également possible d'ajouter les acteurs sur le diagramme de classes participantes en respectant la règle suivante : un acteur ne peut être lié qu'à un dialogue.
- Certaines classes possèdent un comportement dynamique complexe. Ces classes auront intérêt à être détaillées par des diagrammes d'états-transitions.
- L'attribution des bonnes responsabilités aux bonnes classes est l'un des problèmes les plus délicats de la conception orientée objet. Lors de la phase d'élaboration du diagramme de classes participantes, le chef de projet a la possibilité de découper le travail de son équipe d'analystes par cas d'utilisation. L'analyse et l'implémentation des fonctionnalités dégagées par les cas d'utilisation définissent alors les itérations à réaliser. L'ordonnancement des itérations étant défini par le degré d'importance et le coefficient de risque affecté à chacun des cas d'utilisation.

Mise en œuvre d'UML

6 Diagramme d'activités de navigation

- Les IHM modernes facilitent la communication entre l'application et l'utilisateur en offrant toute une gamme de moyens d'action et de visualisation comme des menus déroulants ou contextuels, des palettes d'outils, des boîtes de dialogues, des fenêtres de visualisation, etc. Cette combinaison possible d'options d'affichage, d'interaction et de navigation aboutis aujourd'hui à des interfaces de plus en plus riches et puissantes.
- UML offre la possibilité de représenter graphiquement cette activité de navigation dans l'interface en produisant des diagrammes dynamiques. On appelle ces diagrammes des diagrammes de navigation. Le concepteur à le choix d'opter pour cette modélisation entre des diagrammes d'états-transitions et des diagrammes d'activités. Les diagrammes d'activités constituent peut-être un choix plus souple et plus judicieux.
- Les diagrammes d'activités de navigation sont à relier aux classes de dialogue du diagramme de classes participantes. Les différentes activités du diagramme de navigation peuvent être stéréotypées en fonction de leur nature : « fenêtre », « menu », « menu contextuel », « dialogue », etc. La modélisation de la navigation à intérêt à être structurée par acteur.

7 Diagramme d'interaction

- Maintenant, il faut attribuer précisément les responsabilités de comportement (diagrammes de séquence système) aux classes d'analyse du diagramme de classes participantes. Les résultats de cette réflexion sont présentés sous la forme de diagrammes d'interaction UML. Inversement, l'élaboration de ces diagrammes facilite grandement la réflexion.
- Parallèlement, une première ébauche du diagramme de classes de conception est construite et complétée. Durant cette phase, l'ébauche du diagramme de classes de conception reste indépendante des choix technologiques qui seront faits ultérieurement. Pour chaque service ou fonction, il faut décider quelle est la classe qui va le contenir. Les diagrammes d'interactions sont particulièrement utiles au concepteur pour représenter graphiquement l'allocation des responsabilités. Chaque diagramme représente un ensemble d'objets de classes différentes collaborant dans le cadre d'un scénario d'exécution du système.
- Dans les diagrammes d'interaction, les objets communiquent en s'envoyant des messages qui invoquent des opérations sur les objets récepteurs. Il est ainsi possible de suivre visuellement les interactions dynamiques entre objets et les traitements réalisés par chacun d'eux. Par rapport au diagrammes de séquences système on remplace ici le système, vu comme une boîte noire, par un ensemble d'objets en collaboration(dialogues, contrôles et entités). Les règles s'appliquant au diagramme des classes participantes doivent cependant être transposées car, pour que deux objets puissent interagir, il faut que :
    - les classes dont ils sont issus soient en association dans le diagramme de classes participantes ;
    - l'interaction respecte la navigabilité de l'association en question

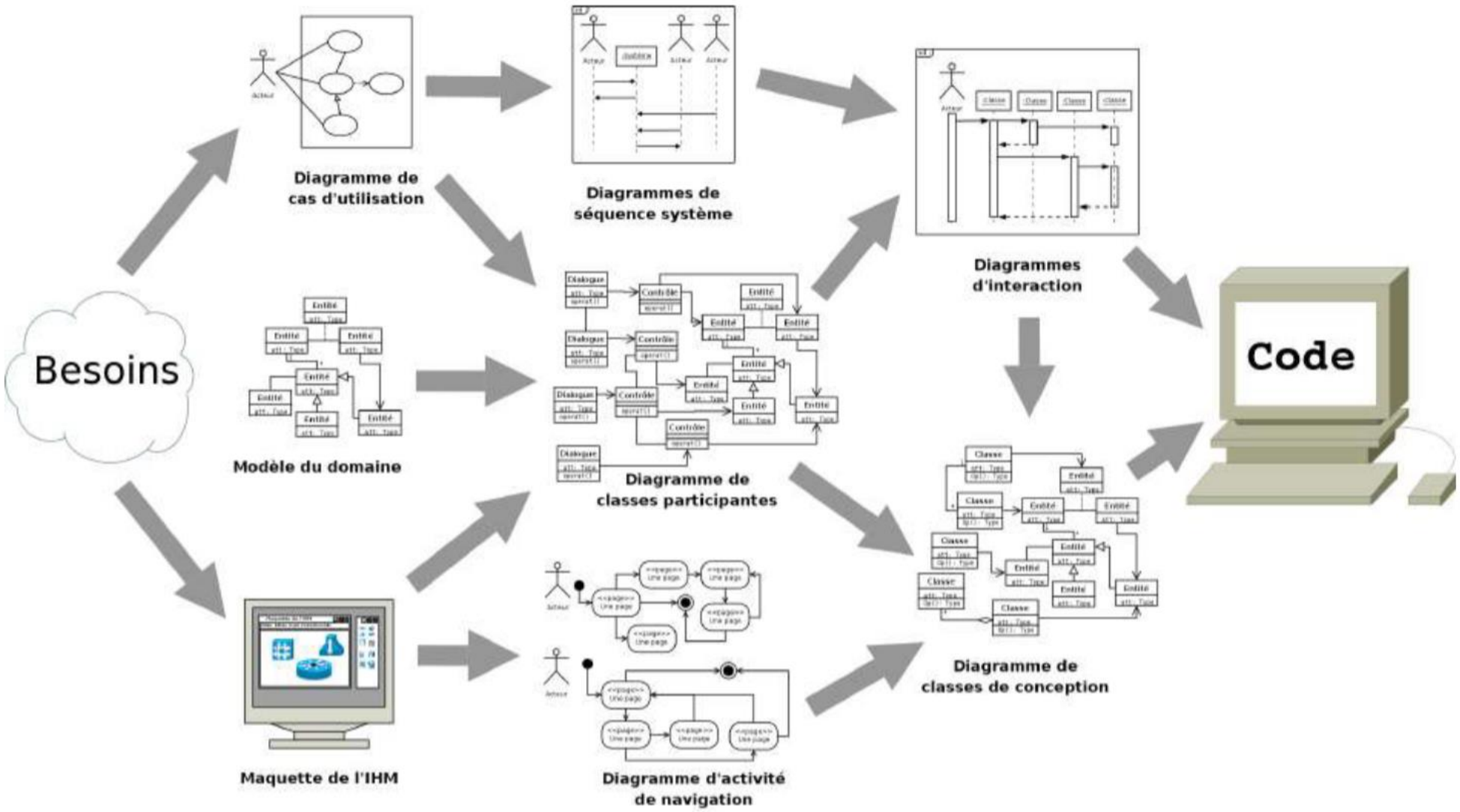
Mise en œuvre d'UML

8 Diagramme des classes de conception

- L'objectif de cette étape est de produire le diagramme de classes qui servira pour l'implémentation. Une première ébauche du diagramme de classes de conception a déjà été élaborée en parallèle du diagrammes d'interaction. Il faut maintenant le compléter en précisant les opérations privées des différentes classes. Il faut prendre en comptes les choix techniques, comme le choix du langage de programmation, le choix des différentes librairies utilisées (notamment pour l'implémentation de l'interface graphique), etc.
- Pour une classe, le couplage est la mesure de la quantité d'autre classes auxquelles elle est connectée par des associations, des relations de dépendances, etc. Durant toute l'élaboration du diagramme de classes de conception, il faut veiller à conserver un couplage faible pour obtenir une application plus évolutive et plus facile à maintenir. L'utilisation des design patterns est fortement conseillée lors de l'élaboration du diagramme de classes de conception. Parfois, les classes du type entités ont intérêt à être implémentées dans une base de données relationnelle.

9 Implémentation

Mise en œuvre d'UML

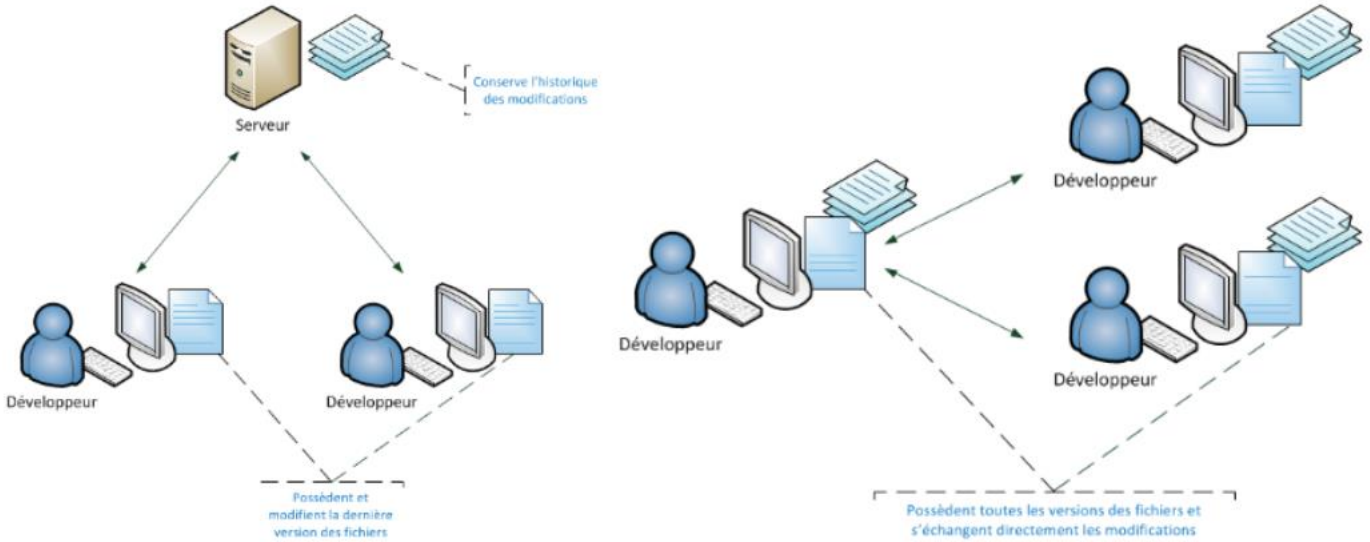


Système de suivi de versions

- En travaillant en commun sur un projet avec support numérique (éventuellement projet personnel), on en vient fréquemment à se poser ces questions :
  - Qui a modifié le fichier X, il marchait bien avant et maintenant il provoque des bugs ! ;
  - Quelqu'un peut m'aider en travaillant sur le fichier X pendant que je travaille sur le fichier Y ?
  - Attention à ne pas toucher au fichier Y car si on travaille dessus en même temps je risque d'écraser les modifications !
  - Qui a ajouté cette ligne de code dans ce fichier ? Elle ne sert à rien !
  - A quoi servent ces nouveaux fichiers et qui les a ajoutés au code du projet ?
  - Quelles modifications avions-nous faites pour résoudre le bug ?
- La gestion des versions est un travail fastidieux et méthodique. Un système de suivi de versions permet d'automatiser ces tâches

Système de suivi de versions

- Un système de suivi de versions (version control system) est un ensemble d'outils logiciels pou mémoriser, faciliter le travail collaboratif et retrouver différentes versions d'un projet. Cet outil permet de faciliter l'échange de fichiers, de tracer les modifications et de gérer les conflits de versions.
- Chaque modification du projet est mémorisée, avec une justification. De plus, la fusion des modifications est quasi automatique si deux personnes travaillent sur un même fichier.
- Un logiciel de suivi de versions peut être centralisé, avec un serveur qui sert de point de rencontre entre les développeurs et qui enregistre l'historique des versions, ou distribué, c'est-à-dire sans serveur, où les développeurs conservent l'historique des modifications et se transmettent les nouveautés.



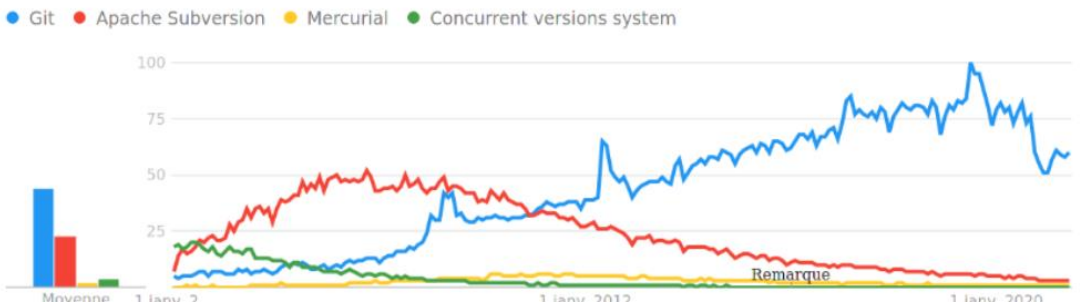
Système de suivi de versions

- Il en existe plusieurs, dont Mercurial et Git sont les plus utilisés, mais aussi SVN ou Source Safe, un peu dépréciés aujourd'hui

- Git a été développé par Linus Torvalds pur faciliter le développement du noyau Linux. C'est un logiciel libre et open source disponible sur toutes les plateformes.

git	mercurial	SUBVERSION
libre	licence GNU / GPL	libre
décentralisé	décentralisé	centralisé
branches (+)	branches (+)	branches (-)
cryptage SHA1	cryptage SHA1	pas de cryptage
métadonnées (.git)		stockages de fichiers
14M. users	m̂ principe que Git	
Gitlab		
grosse communauté		

https://en.wikipedia.org/wiki/Comparison_of_version-control_software



Système de suivi de versions

Installer Git

- Sous Linux : `sudo apt-get install git-core gitk –y`

Gitk est une interface graphique qui aide à visualiser les logs (facultatif).

- Sous Windows : il faut installer msysgit : https://gitforwindows.org. Cela installe msys, un système d'émulation des commandes unix sous windows et git simultanément. Lancer ensuite une console **git bash,** avec les commandes de base unix. C'est un émulateur.
- Sous MacOS (avec homebrew) : `brew install git`. Git est déjà fourni avec Xcode

Système de suivi de versions

Installer Git

- Dans la console, activer la couleur :

```
git config --global color.diff auto
git config --global color.status auto
git config --global color.branch auto
```

- Configurer votre nom (ou pseudo) et votre email :

```
git config --global user.name "votre_pseudo"
git config --global user.email "moi@email.com"
```

- Configurer l'éditeur que git vous présentera pour rédiger les messages de commit (ici, je propose emacs, mais on peut aussi choisir d'autres alternatives comme vim, sublimetext) :

```
git config --global core.editor emacs
```

- Pour créer un dépôt :

```
cd /home/comet/GIT/
mkdir monPremierDepot
cd monPremierDepot
```

- Si votre projet existe déjà, inutile d'en créer un nouveau.
- Enfin, initialisez le dépôt git avec la commande :

```
git init
```

- Un dossier caché .git vient également d'être créé.

Système de suivi de versions

- Un dépôt git est un répertoire (ou dossier) dans lequel toute modification peut être suivie dans le temps. Tous les sous-répertoires font partie du dépôt.
- Toute commande git cherche à savoir s'il se trouve dans un dépôt ou non, en testant la présence d'un sous-répertoire caché .git/ ou en remontant l'arborescence.
- Tout fichier peut être dans l'un des 3 états possibles suivants :
  - modifié : Le fichier a été localement modifié, mais git n'a pas validé les différences dans sa base de données locale. Les fichiers dans cet état sont dans le répertoire de travail (ou working directory).
  - indexé : Le fichier fait partie des modifications qui sont prêtes à être validées pour le prochain commit (instantané). Les fichiers dans cet état sont dans l'index (ou dans le staging area). Commande : **git add**
  - validé : Les données dans cet état sont en sécurité dans la base de données locale de git (aussi appelée git directory). Les modifications ainsi sauvegardées font partie d'un instantané du projet. Commande : **git commit**

Système de suivi de versions

- git enregistre dans une BD locale une succession de commits. Un commit a plusieurs caractéristiques fondamentales :
  - unicité : Un commit est unique et identifié par un numéro de série
  - auteur : Un commit est lié à un auteur (nom + adresse mail)
  - date : Un commit est horodaté
  - commentaire : Un instantané est obligatoirement accompagné d'un commentaire de son auteur.
- Un commit avec git est local : personne n'est au courant. Il est donc possible d'annuler un commit erroné.
- L'auteur, la date et l'identifiant du commit sont automatiquement ajoutés par git.
- Pour annuler tous les changements faits et pas encore commit :

  ```
  git revert
  ```
- Pour supprimer le dernier commit (créer un nouveau commit qui fait l'inverse) :

  ```
  git reset -hard
  ```
- Pour revenir à la dernière version commit du projet :

  ```
  git checkout master
  ```

Système de suivi de versions

- GitHub est un service en ligne qui permet d'héberger des dépôts (repository) de code. GitHub est un outil gratuit pour héberger du code open source, et propose également des plans payants pour les projets de code privés. C'est le numéro 1 mondial et il héberge plus d'une dizaine de millions de repositories !
- Pour créer votre compte GitHub, rendez-vous sur sa page d'accueil où vous pourrez entrer un nom d'utilisateur, un mot de passe, etc. Une fois votre compte créé, vous aurez accès à votre dashboard et découvrirez toutes les fonctionnalités de GitHub.
- À partir de GitHub, vous pouvez copier un repository sur votre machine. Pour cela, il vous suffit de rechercher le repository qui vous intéresse sur GitHub, de vous y placer, puis d'utiliser l'option "clone URL" en bas à droite de l'écran. Cette option vous propose un lien SSH, HTTPS ou Subversion et utiliser la commande :

```
git clone lienFourniParGitHub
```

Système de suivi de versions

- Pour synchroniser les modifications que vous faites dans votre dépôt sur votre machine avec votre repo sur GitHub :
  - Ouvrez votre terminal et placez-vous dans votre dépôt local.
  - Faites un/des commit(s) des modifications que vous avez ajoutées sur ce repo.
  - Envoyez ces modifications sur votre dépôt GitHub en utilisant la commande

    ```
    git push origin master
    ```
- La branche master est la branche qui contient le code courant de votre dépôt GitHub. On reviendra plus tard sur le concept de branche.
- Le remote sur lequel vous envoyez votre code est appelé origin par défaut. Ici, ce remote est GitHub. Si vous aviez plusieurs remotes (par exemple, votre téléphone portable un 2e ordinateur, ou encore un autre service de gestion de code), vous pourriez envoyer votre code sur un remote “téléphone” ou “ordi2”, ou “bitbucket”
- Une fois que vous avez push votre code en ligne, vous pouvez aller consulter votre dépôt sur GitHub. Vous y retrouverez les derniers commits effectués en cliquant sur l’option “Commits” dans votre dépôt.

Système de suivi de versions

- Pour récupérer en local les dernières modifications du dépôt GitHub, on utilise la commande
  `git pull`
- Pensez à synchroniser régulièrement votre code local avec vos dépôts en ligne à l'aide des commandes `git push` et `git pull`. C'est particulièrement important lorsque vous travaillez à plusieurs sur un projet, pour que tout le monde avance sur la même base.
- Lorsque plusieurs utilisateurs souhaitent modifier un même fichier, des conflits peuvent apparaitre. Il arrive très souvent qu'il y aie des conflits, qui empêchent de fusionner les modifications effectuées sur un fichier, par exemple lorsque plusieurs personnes travaillent en même temps sur un même fichier. Git va reconnaître qu'il existe un conflit entre les fichiers et lance un message d'erreur. Il faut alors modifier les fichiers litigieux, puis faire un commit sans message.

Système de suivi de versions

- Les branches permettent de travailler sur des versions de code qui divergent de la branche principale contenant votre code courant. Travailler sur plusieurs branches est très utile lorsque vous souhaitez tester un expérimentation sur votre projet, ou encore pour vous concentrer sur le développement d'une fonctionnalité spécifique. Lorsque vous initialisez un repo Git, votre code est placé dans la branche principale appelée master par défaut.
- Pour voir les branches présentes dans un dépôt, on utilise la commande `git branch`. Elle retourne les branches présentes, et ajoutera une étoile devant la branche dans laquelle vous êtes placés.
- Pour créer une nouvelle branche, il vous suffit d'ajouter le nom de la branche à créer à la suite de la commande précédente :
  `git branch new-branch`
- Pour se placer dans une autre branche du dépôt, on ajoute le mot-clef checkout
  `git checkout new-branch`
- Lorsque le travail sur une branche est achevé, il faut remettre les modifications faites sur les différentes branches dans la branche principale master. Pour ajouter à la branche principale ces modifications, on fusionne les branches. On se place dans la branche A :
  `git checkout brancheA`

  Puis on utilise la commande git merge :

  `git merge brancheB`
- La gestion des conflits éventuels (p ex un fichier avec le même nom mais un contenu différent) est similaire au conflit de versions sur la même branche : il faut choisir la version à garder puis faire un commit sans message

Système de suivi de versions

- Pour retrouver qui a modifié une ligne précise de code dans un projet, faire une recherche avec `git log` peut s'avérer compliqué, surtout si le projet contient beaucoup de commits. Il existe un autre moyen plus direct de retrouver qui a fait une modification particulière dans un fichier : la commande
  `git blame fichier`
- Cette commande liste toutes les modifications qui ont été faites sur le fichier ligne par ligne. À chaque modification est associé le début du sha du commit correspondant. On peut ensuite utiliser la commande git show qui renvoie directement les détails du commit recherché en saisissant le début de son sha :

  ```
  git show 05b1233
  ```

Architecture logicielle

- L'architecture est « l'art de construire les édifices ». Ce mot est avant tout lié au domaine du génie civil : on pense à l'architecture d'un monument ou encore d'un pont. Par analogie, l'architecture logicielle peut être définie comme étant « l'art de construire les logiciels ». Selon le contexte, l'architecture logicielle peut désigner :
  - L'activité d'architecture, c'est-à-dire une phase au cours de laquelle on effectue les grands choix qui vont structurer une application : langages et technologies utilisés, découpage en sous-parties, méthodologies mises en œuvre...
  - Le résultat de cette activité, c'est-à-dire la structure d'une l'application, son squelette. Dans le domaine du génie civil, on n'imagine pas se lancer dans la construction d'un bâtiment sans avoir prévu son apparence, étudié ses fondations et son équilibre, choisi les matériaux utilisés, etc. Dans le cas contraire, on va au devant de graves désillusions..
- Cette problématique se retrouve dans le domaine informatique. Comme un bâtiment, un logiciel est fait pour durer dans le temps. Il est presque systématique que des projets informatiques aient une durée de vie de plusieurs années. Plus encore qu'un bâtiment, un logiciel va, tout au long de son cycle de vie, connaître de nombreuses modifications qui aboutiront à la livraison de nouvelles versions, majeures ou mineures. Les évolutions par rapport au produit initialement créé sont souvent nombreuses et très difficiles à prévoir au début du projet. Par exemple VLC n'était à l'origine qu'un projet étudiant destiné à diffuser des vidéos sur le campus de l'Ecole Centrale de Paris. Sa première version remonte à l'année 2001.

Architecture logicielle

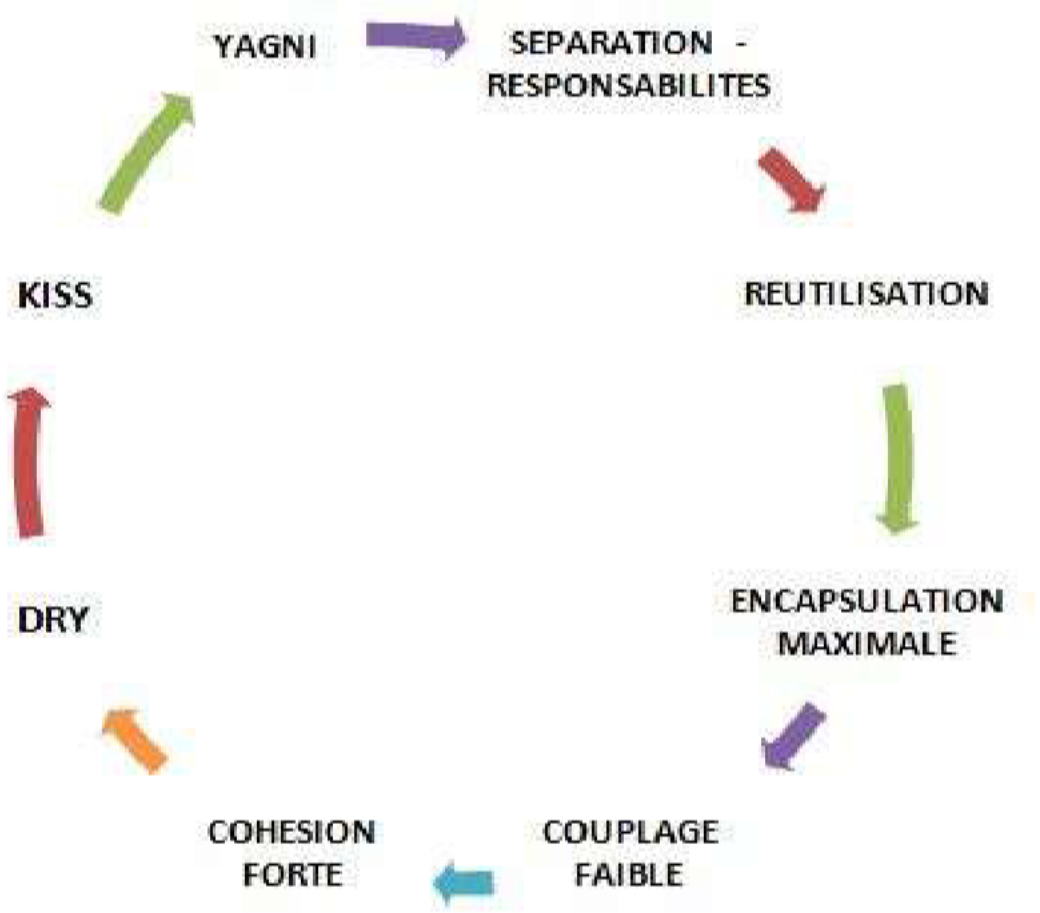
- Les objectifs de l'architecture sont que le logiciel créé réponde aux besoins et résiste aux nombreuses modifications qu'il subira au cours de son cycle de vie. Contrairement à un bâtiment, un logiciel mal pensé ne risque pas de s'effondrer. En revanche, une mauvaise architecture peut faire exploser le temps nécessaire pour réaliser les modifications, et donc leur coût. Les deux objectifs principaux de toute architecture logicielle sont la réduction des coûts (création et maintenance) et l'augmentation de la qualité du logiciel. La qualité du code source d'un logiciel peut être évaluée par un certain nombre de mesures appelées métriques de code : indice de maintenabilité, complexité cyclomatique, etc.
- Il n'existe pas de vrai consensus concernant le sens des mots « architecture » et « conception » dans le domaine du développement logiciel. Ces deux termes sont souvent employés de manière interchangeable. Il arrive aussi que l'architecture soit appelée « conception préliminaire » et la conception proprement dite « conception détaillée ». Certaines méthodologies de développement incluent la définition de l'architecture dans une phase plus globale appelée « conception ». Cela dite, la distinction suivante est généralement admise et sera utilisée dans le cadre de ce cours :
  - L'architecture logicielle (software architecture) considère le logiciel de manière globale. Il s'agit d'une vue de haut niveau qui définit le logiciel dans ses grandes lignes : que fait-il ? Quelles sont les sous-parties qui le composent ? Interagissent- elles ? Sous quelle forme sont stockées ses données ? Etc.
  - La conception logicielle (software design) intervient à un niveau de granularité plus fin et permet de préciser comment fonctionne chaque sous-partie de l'application. Quel logiciel est utilisé pour stocker les données ? Comment est organisé le code ? Comment une sous-partie expose-t-elle ses fonctionnalités au reste du système ? Etc.
- La perspective change selon la taille du logiciel et le niveau auquel on s'intéresse à lui :
  - sur un projet de taille modeste, architecture et conception peuvent se confondre.
  - à l'inverse, certaines sous-parties d'un projet de taille conséquente peuvent nécessiter en elles-mêmes un travail d'architecture qui, du point de vue de l'application globale, relève plutôt de la conception...

Architecture logicielle

- Tout logiciel, au-delà d'un niveau minimal de complexité, est un édifice qui mérite une phase de réflexion initiale pour l'imaginer dans ses grandes lignes. Au cours de cette phase, on effectue les grands choix structurant le futur logiciel : langages, technologies, outils... Elle consiste notamment à identifier les différents éléments qui vont composer le logiciel et à organiser les interactions entre ces éléments. Selon le niveau de complexité du logiciel, l'activité d'architecture peut être une simple formalité ou bien un travail de longue haleine. L'activité d'architecture peut donner lieu à la production de diagrammes représentant les éléments et leurs interactions selon différents formalismes, par exemple UML.
- L'activité d'architecture intervient traditionnellement vers le début d'un projet logiciel, dès le moment où les besoins auxquels le logiciel doit répondre sont suffisamment identifiés. Elle est presque toujours suivie par une phase de conception. Les évolutions d'un projet logiciel peuvent nécessiter de nouvelles phases d'architecture tout au long de sa vie. C'est notamment le cas avec certaines méthodologies de développement itératif ou agile, où des phases d'architecture souvent brèves alternent avec des phases de production, de test et de livraison.
- De manière très générale, un logiciel sert à automatiser des traitements sur des données. Toute application informatique est donc confrontée à trois problématiques :
  - la problématique de présentation : consiste à gérer les interactions avec l'extérieur, en particulier l'utilisateur : saisie et contrôle de données, affichage.
  - la problématique des traitements : consiste à effectuer sur les données des opérations (calculs) en rapport avec les règles métier.
  - la problématique des données : consiste à accéder et stocker les informations qu'il manipule, notamment entre deux utilisations.
- La phase d'architecture d'une application consiste aussi à choisir comment sont gérées ces trois problématiques, autrement dit à les répartir dans l'application créée.
- Le résultat de l'activité du même nom, l'architecture d'un logiciel décrit sa structure globale, son squelette. Elle décrit les principaux éléments qui composent le logiciel, ainsi que les flux d'échanges entre ces éléments. Elle permet à l'équipe de développement d'avoir une vue d'ensemble de l'organisation du logiciel, et constitue donc en elle-même une forme de documentation. On peut décrire la structure d'un logiciel selon différents points de vue. Entre autres, une vue logique mettre l'accent sur le rôle et les responsabilités de chaque partie du logiciel. Une vue physique présentera les processus, les machines et les liens réseau nécessaires

Conception logicielle

- Cette partie présente les grands principes qui doivent guider la création d'un logiciel, et plus généralement le travail du développeur au quotidien



Conception logicielle

- Le principe de séparation des responsabilités (separation of concerns) vise à organiser un logiciel en plusieurs sous-parties, chacune ayant une responsabilité bien définie. Ce principe est sans doute le principe de conception le plus essentiel. Ainsi construite de manière modulaire, l'application sera plus facile à comprendre et à faire évoluer. Au moment où un nouveau besoin se fera sentir, il suffira d'intervenir sur la ou les sous-parties concernées. Le reste de l'application sera inchangée : cela limite les tests à effectuer et le risque d'erreur. Une construction modulaire encourage également la réutilisation de certaines parties de l'application.
- Le principe de responsabilité unique (single responsibility principle) stipule quant à lui que chaque sous-partie atomique d'un logiciel (exemple : une classe) doit avoir une unique responsabilité (une raison de changer) ou bien être elle-même décomposée en sous-parties. Exemples d'applications de ces deux principes :
  - Une sous-partie qui s'occupe des affichages à l'écran ne devrait pas comporter de traitements métier, ni de code en rapport avec l'accès aux données.
  - Un composant de traitements métier (calcul scientifique ou financier, etc.) ne doit pas s'intéresser ni à l'affichage des données qu'il manipule, ni à leur stockage.
  - Une classe d'accès à une base de données (connexion, exécution de requêtes) ne devrait faire ni traitements métier, ni affichage des informations.
  - Une classe qui aurait deux raisons de changer devrait être scindée en deux classes distinctes.

Conception logicielle

Réutilisation

- Des logiciels de plus en plus complexes doivent être réalisés dans des délais de plus en plus courts, tout en maintenant le meilleur niveau de qualité possible. Une réponse à ces exigences contradictoires passe par la réutilisation de briques logicielles de base appelées librairies, modules ou plus généralement composants. En particulier, la mise à disposition de milliers de projets open source via des plates-formes comme « GitHub » ou des outils comme « NuGet, Composer ou npm » ont permis aux équipes de développement de faire des gains de productivité remarquables en intégrant ces composants lors de la conception de leurs applications.
- A l'heure actuelle, il n'est pas de logiciel de taille significative qui n'intègre plusieurs dizaines, voire des centaines de composants externes. Déjà testé et éprouvé, un composant logiciel fait simultanément baisser le temps et augmenter la qualité du développement. Il permet de limiter les efforts nécessaires pour traiter les problématiques techniques afin de se concentrer sur les problématiques métier, celles qui sont en lien direct avec ses fonctionnalités essentielles. Voici parmi bien d'autres quelques exemples de problématiques techniques adressables par des composants logiciels :
    - Accès à une base de données (connexion, exécution de requêtes).
    - Calculs scientifiques.
    - Gestion de l'affichage (moteur 3D).
    - Journalisation des évènements dans des fichiers.

Encapsulation maximale

- Ce principe de conception recommande de n'exposer au reste de l'application que le strict nécessaire pour que la sous-partie joue son rôle. Au niveau d'une classe, cela consiste à ne donner le niveau d'accessibilité publique qu'à un nombre minimal de membres, qui seront le plus souvent des méthodes. Au niveau d'une sous-partie d'application composée de plusieurs classes, cela consiste à rendre certaines classes privées afin d'interdire leur utilisation par le reste de l'application.

Conception logicielle

Couplage faible

- La définition du couplage est la suivante : « une entité (sous-partie, composant, classe, méthode) est couplée à une autre si elle dépend d'elle », autrement dit, si elle a besoin d'elle pour fonctionner. Plus une classe ou une méthode utilise d'autres classes comme classes de base, attributs, paramètres ou variables locales, plus son couplage avec ces classes augmente. Au sein d'une application, un couplage fort tisse entre ses éléments des liens puissants qui la rend plus rigide à toute modification (on parle de « code spaghetti »).
- A l'inverse, un couplage faible permet une grande souplesse de mise à jour. Un élément peut être modifié (exemple : changement de la signature d'une méthode publique) en limitant ses impacts. Le couplage peut également désigner une dépendance envers une technologie ou un élément extérieur spécifique : un SGBD, un composant logiciel, etc. Le principe de responsabilité unique permet de limiter le couplage au sein de l'application : chaque sous-partie a un rôle précis et n'a que des interactions limitées avec les autres sous parties. Pour aller plus loin, il faut limiter le nombre de paramètres des méthodes et utiliser des classes abstraites ou des interfaces plutôt que des implémentations spécifiques.

Cohésion forte

- Ce principe recommande de placer ensemble des éléments (composants, classes, méthodes) ayant des rôles similaires ou dédiés à une même problématique. Inversement, il déconseille de rassembler des éléments ayant des rôles différents. Exemple : ajouter une méthode de calcul métier au sein d'un composant lié aux données (ou à la présentation) est contraire au principe de cohésion forte.

Conception logicielle

DRY (Don’t Repeat Yourself)

- Ce principe vise à éviter la redondance au travers de l'ensemble de l'application. Cette redondance est en effet l'un des principaux ennemis du développeur. Elle a les conséquences néfastes suivantes :
  - Augmentation du volume de code ;
  - Diminution de sa lisibilité ;
  - Risque d'apparition de bugs dû à des modifications incomplètes.
- La redondance peut se présenter à plusieurs endroits d'une application, parfois de manière inévitable (réutilisation d'un existant). Elle prend souvent la forme d'un ensemble de lignes de code dupliquées à plusieurs endroits pour répondre au même besoin, comme dans l'exemple suivant.
- Le principe DRY est important mais ne doit pas être appliqué de manière trop scrupuleuse. Vouloir absolument éliminer toute forme de redondance conduit parfois à créer des applications inutilement génériques et complexes. C'est l'objet des deux prochains principes.

```
function A() {
// ...
// Code dupliqué
// ...
}
functionB() {
// ...
// Code dupliqué
// ...
}
```

```
function C() {
// Code auparavant dupliqué
}
function A() {
// ...
// Appel à C()
// ...
}
function B() {
// ...
// Appel à C()
// ...
    }
```

Conception logicielle

KISS (Keep It Simple, Stupid)

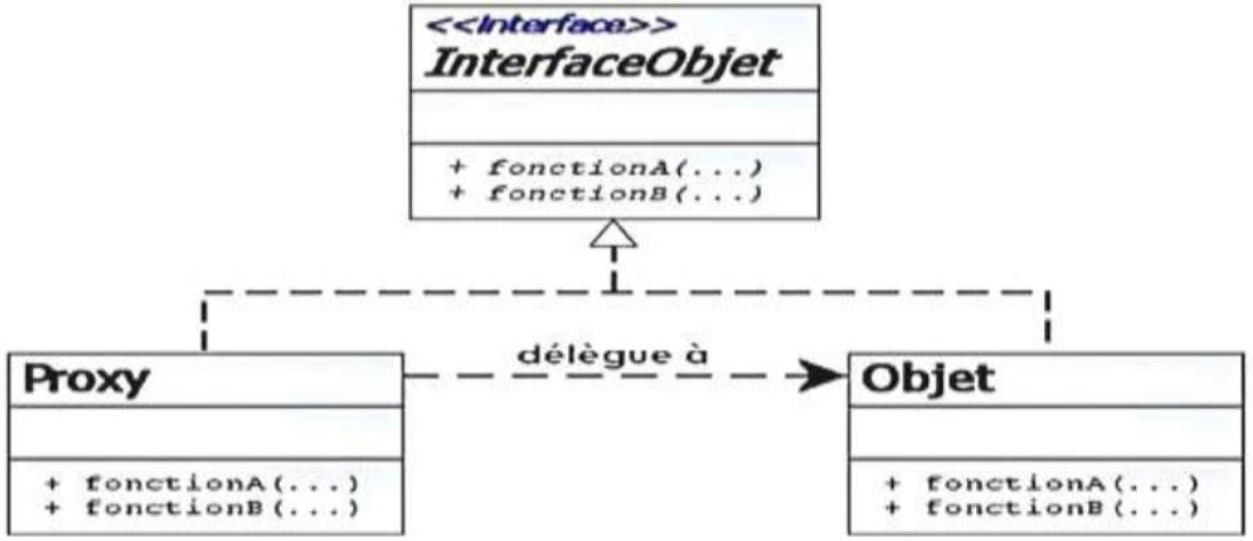
- KISS est un autre acronyme signifiant *Keep It Simple, Stupid* et qu'on peut traduire par « Ne complique pas les choses ». Ce principe vise à privilégier autant que possible la simplicité lors de la construction d'une application. Il part du constat que la complexité entraîne des surcoûts de développement puis de maintenance, pour des gains parfois discutables. La complexité peut prendre la forme d'une architecture surdimensionnée par rapports aux besoins (over engineering), ou de l'ajout de fonctionnalités secondaires ou non prioritaires. Une autre manière d'exprimer ce principe consiste à affirmer qu'une application doit être créée selon l'ordre de priorité ci-dessous :
  - *Make it work.*
  - *Make it right.*
  - *Make it fast.*

YAGNI (You Ain’t Gonne Need It)

- Ce troisième acronyme signifie « You Ain't Gonna Need It ». Corollaire du précédent, il consiste à ne pas se baser sur d'hypothétiques évolutions futures pour faire les choix du présent, au risque d'une complexification inutile (principe KISS). Il faut réaliser l'application au plus simple et en fonction des besoins actuels. Le moment venu, il sera toujours temps de procéder à des changements (refactorisation ou refactoring) pour que l'application réponde aux nouvelles exigences

Patrons logiciels

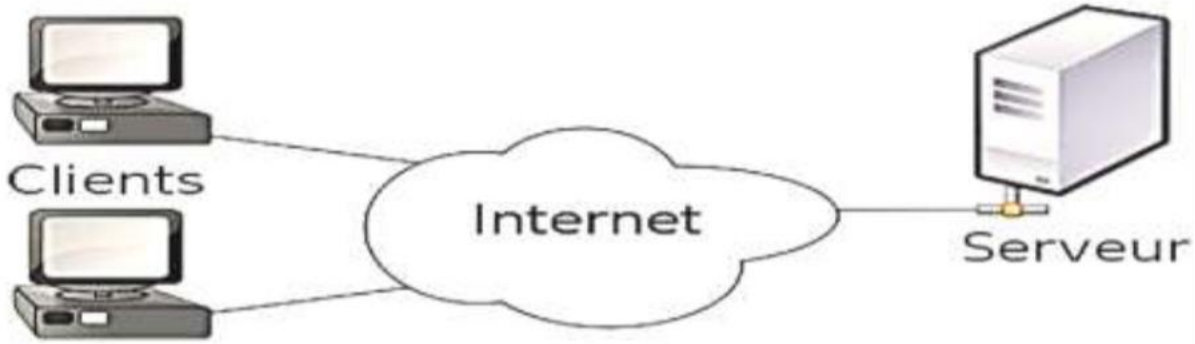
- Un patron de conception (design pattern) aussi appelé « Motif de Conception » est une solution standard à un problème de conception. L'ensemble des patrons de conception constitue un catalogue de bonnes pratiques issu de l'expérience de la communauté. Leurs noms forment un vocabulaire commun qui permet d'identifier immédiatement la solution associée.
- Les patrons de conception ont été popularisés par le livre Design Patterns – Elements of Reusable Object-Oriented Software sorti en 1995 et coécrit par quatre auteurs. Ce livre décrit 23 patrons, auxquels d'autres se sont rajoutés. Chaque patron décrit un problème à résoudre puis les éléments de sa solution, ainsi que leurs relations. Le formalisme graphique utilisé est souvent un diagramme de classes UML. L'exemple ci-dessous décrit le patron de conception Proxy, dont l'objectif est de substituer un objet à un autre afin de contrôler l'utilisation de ce dernier



Patrons logiciels

Architecture client/serveur

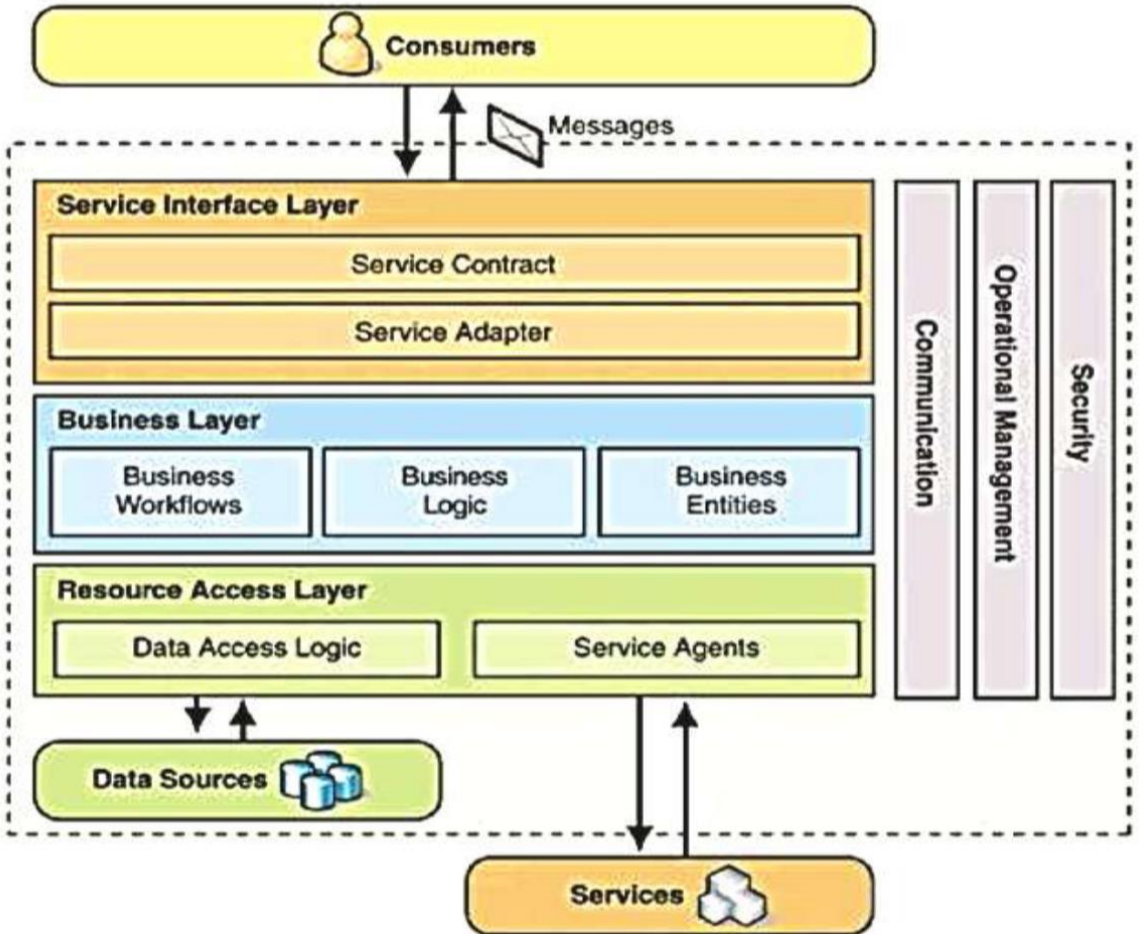
- L'architecture client/serveur caractérise un système basé sur des échanges réseau entre des clients et un serveur centralisé, lieu de stockage des données de l'application.
- Le principal avantage de l'architecture client/serveur tient à la centralisation des données. Stockées à un seul endroit, elles sont plus faciles à sauvegarder et à sécuriser. Le serveur qui les héberge peut être dimensionné pour pouvoir héberger le volume de données nécessaire et répondre aux sollicitations de nombreux clients. Cette médaille a son revers : le serveur constitue le nœud central du système et représente son maillon faible. En cas de défaillance (surcharge, indisponibilité, problème réseau), les clients ne peuvent plus fonctionner. On peut classer les clients d'une architecture client/serveur en plusieurs types :
  - Client léger : destiné uniquement à l'affichage (exemple : navigateur web) ;
  - Client lourd : application native spécialement conçue pour communiquer avec le serveur (exemple : application mobile) ;
  - Client riche : combinant les avantages des clients légers et lourds (exemple : navigateur web utilisant des technologies évoluées pour offrir une expérience utilisateur proche de celle d'une application native) ;
- Le fonctionnement en mode client/serveur est très souvent utilisé en informatique. Un réseau Windows organisé en domaine, la consultation d'une page hébergée par un serveur Web ou le téléchargement d'une application mobile depuis un magasin central (App Store, Google Play) en constituent des exemples.



Patrons logiciels

Architecture en couches

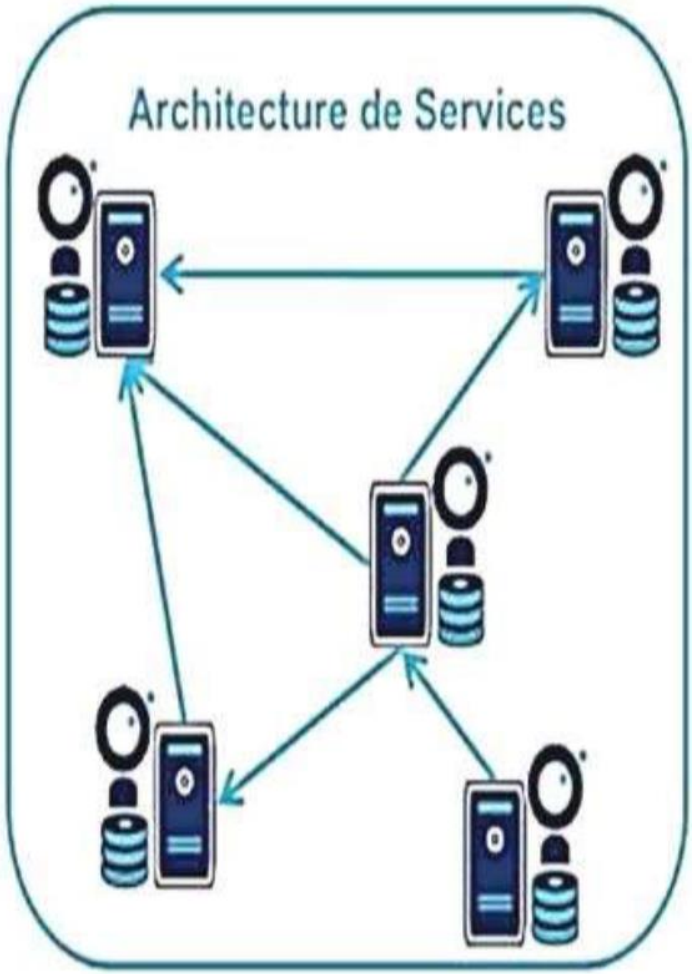
- Une architecture en couches organise un logiciel sous forme de couches (layers). Chaque couche ne peut communiquer qu'avec les couches adjacentes.
- Cette architecture respecte le principe de séparation des responsabilités et facilite la compréhension des échanges au sein de l'application. Lorsque chaque couche correspond à un processus distinct sur une machine, on parle d'architecture « n-tiers », n désignant le nombre de couches. Un navigateur web accédant à des pages dynamiques intégrant des informations stockées dans une base de données constitue un exemple classique d'architecture 3-tiers.



Patrons logiciels

Architecture orientée service

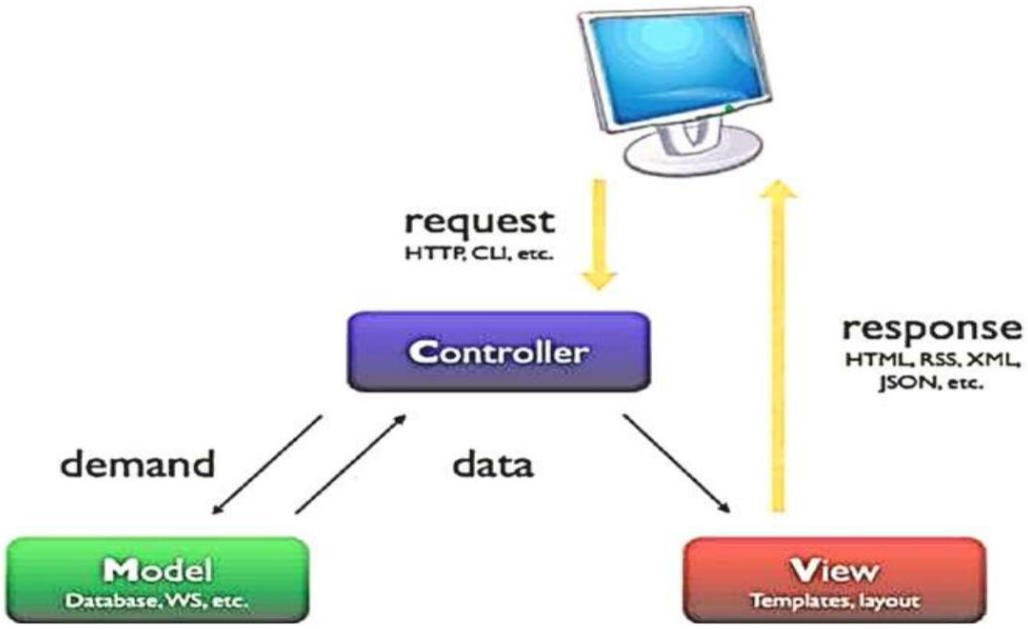
- Une architecture orientée services (SOA : Service-Oriented Architecture) décompose un logiciel sous la forme d'un ensemble de services métier utilisant un format d'échange commun, généralement XML ou JSON.
- Une variante récente, l'architecture micro-services, diminue la granularité des services pour leur assurer souplesse et capacité à évoluer, au prix d'une plus grande distribution du système.



Patrons logiciels

Architecture modèle-vue-contrôleur

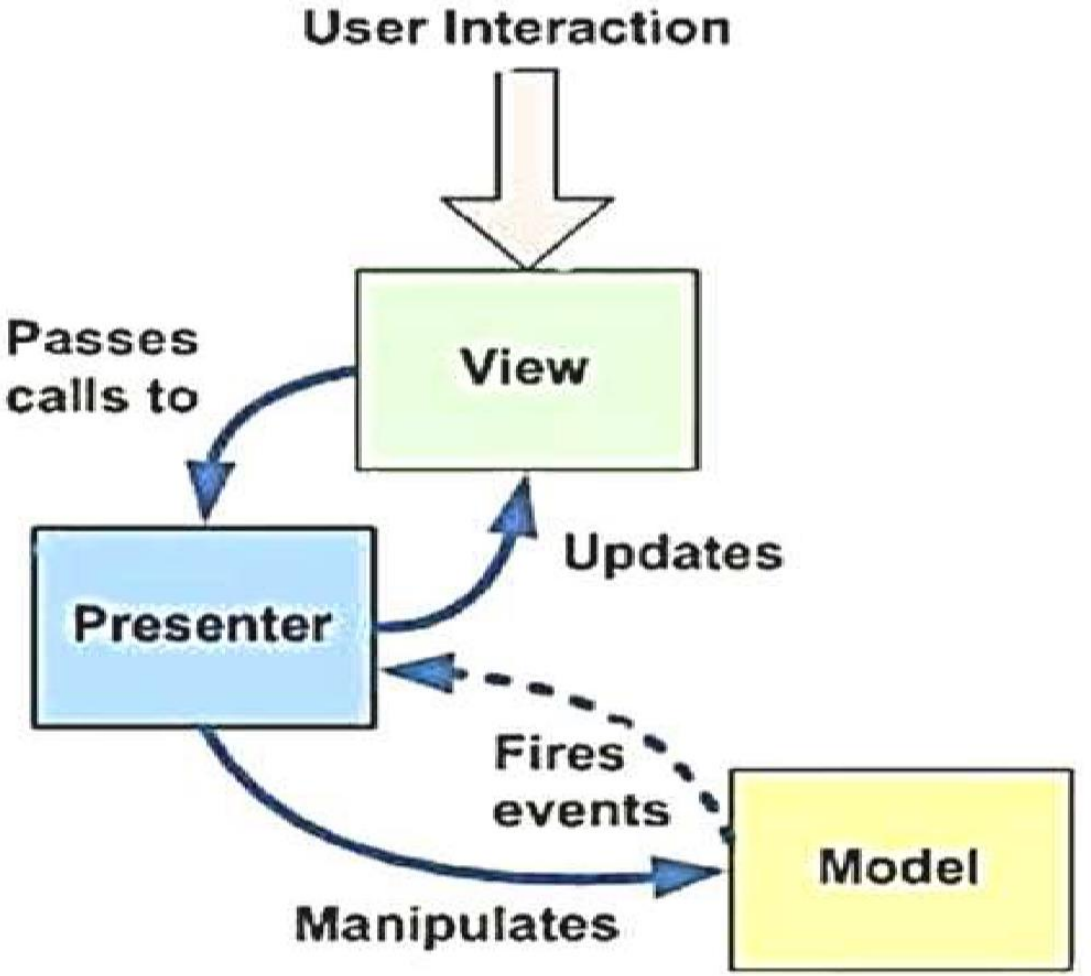
- Le patron Modèle-Vue-Contrôleur, ou MVC, décompose une application en trois sous parties :
  - La partie Modèle qui regroupe la logique métier ainsi que l'accès aux données. Il peut s'agir d'un ensemble de fonctions (Modèle procédural) ou de classes (Modèle orienté objet) ;
  - La partie Vue qui s'occupe des interactions avec l'utilisateur : présentation, saisie et validation des données ;
  - La partie Contrôleur qui gère la dynamique de l'application. Elle fait le lien entre les deux autres parties.
- Ce patron a été imaginé à la fin des années 1970 pour le langage Small-talk afin de bien séparer le code de l'interface graphique de la logique applicative. On le retrouve dans de très nombreux langages : bibliothèques Swing et Model 2 (JSP) de Java, frameworks PHP, ASP.NET MVC …



Patrons logiciels

Architecture modèle-vue-présentation

- Le patron Modèle-Vue-Présentation, ou MVP, est un proche cousin du patron MVC surtout utilisé pour construire des interfaces utilisateurs (UI). Dans une architecture MVP, la partie Vue reçoit les évènements provenant de l'utilisateur et délègue leur gestion à la partie Présentation. Celle-ci utilise les services de la partie Modèle puis met à jour la Vue.
- Dans la variante dite « Passive View » de cette architecture, la Vue est passive et dépend totalement du contrôleur pour ses mises à jour. Dans la variante dite « Supervising Controller », Vue et Modèle sont couplées et les modifications du Modèle déclenchent la mise à jour de la Vue.



Production de code

- L'objectif de cette partie du cours est de présenter les enjeux et les solutions liés à la production du code source d'un logiciel. Le code source est le cœur d'un projet logiciel. Il est essentiel que tous les membres de l'équipe de développement se coordonnent pour adopter des règles communes dans la production de ce code. L'objectif de ces règles est l'uniformisation de la base de code source du projet. Les avantages liés sont les suivants :
  - La consultation du code est facilitée.
  - Les risques de duplication ou d'erreurs liées à des pratiques disparates sont éliminés.
  - Chaque membre de l'équipe peut comprendre et intervenir sur d'autres parties que celles qu'il a lui-même réalisées.
  - Les nouveaux venus sur le projet mettront moins longtemps à être opérationnels.

Convention de nommage

- Une première série de règle concerne le nommage des différents éléments qui composent le code. Il n'existe pas de standard universel à ce sujet. La convention la plus fréquemment adoptée se nomme « camelCase ».
- Elle repose sur trois grands principes :
  - Les noms des classes (et des méthodes en C#, pour être en harmonie avec le framework .NET) commencent par une lettre majuscule.
  - Les noms de tous les autres éléments (variables, attributs, paramètres, etc.)commencent par une lettre minuscule.
  - Si le nom d'un élément se compose de plusieurs mots, la première lettre de chaque mot suivant le premier s'écrit en majuscule
- On peut ajouter à cette convention une règle qui impose d'utiliser le pluriel pour nommer les éléments contenant plusieurs valeurs, comme les tableaux et les listes

Production de code

Langue utilisée

- La langue utilisée dans la production du code doit être unique sur tout le projet. Le français et l'anglais ont chacun leurs avantages et leurs inconvénients. On choisira de préférence l'anglais pour les projets de taille importante ou destinés à être publiés en ligne.

Formatage du code

- La grande majorité des IDE et des éditeurs de code offrent des fonctionnalités de formatage automatique du code. A condition d'utiliser un paramétrage commun, cela permet à chaque membre de l'équipe de formater rapidement et uniformément le code sur lequel il travaille. Les paramètres de formatage les plus courants sont :
  - Taille des tabulations (2 ou 4 espaces) ;
  - Remplacement automatique des tabulations par des espaces ;
  - Passage ou non à la ligne après chaque accolade ouvrante ou fermante ;
  - Ajout ou non d'un espace avant une liste de paramètres

Commentaires

- L'ajout de commentaires permet de faciliter la lecture et la compréhension d'une portion de code source. L'ensemble des commentaires constitue une forme efficace de documentation d'un projet logiciel. Il n'y a pas de règle absolue, ni de consensus, en matière de taux de commentaires dans le code source. Certaines méthodologies de développement agile (eXtremeProgramming) vont jusqu'à affirmer qu'un code bien écrit se suffit à lui-même et ne nécessite aucun ajout de commentaires.
- Dans un premier temps, il vaut mieux se montrer raisonnable et commenter les portions de code complexes ou essentielles : en-têtes de classes, algorithmes importants, portions atypiques, etc. Il faut éviter de paraphraser le code source en le commentant, ce qui alourdit sa lecture et n'est d'aucun intérêt.

Gestion des versions

- Seule une petite minorité de logiciels sont conçus par un seul développeur. La grande majorité des projets sont réalisés et/ou maintenus par une équipe de plusieurs personnes travaillant sur la même base de code source. Ce travail en parallèle est source de complexité :
  - Comment récupérer le travail d'un autre membre de l'équipe ?
  - Comment publier ses propres modifications ?
  - Comment faire en cas de modifications conflictuelles (travail sur le même fichier source qu'un ou plusieurs collègues) ?
  - Comment accéder à une version précédente d'un fichier ou du logiciel entier?
- Pour les raisons précédentes, tout projet logiciel d'entreprise (même mono-développeur) doit faire l'objet d'une gestion des versions (Revision Control System ou versioning). La gestion des versions vise les objectifs suivants :
  - Assurer la pérennité du code source d'un logiciel ;
  - Permettre le travail collaboratif ;
  - Fournir une gestion de l'historique du logiciel.
- La gestion des versions la plus basique consiste à déposer le code source sur un répertoire partagé par l'équipe de développement. Cette gestion ne répond pas à tous les objectifs précédents, il existe une catégorie de logiciels spécialisés dans la gestion des versions (SVN, Git, Mercurial) qui sont un standard du développement.

Gestion des versions

- Un logiciel de gestion des versions est avant tout un dépôt de code qui héberge le code source du projet. Chaque développeur peut accéder au dépôt afin de récupérer le code source, puis de publier ses modifications. Les autres développeurs peuvent alors récupérer le travail publié.
- Le logiciel garde la trace des modifications successives d'un fichier. Il permet d'en visualiser l'historique et de revenir à une version antérieure. Un logiciel de gestion des versions permet de travailler en parallèle sur plusieurs problématiques (par exemple, la correction des bugs de la version publiée et l'avancement sur la future version) en créant des **branches**. Les modifications réalisées sur une branche peuvent ensuite être intégrées (**merge**) à une autre.
- En cas d'apparition d'un **conflit** (modifications simultanées du même fichier par plusieurs développeurs), le logiciel de gestion des versions permet de comparer les versions du fichier et de choisir les modifications à conserver ou à rejeter pour créer le fichier fusionné final. Le logiciel de gestion des versions permet de regrouper logiquement des fichiers par le biais du tagging: il ajoute aux fichiers source des tags correspondant aux différentes versions du logiciel.
- La très grande majorité des projets logiciels sont menés par des équipes de plusieurs développeurs. Il est de plus en plus fréquent que ces développeurs travaillent à distance ou en mobilité. Le travail en équipe sur un projet logiciel nécessite de pouvoir :
  - Partager le code source entre membres de l'équipe ;
  - Gérer les droits d'accès au code ;
  - Intégrer les modifications réalisées par chaque développeur.
  - Signaler des problèmes ou proposer des améliorations qui peuvent ensuite être discutés collectivement.
- Pour répondre à ces besoins, des plates-formes de publication et de partage de code enligne sont apparues. On les appelle parfois des forges logicielles. La plus importante à l'heure actuelle est la « plate-forme GitHub ».

Tests du logiciel

- La problématique des tests est souvent considérée comme secondaire et négligée par les développeurs. C'est une erreur : lorsqu'on livre une application et qu'elle est placée en production (offerte à ses utilisateurs), il est essentiel d'avoir un maximum de garanties sur son bon fonctionnement afin d'éviter au maximum de coûteuses mauvaises surprises.
- Le test d'une application peut être manuel. Dans ce cas, une personne effectue sur l'application une suite d'opérations prévue à l'avance (navigation, connexion, envoi d'informations...) pour vérifier qu'elle possède bien le comportement attendu. C'est un processus coûteux en temps et sujets aux erreurs (oublis, négligences, etc.). En complément de ces tests manuels, on a tout intérêt à intégrer à un projet logiciel des tests automatisés qui pourront être lancés aussi souvent que nécessaire. Ceci est d'autant plus vrai pour les méthodologies agiles basées sur un développement itératif et des livraisons fréquentes, ou bien lorsque l'on met en place une **intégration continue**. On peut classer les tests logiciels en différentes catégories :
  - **Tests de validation** : Ces tests sont réalisés lors de la recette (validation) par un client d'un projet livré par l'un de ses fournisseurs. Souvent écrits par le client lui-même, ils portent sur l'ensemble du logiciel et permet de vérifier son comportement global en situation. De par leur spectre large et leur complexité, les tests de validation sont le plus souvent manuels. Les procédures à suivre sont regroupées dans un document associé au projet, fréquemment nommé plan de validation.
  - **Tests d'intégration** : Dans un projet informatique, l'intégration est de fait d'assembler plusieurs composants (ou modules) élémentaires en un composants de plus haut niveau. Un test d'intégration valide les résultats des interactions entre plusieurs composants permet de vérifier que leur assemblage s'est produit sans défaut. Il peut être manuel ou automatisé. Un nombre croissant de projets logiciels mettent en place un processus d'intégration continue. Cela consiste à vérifier que chaque modification ne produit pas de régression dans l'application développée. L'intégration continue est nécessairement liée à une batterie de tests qui se déclenchent automatiquement lorsque des modifications sont intégrées au code du projet.
  - Tests unitaires : Contrairement aux tests de validation et d'intégration qui testent des pans entiers d'un logiciel, un test unitaire na valide qu'une portion atomique du code source (exemple : une seule classe) et est systématiquement automatisé. Le test unitaire offre les avantages suivants :
    - Il est facile à écrire. Dédié à une partie très réduite du code, le test unitaire ne nécessite le plus souvent qu'un contexte minimal, voire pas de contexte du tout ;
    - Il offre une granularité de test très fine et permet de valider exhaustivement le comportement de la partie du code testée (cas dégradés, saisie d'informations erronées...) ;
    - Son exécution est rapide, ce qui permet de le lancer très fréquemment (idéalement à chaque modification du code testé) ;
    - Il rassemble les cas d'utilisation possibles d'une portion d'un projet et représente donc une véritable documentation sur la manière de manipuler le code testé ;

Tests du logiciel

- L'ensemble des tests unitaires d'un projet permet de valider unitairement une grande partie de son code source et de détecter le plus tôt possibles d'éventuelles erreurs. En pratique, très peu de parties d'un projet fonctionnent de manière autonome, ce qui complique l'écriture des tests unitaires. Par exemple, comment tester unitairement une classe qui collabore avec plusieurs autres pour réaliser ses fonctionnalités ? La solution consiste à créer des éléments qui simulent le comportement des collaborateurs d'une classe donnée, afin de pouvoir tester le comportement de cette classe dans un environnement isolé et maîtrisé. Ces éléments sont appelés des « tests doubles ». Selon la complexité du test à écrire, un test double peut être :
  - Un dummy : élément basique sans aucun comportement, juste là pour faire compiler le code lors du test ;
  - Un stub : qui renvoie des données permettant de prévoir les résultats attendus lors du test ;
  - Un mock : qui permet de vérifier finement le comportement de l'élément testé (ordre d'appel des méthodes, paramètres passés, etc.).

Documentation logicielle

- Il est pénible, voire parfois très difficile, de se familiariser avec un logiciel par la seule lecture de son code source. En complément, un ou plusieurs documents doivent accompagner le logiciel. On peut classer cette documentation en deux catégories : La documentation technique et La documentation utilisateur.

Documentation technique

- Le mot-clé de la documentation technique est « comment ». Il ne s'agit pas ici de dire pourquoi le logiciel existe, ni de décrire ses fonctionnalités attendues. Ces informations figurent dans d'autres documents comme le cahier des charges. Il ne s'agit pas non plus d'expliquer à un utilisateur du logiciel ce qu'il doit faire pour effectuer telle ou telle tâche : c'est le rôle de la documentation utilisateur. La documentation technique doit expliquer comment fonctionne le logiciel.
- La documentation technique est écrite par des informaticiens, pour des informaticiens. Elle nécessite des compétences techniques pour être comprise. Le public visé est celui des personnes qui interviennent sur le logiciel du point de vue technique : « développeurs, intégrateurs, responsables techniques, éventuellement chefs de projet ». Dans le cadre d'une relation maîtrise d'ouvrage ou maîtrise d'œuvre pour réaliser un logiciel, la responsabilité de la documentation technique est à la charge de la maîtrise d'œuvre. Le contenu de la documentation technique varie fortement en fonction de la structure et de la complexité du logiciel associé. Elle contient le plus souvent :
  - La modélisation : elle précise comment les éléments - métiers ont été modélisés informatiquement au sein du logiciel. Si le logiciel a fait l'objet d'une modélisation orientée objet, la documentation technique inclut une représentation des principales classes (souvent les classes métier) sous la forme d'un diagramme de classes respectant la norme UML. Si le logiciel utilise une base de données, la documentation technique doit présenter le modèle d'organisation des données retenu, le plus souvent sous la forme d'un modèle logique sous forme graphique.
  - L'Architecture : La phase d'architecture d'un logiciel permet, en partant des besoins exprimés dans le cahier des charges, de réaliser les grands choix qui structureront le développement : technologies, langages, patrons utilisés, découpage en sous-parties, outils, etc. La documentation technique doit décrire tous ces choix de conception. L'implantation physique des différents composants (appelés parfois tiers) sur une ou plusieurs machines doit également être documentée.
  - La Production du code source : Les normes et standards utilisés pendant le développement (conventions de nommage, formatage du code, etc.) doivent être présentés dans la documentation technique.
  - La Génération : Le processus de génération (« build ») permet de passer des fichiers sources du logiciel aux éléments exécutables. Si elle n'est pas automatique, elle doit être documentée.
  - Le Déploiement : La documentation technique doit indiquer comment s'effectue le déploiement du logiciel, c'est-à-dire l'installation de ses différents composants sur la ou les machines nécessaires.
  - La Documentation du code source : Il est également possible de documenter un logiciel directement depuis son code source en y ajoutant des commentaires. Certains langages disposent d'un format spécial de commentaire permettant de créer une documentation auto-générée (par exemple Doxygen ou javadoc).

Documentation logicielle

Documentation utilisateur

- Contrairement à la documentation technique, la documentation d'utilisation ne vise pas à faire comprendre comment le logiciel est conçu. Son objectif est d'apprendre à l'utilisateur à se servir du logiciel. La documentation d'utilisation doit être :
  - Utile : une information exacte, mais inutile, ne fait que renforcer le sentiment d'inutilité et gêne la recherche de l'information pertinente ;
  - Agréable : sa forme doit favoriser la clarté et mettre en avant les préoccupations de l'utilisateur et non pas les caractéristiques techniques du produit.
- Le public visé est l'ensemble des utilisateurs du logiciel. Selon le contexte d'utilisation, les utilisateurs du logiciel à documenter peuvent avoir des connaissances en informatique (exemples : cas d'un IDE ou d'un outil de SCM). Cependant, on supposera le plus souvent que le public visé n'est pas un public d'informaticiens. La Conséquence essentielle est que toute information trop technique est à bannir de la documentation d'utilisation. Pas question d'aborder, l'architecture MVC ou les design patterns employés : ces éléments ont leur place dans la documentation technique. D'une manière générale, s'adapter aux connaissances du public visé constitue la principale difficulté de la rédaction de la documentation d'utilisation.

Documentation logicielle

Documentation utilisateur

- Le Manuel utilisateur est la forme la plus classique de la documentation d'utilisation consiste à rédiger un manuel utilisateur, le plus souvent sous la forme d'un document bureautique. Ce document est structuré et permet aux utilisateurs de retrouver les informations qu'ils recherchent. Il intègre très souvent des captures d'écran afin d'illustrer le propos. Un manuel utilisateur peut être organisé de deux façons :
  - Le Guide d'utilisation : ce mode d'organisation décompose la documentation en grandes fonctionnalités décrites pas à pas et dans l'ordre de leur utilisation. Cette organisation plaît souvent aux utilisateurs car elle leur permet d'accéder facilement aux informations essentielles. En revanche, s'informer sur une fonctionnalité avancée ou un détail complexe peut s'avérer difficile.
  - Manuel de référence : dans ce mode d'organisation, on décrit une par une chaque fonctionnalité du logiciel, sans se préoccuper de leur ordre ou de leur fréquence d'utilisation. Cette organisation suit la logique du créateur du logiciel plutôt que celle de son utilisateur. Elle est en général moins appréciée de ces derniers.
- Le Tutoriel : De plus en plus souvent, la documentation d'utilisation inclut un ou plusieurs tutoriels, destinés à faciliter la prise en main initiale du logiciel. Un tutoriel est un guide pédagogique constitué d'instructions détaillées pas à pas en vue d'objectifs simples. Le tutoriel a l'avantage de "prendre l'utilisateur par la main" afin de l'aider à réaliser ses premiers pas avec le logiciel qu'il découvre, sans l'obliger à parcourir un manuel utilisateur plus ou moins volumineux. Il peut prendre la forme d'un document texte, ou bien d'une vidéo ou d'un exercice interactif. Cependant, il est illusoire de vouloir documenter l'intégralité d'un logiciel en accumulant les tutoriels.
- La FAQ : Une Foire Aux Questions (en anglais Frequently Asked questions) est une liste de questions-réponses sur un sujet. Elle peut faire partie de la documentation d'utilisation d'un logiciel. La création d'une FAQ permet d'éviter que les mêmes questions soient régulièrement posées.
- L’Aide en ligne : L'aide en ligne est une forme de documentation d'utilisation accessible depuis un ordinateur. Il peut s'agir d'une partie de la documentation publiée sur Internet sous un format hypertexte. Quand une section de l'aide en ligne est accessible facilement depuis la fonctionnalité d'un logiciel qu'elle concerne, elle est appelée aide contextuelle ou aide en ligne contextuelle.

TD : système de gestion des stocks

- Commande : Un logiciel de gestion des stocks pour un épicier
- 1) Analyser le besoin
- 2) Définir l’architecture
- 3) Définir les classes
- 4) Etablir un plan de test
- 5) Quelle documentation établir ?
- 6) Comment organiser le travail ?

Méthodes de développement logiciel

- Les méthodes d'analyse et de conception fournissent des notations standards et des conseils pratiques qui permettent d'aboutir à des conceptions « raisonnables », mais nous ferons toujours appel à la créativité du concepteur. Il existe différentes manières pour classer ces méthodes, dont :
  - La distinction compositionnelle / décompositionnelle : met en opposition d'une part les méthodes ascendantes qui consistent à construire un logiciel par composition à partir de modules existants et, d'autre part, les méthodes descendantes qui décomposent récursivement le système jusqu'à arriver à des modules programmables « simplement ».
  - La distinction fonctionnel / orientée objet : Dans la stratégie fonctionnelle un système est vu comme un ensemble d'unités en interaction, ayant chacune une fonction clairement définie. Les fonctions disposent d'un état local, mais le système a un état partagé, qui est centralisé et accessible par l'ensemble des fonctions. Les stratégies orientées objet considèrent qu'un système est un ensemble d'objets interagissant. Chaque objet dispose d'un ensemble d'attributs décrivant son état et l'état du système est décrit (de façon décentralisé) par l'état de l'ensemble. La décomposition fonctionnelle du haut vers le bas a été largement utilisée aussi bien dans de petits projets que dans de très grands, et dans divers domaines d'application. La méthode orientée objet a eu un développement plus récent. Elle encourage la production de systèmes divisés en composants indépendants, en interaction.

- Dans le cadre de cours, nous distinguerons alors quatre types des méthodes à savoir :
  - les méthodes fonctionnelles, basées sur les fonctionnalités du logiciel ;
  - les méthodes objet, basées sur différents modèles (statiques, dynamiques et fonctionnels) de développement logiciel.
  - les méthodes adaptatives ou Agiles, basées sur le changement des besoins ;
  - les méthodes spécifiques, basées sur les découpages temporels particuliers.

Méthodes de développement logiciel

Les méthodes fonctionnelles

- Les méthodes fonctionnelles ont pour origine la programmation structurée. Cette approche consiste à décomposer une fonctionnalité (ou fonction) du logiciel en plusieurs sous fonctions plus simples. Il s'agit d'une conception « top-down », basée sur le principe « diviser pour mieux régner». L'architecture du système est le reflet de cette décomposition fonctionnelle. La programmation peut ensuite être réalisée soit à partir des fonctions de haut niveau (développement « top-down »), soit à partir des fonctions de bas niveau (développement « bottom-up »).
- Cette méthode présente comme les inconvénients suivants :
  - L'architecture étant basée sur la décomposition fonctionnelle, une évolution fonctionnelle peut remettre en cause l'architecture. Cette méthode supporte donc mal l'évolution des besoins.
  - Cette méthode ne favorise pas la réutilisation de composants, car les composants de bas niveau sont souvent ad hoc et donc peu réutilisables.

Méthodes de développement logiciel

Les méthodes objet

- Les approches objet sont basées sur une modélisation du domaine d'application. Les « objets »sont une abstraction des entités du monde réel. De façon générale, la modélisation permet de réduire la complexité et de communiquer avec les utilisateurs. Plus précisément un modèle :
  - Aide à visualiser un système tel qu'il est ou tel qu'on le souhaite ;
  - Permet de spécifier la structure ou le comportement d'un système ;
  - Fournit un guide pour la construction du système ;
  - Documente les décisions prises lors de la construction du système.
- Ces modèles peuvent être comparés avec les plans d'un architecte : suivant la complexité du système on a besoin de plans plus ou moins précis. Pour construire une niche, on n'a pas besoin de plans, pour construire un chalet il faut un plan, pour construire un immeuble, on a besoin d'un ensemble de vues (plans au sol, perspectives, maquettes). Dans les méthodes objet, on distingue trois aspects :
  - Un aspect statique : Dans lequel, on identifie les objets, leurs propriétés et leurs relations ;
  - Un aspect dynamique : Dans lequel, on décrit les comportements des objets, en particuliers leurs états possibles et les évènements qui déclenchent les changements d'état ;
  - Un aspect fonctionnel : qui, à haut niveau, décrit les fonctionnalités du logiciel, ou, a plus bas niveau, décrit les fonctions réalisées par les objets par l'intermédiaire des méthodes.
- Les intérêts des approches objet sont les suivants :
  - Les approches objet sont souvent qualifiées de « naturelles » car elles sont basées sur le domaine d'application. Cela facilite en particulier la communication avec les utilisateurs.
  - Ces approches supportent mieux l'évolution des besoins que les approches fonctionnelles car la modélisation est plus stable, et les évolutions fonctionnelles ne remettent par l'architecture du système en cause.
  - Les approches objet facilitent la réutilisation des composants (qui sont moins spécifiques que lorsqu'on réalise une décomposition fonctionnelle).

Méthodes de développement logiciel

Les méthodes prédictives

- Ce sont des méthodes qui correspondent à un cycle de vie du logiciel en cascade ou en V, sont basées sur une planification très précise et très détaillée, qui a pour but de réduire les incertitudes liées au développement du logiciel. Cette planification rigoureuse ne permet pas d'évolutions dans les besoins des utilisateurs, qui doivent donc être figés dès l'étape de définition des besoins.

Les méthodes agiles

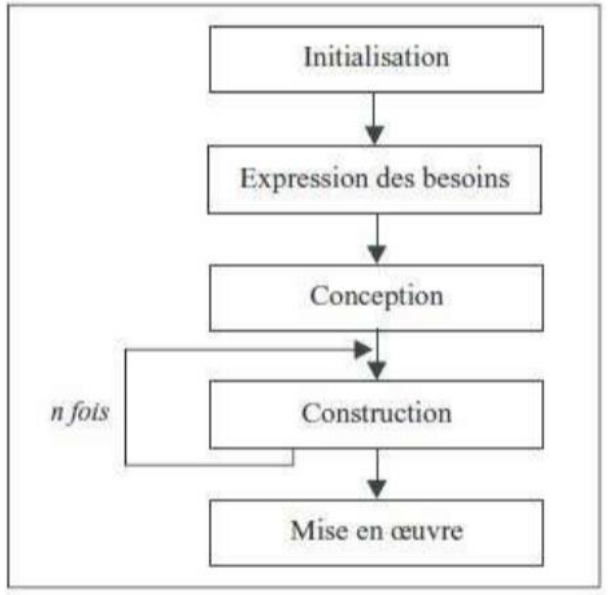
- Ce sont des méthodes qui correspondent à un cycle de vie itératif, qui considèrent que les changements (des besoins des utilisateurs, mais également de l'architecture, de la conception, de la technologie) sont inévitables et doivent être pris en compte par les modèles de développement. Ces méthodes privilégient la livraison de fonctionnalités utiles au client à la production de documentation intermédiaire sans intérêt pour le client.
- Ainsi, Toutes les méthodes agiles prennent en compte dans leur modèle de cycle de vie trois exigences :
  - Une forte participation entre développeurs et utilisateurs,
  - Des livraisons fréquentes de logiciel et ;
  - Une prise en compte de possibles changements dans les besoins des utilisateurs au cours du projet.

- C'est pourquoi toutes font appel, d'une façon ou d'une autre, à un modèle itératif et incrémental. De plus, elles préconisent en général des durées de cycle de vie des projets ne dépassant pas un an. Parmi les méthodes agiles, les plus usuelles ; on peut citer :
  - La méthode RAD (Rapid Application Development) ;
  - La méthode DSDM (Dynamic Systems Development Method);
  - La méthode XP (eXtreme Programming)
  - La méthode SCRUM.

Méthodes de développement logiciel

RAD (Rapide Application Development)

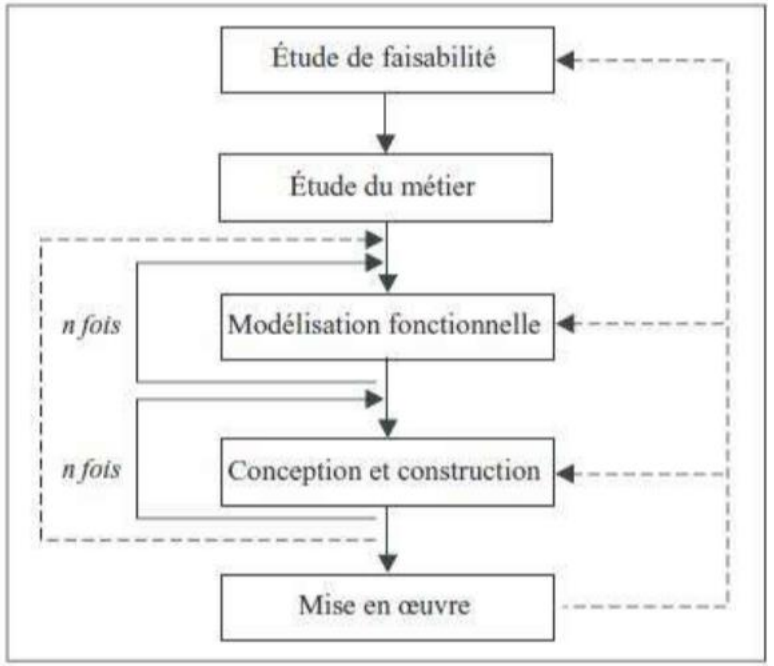
- La méthode RAD, est un modèle linéaire, structuré en cinq phases, et dont le modèle itératif intervient à la phase construction du logiciel en vu de la séquencer en plusieurs modules successivement livrés.
- La participation des utilisateurs est placée au cœur du cycle. En effet, le déroulement d'une phase comprend une ou plusieurs sous-phases, et chaque sous- phase présente une structure à trois temps, dans laquelle la tenue d'une session participative joue un rôle central. Des travaux préparatoires rassemblent et construisent le matériau (modèle ou prototype) qui sera ensuite discuté par les différents acteurs et ajusté.



Méthodes de développement logiciel

DSDM (Dynamic Systems Development Method)

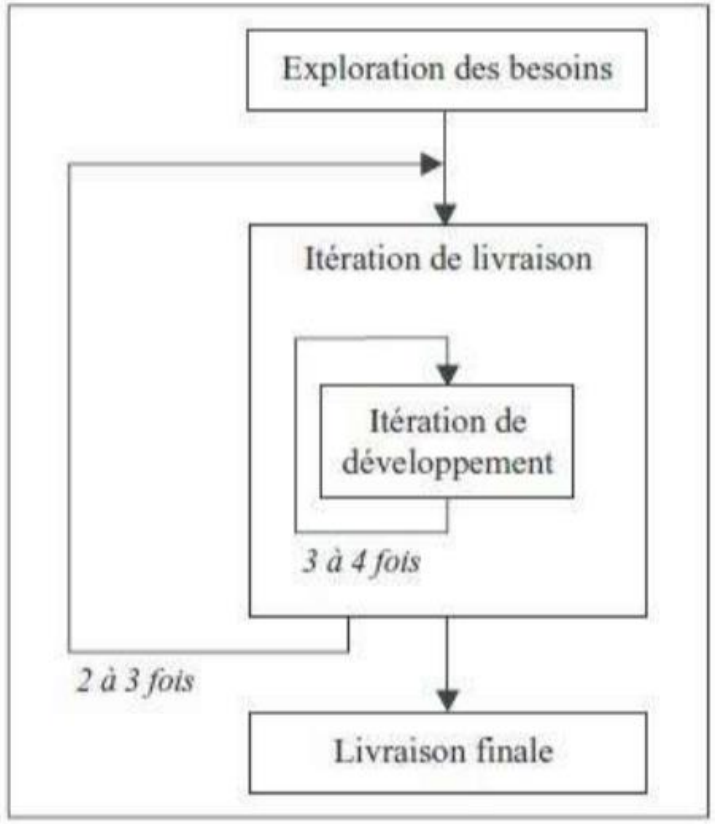
- La méthode DSDM a évolué au cours des années. L'actuelle version distingue le cycle de vie du système et le cycle de vie du projet. Le premier comprend, outre les phases du projet lui-même, une phase de pré-projet qui doit conduire au lancement du projet et une phase post-projet qui recouvre l'exploitation et la maintenance de l'application.
- Le cycle de vie du projet comprend cinq phases, dont deux sont cycliques. Les flèches pleines indiquent un déroulement normal. Les flèches en pointillé montrent des retours possibles à une phase antérieure, soit après la phase conception et construction, soit après celle de mise en œuvre.
- Après une étude de faisabilité, la phase étude du métier permet, à travers des ateliers (workshops) entre équipe de projet et manageurs, de définir le périmètre du projet, avec une liste d'exigences prioritaires et une architecture fonctionnelle et technique du futur système.
- La phase modélisation fonctionnelle est une suite de cycles. Chacun permet de définir précisément les fonctionnalités souhaitées et leur priorité. L'acceptation par toutes les parties prenantes d'un prototype fonctionnel, sur tout ou partie du périmètre, permet de passer à la phase conception et construction. L'objectif de cette phase est de développer un logiciel testé, par des cycles successifs de développement/acceptation par les utilisateurs.



Méthodes de développement logiciel

XP (eXtreme Programming)

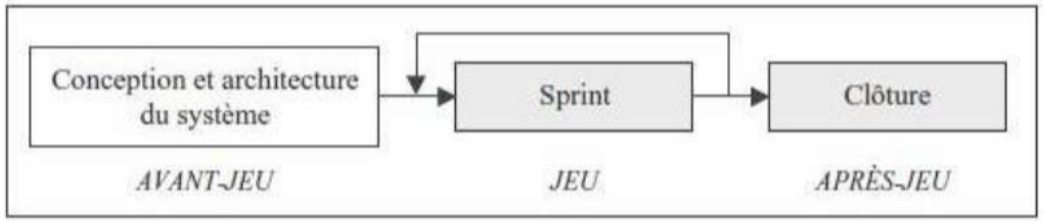
- La méthode XP, focalisée sur la partie programmation du projet, propose un modèle itératif avec une structure à deux niveaux : d'abord des itérations de livraison (release), puis des itérations de développement. Les premières conduisent à livrer des fonctionnalités complètes pour le client, les secondes portent sur des éléments plus fins appelés scénarios qui contribuent à la définition d'une fonctionnalité.
- Après une phase initiale d'Exploration des besoins, un plan de livraison est défini avec le client. Chaque livraison, d'une durée de quelques mois, se termine par la fourniture d'une version opérationnelle du logiciel. Une itération de livraison est découpée en plusieurs itérations de développement de courte durée (deux semaines à un mois), chacune donnant lieu à la livraison d'une ou plusieurs fonctionnalités pouvant être testées, voire intégrées dans une version en cours.
- De façon plus détaillée, chaque itération de développement commence par l'écriture de cas d'utilisation ou scénarios (user stories), c'est-à-dire des fonctions simples, concrètement décrites, avec les exigences associées, qui participent à la définition d'une fonctionnalité plus globale. Utilisateurs et développeurs déterminent ensemble ce qui doit être développé dans la prochaine itération. Une fonctionnalité est ainsi découpée en plusieurs tâches. Les plans de test sont écrits, les développeurs sont répartis en binôme ; ils codent les tâches qui leur sont affectées, puis effectuent avec les utilisateurs des tests d'acceptation. En cas d'échec, on revoit les scénarios et on reprend la boucle. Sinon, on continue jusqu'à avoir développé tous les scénarios retenus. Une version livrable est alors arrêtée et mise à disposition, ainsi que la documentation.



Méthodes de développement logiciel

SCRUM

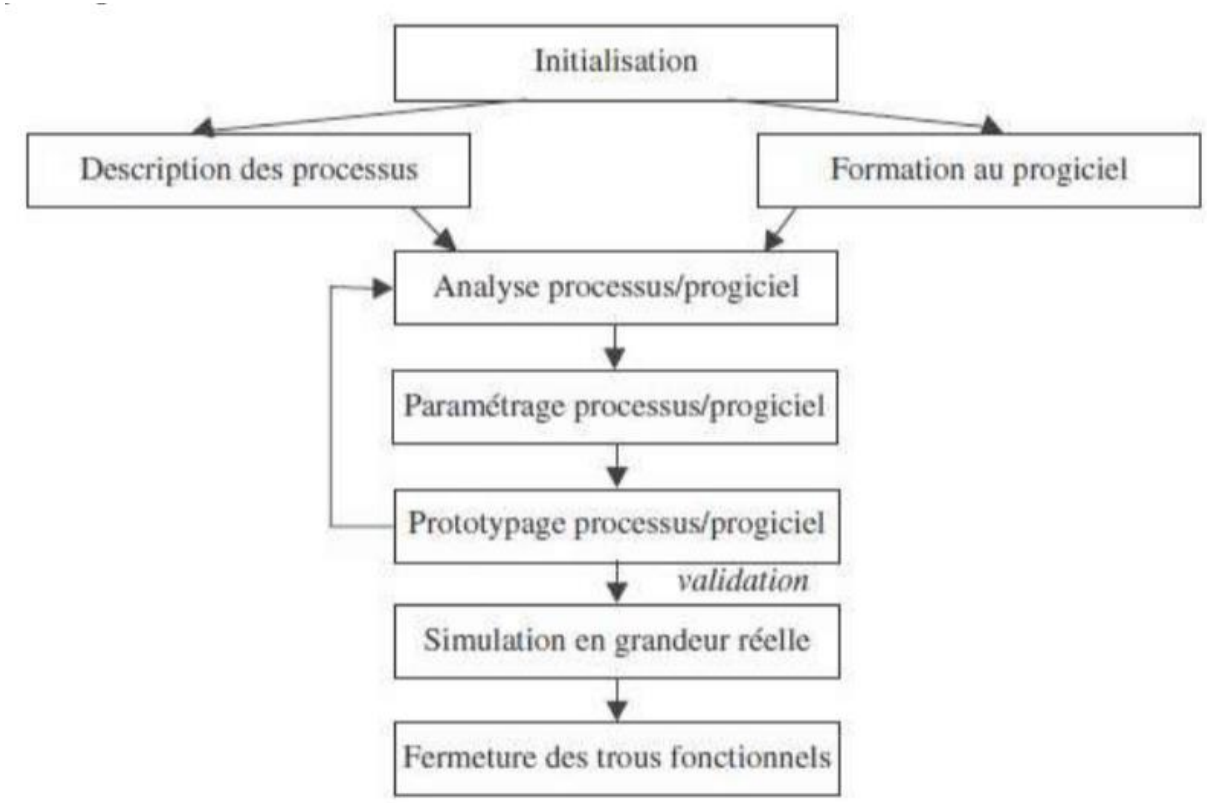
- La méthode SCRUM emprunte au vocabulaire du jeu le qualificatif des trois phases du cycle préconisé.
  - La phase d'Avant-jeu (pre-game), Conception et architecture du système, se déroule de façon structurée, en général linéaire, et permet de déterminer le périmètre, la base du contenu du produit à développer et une analyse de haut niveau.
  - La phase de Jeu (game) est itérative et qualifiée d'empirique, dans la mesure où le travail effectué ne fait pas l'objet d'une planification. Une itération, dont la durée oscille entre une et quatre semaines, est appelée un **sprint**, en référence à ces poussées rapides et fortes que les joueurs de rugby peuvent effectuer sur le terrain. Un sprint est découpé en quatre sous-phases :
    - Développement (develop) : il s'agit de déterminer l'objectif visé au terme de l'itération, de le répartir en « paquets » de fonctions élémentaires, de développer et tester chaque paquet.
    - Emballage (wrap) : on referme les « paquets » et on les assemble pour faire une version exécutable.
    - Revue (review) : une revue élargie permet de faire le point sur les problèmes et l'avancement.
    - Ajustement (adjust) : ajuster le travail restant.
  - La phase d'Après-Jeu (postgame), clôture, vise à livrer un produit complet et documenté. Comme dans la première phase, on peut en planifier les tâches et les dérouler de façon linéaire.



Méthodes de développement logiciel

Méthode spécifique : le cycle ERP

- La mise en place d'un progiciel de gestion intégré, souvent appelé du terme anglo-saxon ERP (Enterprise Resource Planning) s'appuie sur un découpage spécifique.
- En effet, il s'agit de construire, en tirant le meilleur parti du progiciel, un système améliorant la performance de l'entreprise. Deux étapes doivent donc être menées en parallèle : description des processus et formation au progiciel. Ensuite, il y a autant de cycles d'analyse — paramétrage — prototypage qu'il y a de processus. La validation par le comité de pilotage permet une simulation en grandeur réelle. Il faut alors prendre en compte ce qui est resté en dehors du champ couvert par le progiciel



Méthodes de développement logiciel

Le modèle RUP (Rational Unified Process)

- Le modèle RUP (Rational Unified Process) est représentatif d'une approche combinant plusieurs modèles. Sa structure fait l'objet d'un assez large accord, notamment parmi les praticiens. Il peut être lu de la façon suivante :
  - 1. Le cycle est constitué de quatre phases principales, que l'on retrouve globalement dans toutes les approches descendantes : étude préalable (opportunité), conception de la solution détaillée (élaboration), développement de la solution (construction) et mise en œuvre (transition).
  - 2. Il existe six types de tâches qui, au lieu d'être affectées exclusivement à une phase, se retrouvent à des degrés divers dans chacune des phases. Par exemple, l'étude des besoins peut apparaître jusqu'à la fin du projet, mais la plus grande partie est effectuée dans les deux premières phases. L'implémentation (développement) a principalement lieu dans la phase de construction, mais on peut réaliser un prototype dès la première phase. Certaines tâches, comme la direction de projet, s'effectuent sur toute la durée du projet.
  - 3. Certaines phases peuvent être menées de façon cyclique. Ainsi, l'élaboration se fait en deux cycles, conduisant par exemple à la production de spécifications externes (vision utilisateur) et spécifications techniques (vision développeur). La construction est itérative et incrémentale. De plus, l'ensemble du modèle représente un tour de spirale, dans le cas d'une approche globale en spirale.

TP : gestion des congés en entreprise

- Commande : Un logiciel de gestion des absences et des congés. Un manager doit pouvoir suivre les congés de son équipe. La RH suit les congés de tout le monde. Les demandes sont faites via l'interface par les employés, le N+1 valide pour ses subordonnés. Il y a plusieurs types de congés : vacances, RTT, congés maladie, jour fériés, congés de droit (mariage etc.).

- 1) Analyser le besoin
- 2) Modélisation avec UML
- 3) Etablir un plan de test
- 4) Implémenter les principales fonctionnalités

- Déposer le code et un rapport (documents hors code et autoanalyse) sur une nouvelle branche de github.com/mlesellier/Genie-logiciel

Projet pour le 11 décembre

- Préparer le projet d'un jeu de Scrabble
  - Pouvoir simuler le jeu (identifier les acteurs, implémenter les règles)
  - Vérifier que les mots proposés par un joueur sont corrects
  - Envisager la possibilité d'extensions (libre)

- Rendu demandé
  - Schéma du type de développement
  - Diagramme des cas d'utilisation
  - Diagramme des classes
  - Présentation courte (10 minutes/groupe)

Projet pour le 18 décembre

- Gestion d’une bibliothèque
  - Concevoir un système pour gérer les emprunts et retours de livres. Un bibliothécaire doit pouvoir ajouter des livres à la base.

- Rendu demandé
  - Choisir le schéma de développement approprié
  - Modéliser avec UML
  - Fournir un prototype en Python (et une petite base de test)
  - Présentation courte (10 minutes/groupe)

Projet pour le 8 janvier

- Risk
- Monopoly
- Colons de Catane

- Chaque jeu doit avoir un menu et une interface graphique qui doit permettre de jouer une partie
- Une structure MVC est demandée, elle doit clairement transparaitre dans le diagramme des classes
- Le développement se fera sur GitHub, créer l'espace le 19 décembre et envoyer le lien à maximilien.lesellier@univ-antilles.fr
- Commencez par établir un rétroplanning ou bien le programme de développement (par exemple les sprints si vous choisissez l'organisation SCRUM) et mettez le sur le git. Les documents de suivi de projet seront consultés pendant ces trois semaines, de même que les étapes de développement.
- Une présentation de 30 minutes environ, comprenant une démonstration du jeu et 5 à 10 minutes de questions sera faite le 8 janvier.
- La présentation sera remise avec le projet sur GitHub. Elle doit mettre en évidence les choix d'architecture, les packages utilisés, la manière d'implémenter les tests systématiques (tests unitaires, tests d'intégration), un début de documentation. Elle doit aussi décrire l'organisation du projet et justifier les choix.
- Il est attendu que vous produisiez un sous-ensemble pertinent et articulé des diagrammes UML présentés dans ce cours.
- Vous êtes encouragés à présenter des innovations ou des extensions à ces jeux de plateau très basiques.