Notes de cours

Génie Logiciel

Année académique 2008/2009

Sommaire

Notes Importantes

Pre-requis

- Analyse et conception (MERISE), OO (UML)
- Programmation orientée objet (Java, C++, C#, …)

Objectifs du cours

Ce cours a pour principal objectif de parcourir les différentes approches, techniques et méthodes utilisées dans la réalisation des logiciels critiques (à fort impact sur l'économie, la vie humaine, …) et de grande taille (mobilisant des ressources considérables). Il doit permettre à l'apprenant de :

- Comprendre ce que c'est le Génie Logiciel ainsi que ses objectifs,
- Connaître les différentes approches de développement logiciel,
- Percevoir de façon générique le processus unifié et ses caractéristiques,
- Maîtriser les différents enchaînements d'activités du processus unifié,
- Mettre en œuvre le processus unifié dans le cadre d'un projet.

Programme

Chapitre I . Introduction au Génie Logiciel Chapitre II .

Les modèles classiques de cycle de vie Chapitre III .

Introduction au processus de développement Chapitre IV .

Le Processus Unifié

Chapitre V . Les principaux enchaînements d'activités

Chapitre VI . Développement itératif et incrémental

Bibliographie

1. **Patrice CLEMENTE** : Cours de Génie Logiciel, 2éme année STI, ENSI BOURGES, 2003.
2. **Anne-Marie Hugues** : Génie Logiciel, 2002. http://www.polytech.unice.fr/~hugues/GL/
3. **Pierre-Alain MULLER**, **Nathalie GAERTNER**. Modélisation objet avec UML. Eyrolles. Edition 2000, 5ème tirage 2004.
4. **Ivar JACOBSON**, **Grady BOOCH**, **James RUMBAUGH**. Le Processus unifié de développement logiciel. Eyrolles, 2000.
5. **Alfred Strohmeler, Didier Buchs.** Génie Logiciel: principes, méthodes et techniques. Presses Polytechniques et Universitaires Romandes, 1996.

Mise en œuvre :

- Un projet ??
- Application au projet Tuteuré

Chapitre I . Introduction au Génie Logiciel

I.1. Définition

Qu'est ce que le GL ? Quand est-il né ? Pourquoi ?

Génie logiciel (« Software Engineering » en anglais) : Domaine des « sciences de l'ingénieur » dont la finalité est la *conception*, la *réalisation* et la *maintenance de systèmes logiciels complexes, sûrs et de bonne qualité*.

Un ***Logiciel*** est un ensemble *bien documenté* (à tous les niveaux) de programmes construit pour satisfaire les *besoins de ses utilisateurs*.

Le ***Génie Logiciel*** définit alors l'ensemble des méthodes et des techniques pouvant *faciliter* la conception et la réalisation des *logiciels complexes* et de *bonne qualité*.

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***.

Petit historique

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

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.

I.2. Les objectifs du GL

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**

=> Règle du CQFD[1] : **Coût Qualité Fonctionnalités Délai**.

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

[1] Cf Patrice CLEMENTE

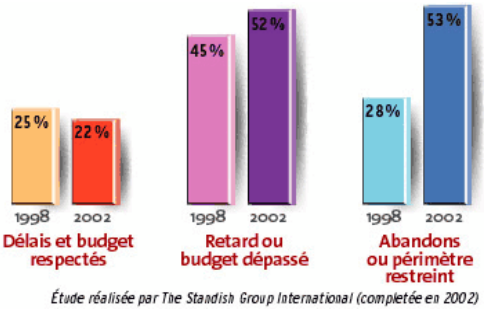
coûts (le coût du logiciel dépassait déjà celui du matériel) et des délais, la mauvaise qualité des produits (les produits ne répondent pas aux besoins définis et des erreurs persistent dans le produit final). Ces problèmes suscitent des ***risques humains*** et ***économiques*** comme l'illustrent les 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),
- Au Cameroun, les employés de l'état ont eu un retard sur le payement de leur salaire à cause d'une panne du logiciel traitant les salaires au CENADI.
- En 1999, les étudiants de la faculté des arts, lettres en sciences humaines de l'université de Yaoundé I entrent en grève à cause d'une erreur de calcul des moyennes sur les modules.
- Les origines de l'échec d'Ariane 5 (1996) sont liées à une exception logicielle qui a provoquée l'autodestruction d'un autre module empêchant ainsi la transmission des données d'attitude correctes (=> trajectoire erratique).
- En 2007, Express Union avec le passage de la numérotation téléphonique à 8 chiffres : Etait-ce vrai ?
- 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, …

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, soit deux fois plus qu'en 1998 (Le Monde 23/10/01).

Une autre étude menée par le même groupe sur la conduite des projets informatiques montre le résultat suivant :



Étude réalisée par The Standish Group International (complétée en 2002)

Les réponses à la crise

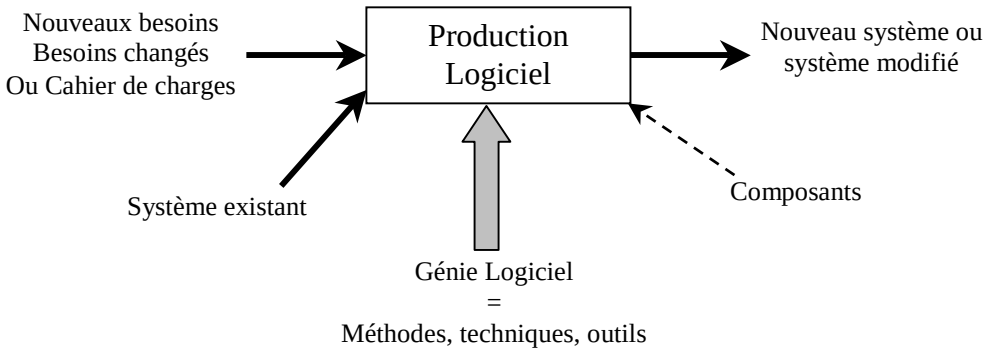
La solution imaginée pour répondre à cette crise a été l'**industrialisation de la production du logiciel**. Cette industrialisation vise la maîtrise du processus de développement et définit pour cela des procédés de fabrication de manière à satisfaire les ***besoins des utilisateurs***, la ***qualité*** du logiciel, les ***coûts*** et les ***délais*** de production.

Le GL a ainsi pour principal objectif la recherche permanente des moyens pour réussir à maîtriser le développement, le fonctionnement, la maintenance du logiciel. Ces moyens sont constitués de :

- Les ***modèles*** : Un modèle est une représentation simplifiée d'une réalité. Les modèles sont conçus avec un ensemble de concepts, dotés de règles d'utilisation et de représentation (souvent graphiques). Ils guident le raisonnement dans l'identification des aspects pertinents du domaine qu'on étudie.
- Les ***langages*** : Ils permettent de décrire les aspects pertinents du système. Ils peuvent être textuels ou graphiques, naturels ou formels.
- La ***démarche*** : Elle découpe le processus de conception en étapes successives à enchaîner.
- Les ***outils*** : Ils constituent le support automatique ou semi-automatique pour les méthodes. Ils permettent ainsi d'automatiser partiellement ou totalement certaines phases du processus de conception. On trouve ainsi des outils d'aide à la conception, des outils de simulation, des outils d'aide à la vérification, etc.

Remarque : Les modèles et les langages sont en général indissociables. Les modèles associés aux langages et une démarche constituent une méthode. Les ***techniques*** permettent un développement cohérent en précisant comment utiliser les méthodes et les outils.

La figure ci-dessous illustre la production du logiciel sous contrôle du génie logiciel.



I.3. Les difficultés du GL

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 (que pense le néophyte ?) ;
- 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 => 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) : La jeunesse du GL
- La réutilisation tarde à être effective : le développement par assemblage de composants est encore jeune dans le domaine logiciel (EJB, composants CORBA, .NET, ...).

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.

I.4. Les logiciels et métiers du GL

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, ...).

Pour produire ces différents systèmes, le GL en tant que vaste domaine faisant intervenir plusieurs acteur 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 planing and control)
- Formateur (software engineering education)

I.5. 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.

Illustration : La qualité des beignets ? d'un bâtiment ?

Idem GL : Il fait exactement ce que j'avais demandé, NON c'est difficile d'utiliser, AH c'est trop lent, ça consomme trop de ressource processeur, ça plante tout le temps, MERDE ça ne me rapporte rien, Il n'a pas produit le diagramme de cas d'utilisation.

Rappel : Le GL vise à produire des logiciels de qualité.

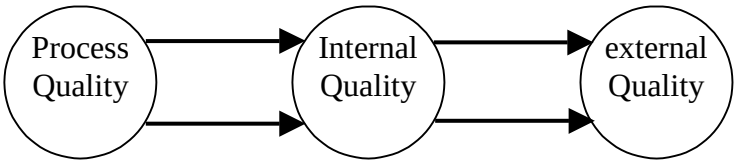
Qu'est ce qu'un logiciel de « bonne qualité » ?

CELA DEPEND (de quoi ?) Celui qui parle de qualité (acteur qui apprécie le logiciel : utilisateur, développeur, testeur, manager, …).

On distingue :

1. 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, …).
2. 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 ? L'environnement de travail, …).



La qualité liée au coût dépend de l'originalité (fonctionnalités/services rendus) du logiciel.

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. Privilégier une qualité en fonction de la nature du logiciel.

Ce que n'est pas la qualité d'un logiciel : 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.

Chapitre II . Les modèles classiques de cycle de vie

II.1. Introduction

La notion de cycle de vie permet de caractériser les étapes successives que peut prendre un logiciel depuis sa demande jusqu'à sa mort. Cela couvre de la création du logiciel à sa disparition en passant par son utilisation. Le but du découpage du processus est de maîtriser les risques, maîtriser au mieux les délais et les coûts, obtenir une qualité conforme aux exigences.

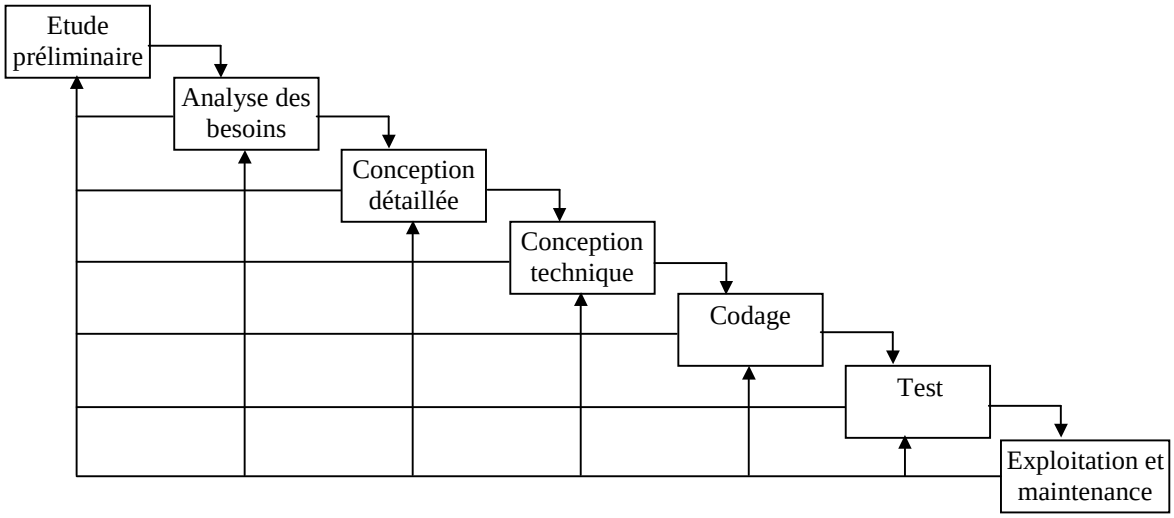
Remarque : Il ne faut pas confondre cycle de vie et cycle de développement. Le cycle de développement s'insère dans le cycle de vie et est abusivement appelé cycle de vie.

Historiquement, le modèle utilisé dans les premiers jours du développement logiciel consistait à coder et à réparer (« code and fix »). Ce modèle comprend deux étapes : écrire un peu le code, puis améliorer et corriger les erreurs. On commence donc à coder, puis on réfléchit sur les besoins afin d'adapter le code.

Les principaux modèles classiques de cycle de vie sont : le modèle en cascade (waterfall), le modèle en V, le modèle incrémental, le modèle en spirale. Tous ces modèles ne sont pas les mêmes, mais l'idée est toujours la même : le logiciel est développé en phases.

II.2. Modèle en cascade

C'est le tout premier modèle classique du cycle de vie. D'autres noms sont parfois donnés à ses phases et un découpage plus fin est souvent utile.



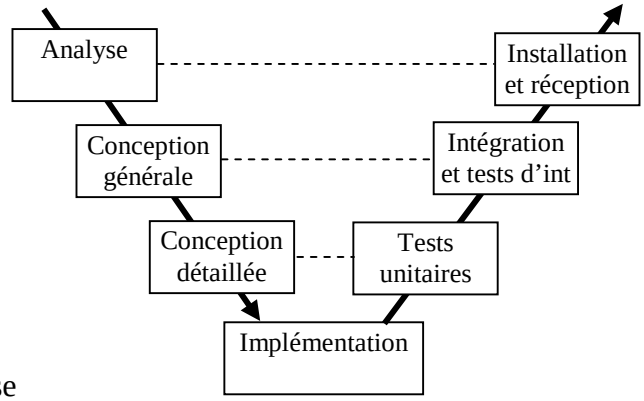
1. **Etude préliminaire** ou **étude de faisabilité** : Elle concerne la définition globale du problème. Il faut étudier la stratégie de l'entreprise et décide de la nécessité de fabriquer ou acheter un nouveau produit.
2. **Analyse des besoins** (quoi faire ?) : identifier les besoins de l'utilisateur (=> produire le cahier de charges)
3. **Conception détaillée** (comment faire) : organiser les besoins de l'utilisateur dans différents modules du système, faire la description détaillée des fonctions de chaque module en vue de l'implémentation.
4. **conception technique** (Avec quoi ?) : étudier le contexte d'implémentation (matériels, outils logiciels, …)
5. **Codage** : coder chaque module et le tester indépendamment des autres (test unitaires)
6. **Test** : Intégrer les différents modules et les tester dans l'ensemble par rapport aux besoins de

l'utilisateur.

7. **Exploitation et maintenance** : Maintenir (corrective, évolutive, adaptative) le logiciel en cours d'utilisation jusqu'à son retrait. C'est pendant cette phase que l'effort de documentation est bénéfique.

II.3. Modèle en V

Dérivé du modèle en cascade, le modèle en V montre non seulement l'enchaînement des phases, mais aussi les relations logiques entre phases plus éloignées : ce sont des liens de validation.

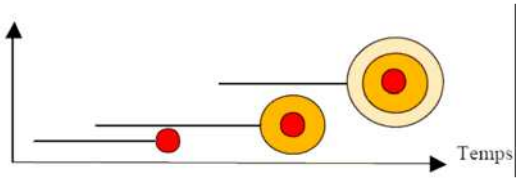


Quoi faire ? Analyse
Comment faire ? Conception générale
Comment faire par morceau ? conception détaillée

Comme les phases des modèles précédents sont successives, une difficulté majeure est rencontrée lorsque les besoins exprimés par le client ne sont pas complet au début du cycle. Il est alors possible de construire une maquette (prototype) qui simule le comportement du logiciel tel qu'il sera perçu par l'utilisateur.

II.4. Modèle incrémental

Ici, le produit est délivré en plusieurs fois, de manière incrémentale, c'est à dire en le complétant au fur et à mesure et en profitant de l'expérimentation opérationnelle des incréments précédents. Chaque incrément peut donner lieu à un cycle de vie classique plus ou moins complet. Les premiers incréments peuvent être des maquettes ou des prototypes (réutilisables pour passer au prochain incrément en les complétant et/ou en optimisant leur implantation). Le risque de cette approche est celui de la remise en cause du noyau ou des incréments précédents.

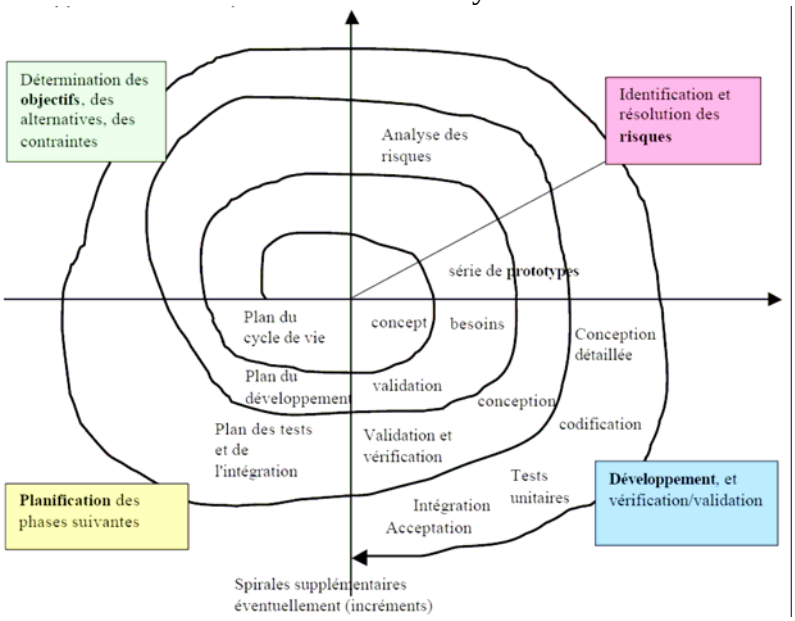


Dans le modèle incrémental, *chaque développement est moins complexe, les intégrations sont progressives, il y a la possibilité de livraison et de mise en service après chaque incrément, meilleure évaluation du temps et de l'effort de programmation*. Par contre, les risques peuvent être la mise en cause du noyau et des incréments précédents, l'éventuelle difficulté d'intégrer les nouveaux incréments.

II.5. Modèle en spirale

Ce modèle met l'accent sur l'activité d'analyse des risques. Chaque cycle de la spirale se déroule en quatre phases :

1. Détermination des objectifs du cycle, des alternatives pour les atteindre et des contraintes,
2. Identification et résolution des risques : évaluation des alternatives et, éventuellement maquettage,
3. Développement et vérification/validation de la solution retenue, un modèle « classique » (cascade ou en V) peut être utilisé ici,
4. Planification, revue des résultats et vérification du cycle suivant.



Dans le développement logiciel, tout dépend des circonstances et il n' y a pas de modèle idéal de cycle de vie. Souvent, un même projet peut mêler différentes approches, comme le prototypage pour les sous-systèmes à haut risque et la cascade pour les sous systèmes bien connus et à faible risque.

Le modèle de cycle de vie n'est pas une solution universelle. Malgré toutes les précautions prises, le processus de développement peut être bouleversé par des facteurs d'instabilité. Il en existe deux principales classes : les ***facteurs d'instabilité externes*** (évolution de l'environnement, de la législation, de la technologie, du marché et de la concurrence) et les ***facteurs d'instabilité internes*** (évolution de l'équipe du projet, nouvelles intégrations, …).

Chapitre III . Introduction au processus de développement

III.1. Définition

UML est un langage qui a connu un succès spectaculaire dans la modélisation des systèmes. Ce succès ne doit pas faire oublier qu'il ne s'agit que d'un langage de modélisation dont la vocation n'est pas de couvrir tous les aspects du génie logiciel. Les auteurs de UML sont conscients de l'importance d'un processus de développement, mais ils reconnaissent que cela passe d'abord par la disponibilité d'un langage de modélisation objet performant et stable.

Etant un langage de modélisation, UML ne définit pas un processus de développement particulier. Il sert d'appui pour différentes approches méthodologiques basées sur les objets. UML doit donc être utilisé dans le cadre d'un processus de développement pour qu'on en tire le meilleur profit.

Vu la complexité croissante des systèmes informatiques, les besoins de plus en plus exigeants des utilisateurs, tant au niveau des fonctionnalités que des délais de livraison, l'industrie logicielle a besoin d'un processus capable de guider les développeurs. Le processus a pour but de spécifier les différentes phases d'un projet, de l'élaboration du cahier de charge au déploiement de l'application.

Un processus définit qui fait quoi, à quel moment et de quelle façon pour atteindre un certain objectif.

Illustration : processus de livraison, processus d'inscription, processus d'achat, …

Dans le domaine de l'ingénierie logiciel, le but consiste à élaborer un produit logiciel ou en améliorer un existant. Un processus doit alors fournir les directives garantissant le développement efficace de logiciels de qualité et présenter un ensemble de bonnes pratiques autorisées par l'état de l'art. Un tel processus est indispensable et doit servir de fil conducteur à tous les intervenants (client, utilisateurs, développeurs et responsables).

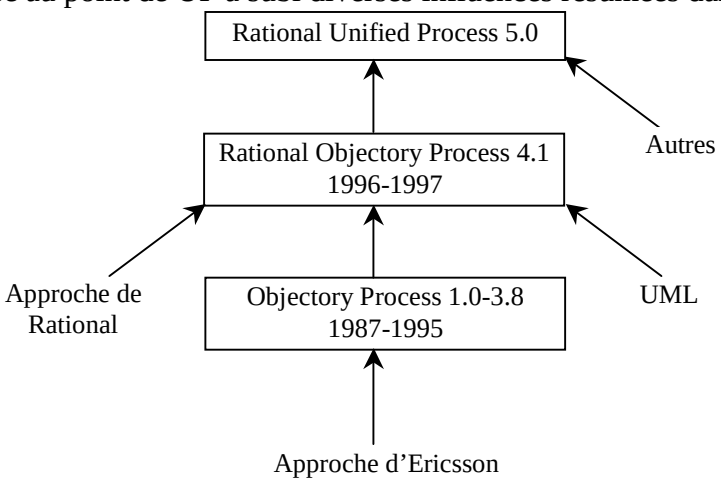
Un **processus de développement logiciel** est un ensemble d'activités coordonnées et régulées pour transformer les besoins d'un utilisateur en produit logiciel.

Le processus unifié est le principal processus de développement logiciel proposé par les auteurs d'UML.

III.2. Historique du processus unifié

Le processus unifié (UP : Unified Process) est un processus générique pouvant être adaptée à une large classe de systèmes logiciels, à différents domaines d'application, à différents types d'entreprises, à différents niveaux de compétences et à différentes tailles de projets. Il est à base de composants et utilise le langage UML pour la construction du système logiciel : UML et UP ont été développé en concert.

La mise au point de UP a subi diverses influences résumées dans la figure ci-dessous :



En 1967, Ericson modélisa le système comme un ensemble de blocs interconnectés (appelés « sous-systèmes » en UML et implémentés sous forme de « composants »). La description de l'architecture consistait à décrire tous les blocs et leurs interconnections (transimission des messages).

En 1987, Jacobson quitta Ericson pour fonder Objectory AB qui développa Objectory Process. Plusieurs versions se sont succédées.

En 1995, Rational Software Corporation ayant acquis Objectory AB, menait en son sein une approche de développement basée sur les notions orientée objet, abstraction, réutilisabilité et prototypage. Le fruit de l'expérience de Rational et les pratiques mises en œuvre se sont associées à Objectery Process pour former Rational Objectory Process (l'architecture était constituée des vues architecturales des différents modèles). UML était alors en cours de développement et servait de langage de modélisation.

Dans le même temps, Rational se lançait dans une politique de fusion-acquisition de sociétés éditrices de logiciels (Raquisite, SQA, Pure-Atria, Performance Awareness et Vigortech). Chacune d'elle contribua, dans son champ d'expertise au développement de Rational Objectory Process et dès les milieux 1998 (Juin), les diverses contributions se sont unifiées et Rational lança la nouvelle version du produit, Rational Unified Process 5.0.

Aujourd'hui, UP est un processus générique de développement et plusieurs variétés de processus y découlent.

III.3. Instances du Processus Unifié

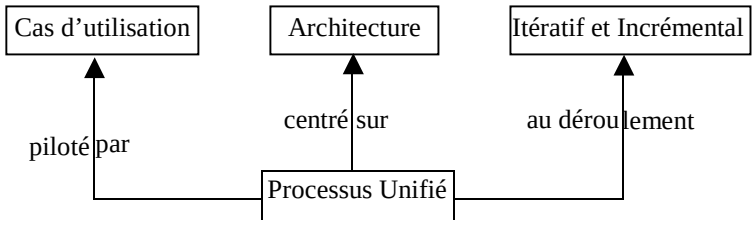
- RUP : Rational Unified Process, instanciation par Rational des préceptes UP
- EUP : Enterprise Unified Process, instanciation intégrant les phases de post-implantation.
- XUP : Extreme Unified Process, instanciation hybride intégrant UP avec Extreme Programming.
- AUP : Agile Unified Process, partie des préceptes UP permettant l'agilité du développement.
- 2TUP : Two Tracks Unified Process, instanciation de UP proposé par Valtech prenant en compte les aléas et contraintes liées aux changements perpétuels et rapides des SI des entreprises.
- EssUP : Essential Unified Process, instanciation de UP proposé par Ivar Jacobson Consulting propose une mouture du processus unifié qui intégre certains concepts de la méthodes Agile.

Chapitre IV . Le Processus Unifié

IV.1. Introduction

Le Processus Unifié est piloté par les cas d'utilisation, centré sur l'architecture et déroulé de manière itérative et incrémentale.

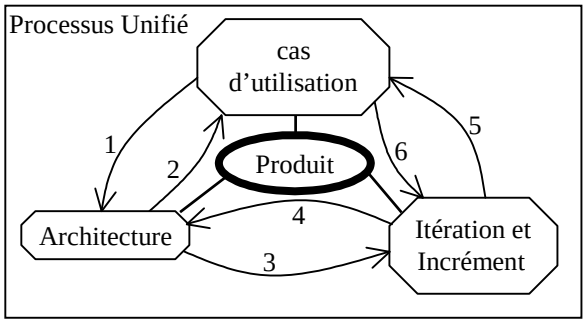
La figure ci-dessous résume les caractéristiques de UP.



- **Piloté par les cas d'utilisation** : toutes les activités, de l'analyse des besoins jusqu'aux tests sont guidés par les cas d'utilisation
- **Centré sur l'architecture** : tous les intervenants au projet de développement, du chef projet au programmeur doivent s'accorder sur la vision commune du système à produire : l'architecture. Elle offre une perspective claire de tout le système, indispensable pour en maîtriser le développement.
- **Au déroulement itératif et incrémental** : l'ensemble du travail est divisé en petites parties, qui sont autant de mini-projets. Chacun d'entre eux représente une itération qui donne lieu à un incrément. Les itérations désignent des étapes de l'enchainement des activités tandis que les incréments correspondent à des stades de développement du produit.

Les auteurs de UP précisent ce qui suit : **Les concepts de « développement piloté par les cas d'utilisation », « centré sur l'architecture » et « itératif et incrémental » sont d'égale importance. L'architecture fournit la structure qui servira de cadre au travail effectué au cours des itérations, tandis que les cas d'utilisation définissent les objectifs et orientent le travail de chaque itération. L'abandon de l'une ou l'autre de ces trois notions clés atténuerait considérablement la valeur du Processus Unifié. Comme un tabouret auquel il manquerait un pied, il s'effondrerait.**

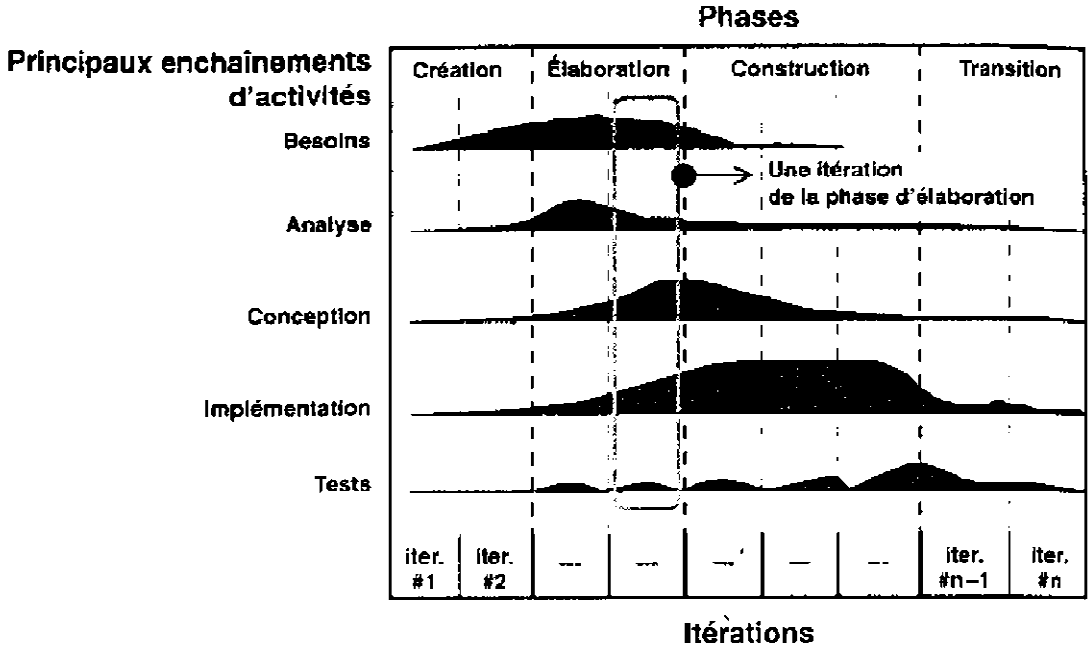
La figure ci-dessous matérialise les relations de dépendance entre ces concepts en vue de développer un produit satisfaisant.



1. orientent le développement
2. guide la réalisation
3. sert de cadre
4. se déploie
5. réalise
6. définissent les objectifs

IV.2. La vie du Processus Unifié

Le Processus Unifié gère le développement suivant deux axes : l'axe horizontal représente le temps et montre le déroulement du cycle de vie du processus et l'axe vertical représente les principaux enchaînements d'activités. La figure suivante représente le cycle de vie de UP selon les deux axes.



IV.2.1. Axe horizontal

Le Processus Unifié répète un certain nombre de fois une série de cycles constituant la vie d'un système. Chaque cycle se déroule sur une certaine durée et s'articule en 4 phases : création, élaboration, construction et transition. Chacune d'elles pouvant se subdiviser à son tour en itérations.

Les différentes phases de cycle sont les suivantes :

Phases	Description
Phase de création ou d'inception	Traduit une idée en vision de produit fini et présente une étude de rentabilité pour ce produit - Que va faire le système pour les utilisateurs ? => un modèle simplifié des cas d'utilisation regroupant les principaux cas d'utilisation. - A quoi peut ressembler l'architecture d'un tel système ? => architecture provisoire, une ébauche relevant les principaux sous-systèmes - Quels sont l'organisation et les coûts du développement de ce produit ? => planifier en détail la phase d'élaboration et fournir une estimation du projet dans son ensemble. Dans cette phase doit être identification et hiérarchiser les risques majeurs
Phase d'élaboration	Permet de préciser la plupart des cas d'utilisation et de concevoir l'architecture du système. L'architecture doit être exprimée sous forme de vue de chacun des modèles. A l'issue de cette phase, le chef de projet doit être en mesure de prévoir les activités et d'estimer les ressources nécessaires à l'achèvement du projet. Cette phase aboutit à l'émergence de la stabilité des cas d'utilisation, d'une **architecture de référence**, d'un plan de développement et la maîtrise des risques pour permettre le respect du contrat. Les cas d'utilisation, l'architecture et les plans sont-ils assez stables et les risques suffisamment maîtrisés pour permettre la réalisation du développement dans le respect du contrat ?
Phase de construction	Moment où l'on construit le produit. Au cours de cette phase, l'architecture de référence se métamorphose en produit complet, elle est maintenant stable. Le produit contient tous les cas d'utilisation que les chefs de projet, en accord avec les utilisateurs ont décidé de mettre au point pour cette version. Celle-ci doit encore avoir des anomalies qui peuvent être en partie

	résolue lors de la phase de transition. Le produit satisfait-il suffisamment aux besoins des utilisateurs pour que certains clients l'adoptent avant sa sortie officielle ?
Phase de transition	Le produit est en version bêta. Un groupe d'utilisateurs expérimentés essayent le produit et rendent compte des anomalies et défauts détectés. Les développeurs corrigent les problèmes signalés et incorporent certaines améliorations suggérées à la version suivante. Cette phase suppose des activités comme la formation des utilisateurs clients, la mise en œuvre d'un service d'assistance et la correction des anomalies constatées.

IV.2.2. Axe vertical

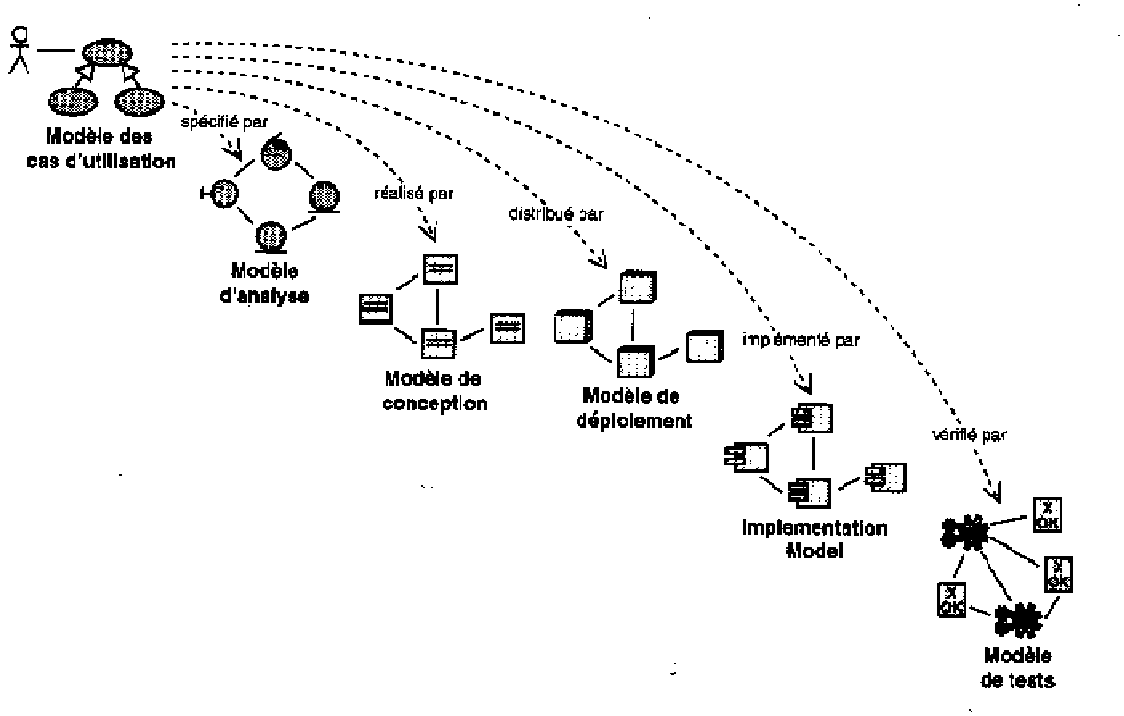
Chaque cycle se conclut par la livraison d'une nouvelle version du système. Ce produit se compose d'un corps de code source réparti sur plusieurs composants pouvant être compilés et exécutés et s'accompagne de manuels et de produits associés.

Les différents modèles à produire pendant les activités sont les suivants :

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

Tous ces modèles sont liés. Ensemble, ils représentent le système comme un tout. Les éléments de chacun des modèles présentent des dépendances de traçabilité. Il est par exemple possible de remonter à un cas d'utilisation (dans le modèle des cas d'utilisation) à partir de sa réalisation (dans le modèle de conception) ou d'un cas de test (dans le modèle de test). La traçabilité facilite la compréhension et les modifications ultérieures.

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



La figure du cycle de vie de UP indique les enchaînements d'activités, tandis que les courbes représentent **approximativement** le degré d'accomplissement des activités dans chaque phase. Une itération couvre généralement les cinq enchaînements d'activités.

IV.3. Processus piloté par les cas d'utilisation

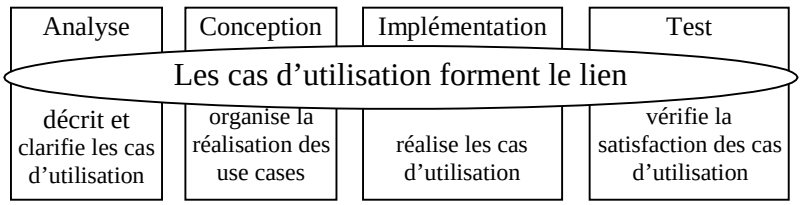
IV.3.1. Intérêt des cas d'utilisation

Le but principal d'un système informatique est de satisfaire les besoins du client. La détermination et la compréhension de ces besoins sont souvent difficiles. Les cas d'utilisation (*use cases*) est une technique de détermination intégrée dans UML. Ils privilégient le point de vue de l'utilisateur. Ils recentrent ainsi l'expression des besoins sur les utilisateurs, en partant du principe qu'un système est avant tout construit pour ses utilisateurs.

Les cas d'utilisation ont pour objectif de comprendre et structurer les besoins des utilisateurs. Ils guident pour cela la **conception**, l'**implémentation** et les **tests**. C'est à partir du modèle des cas d'utilisation que les développeurs créent une série de modèles de conception et d'implémentation réalisant les cas d'utilisation. Chacun des modèles successifs est ensuite révisé pour en contrôler la conformité par rapport au modèle des cas d'utilisation. Enfin les testeurs testent l'implémentation pour s'assurer que les composants du modèle d'implémentation mettent correctement en œuvre les cas d'utilisation.

Dire que UP est piloté par les cas d'utilisation signifie qu'il procède par une série d'enchaînements d'activités dérivés des cas d'utilisation puisque la plupart des activités (analyse, conception, implémentation et test) sont effectuées à partir des cas d'utilisation. Les cas d'utilisation guident ainsi le processus de développement.

La figure ci-dessous présente le lien que les cas d'utilisation nouent entre les différentes étapes du processus.



En résumé, UP est piloté par les cas d'utilisation parce que ces derniers :

1. permettent d'identifier les véritables besoins : Un cas d'utilisation est une séquence d'actions qu'effectue le système afin de produire un résultat satisfaisant pour un acteur particulier. L'identification des cas d'utilisation est fondamentale pour le processus unifié. Ils représentent les fonctionnalités que fournit le système pour rendre service à ses utilisateurs. Il faut se placer du point de vue de chaque type d'utilisateur (au sens acteur) et identifier les cas d'utilisation qui lui permettront de faire son travail. Il est donc indispensable de savoir quels sont les utilisateurs du système à produire. Il ne faut pas se mettre à identifier les fonctions du système sans se poser la question de savoir quels sont ceux qui les utiliseront. Cela conduit généralement à l'identification des cas inutiles ou superflus.
2. dirigent le processus : ils aident les chefs de projets à planifier, affecter et surveiller les tâches effectuées par les développeurs.
3. constituent un mécanisme essentiel de traçabilité entre modèles : il est possible de suivre l'évolution d'un cas d'utilisation de la phase d'analyse à sa réalisation et à travers les cas de test qui en assurent la vérification. Cette traçabilité permet de préserver la cohérence du système et de l'actualiser dans son ensemble en fonction de l'évolution des besoins.
4. servent de guide pour les besoins non fonctionnels : A tout besoin fonctionnel, délimité sous forme de cas d'utilisation, peut être associé une grande partie de besoins non fonctionnels (performance, disponibilité, sécurité, …).

IV.3.2. Réalisation des cas d'utilisation

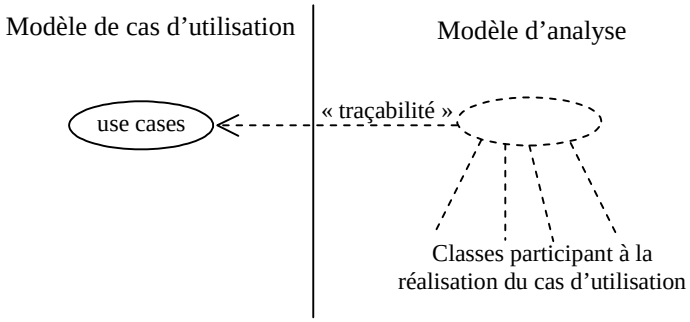
Des cas d'utilisation au modèle d'analyse

Le modèle d'analyse a pour rôle de clarifier en fournissant une spécification détaillée des cas d'utilisation. Il permet une meilleure compréhension de ceux-ci. Il consiste en :

1. une étude/description détaillée des cas d'utilisation l'un après l'autre ;
2. une identification des classificateurs (ainsi que leurs rôles et relations) intervenant dans la réalisation de chaque cas d'utilisation étudié/décrit ;
3. la représentation de la collaboration entre les classes identifiées, chacun jouant effectivement son ou ses rôles.

Le modèle d'analyse tient lieu de première ébauche du modèle de conception, tout en étant un modèle à part entière. La figure ci-dessous illustre la traçabilité entre un cas d'utilisation et sa réalisation dans le modèle d'analyse.

La réalisation d'un cas d'utilisation est représentée par une ellipse en pointillées.

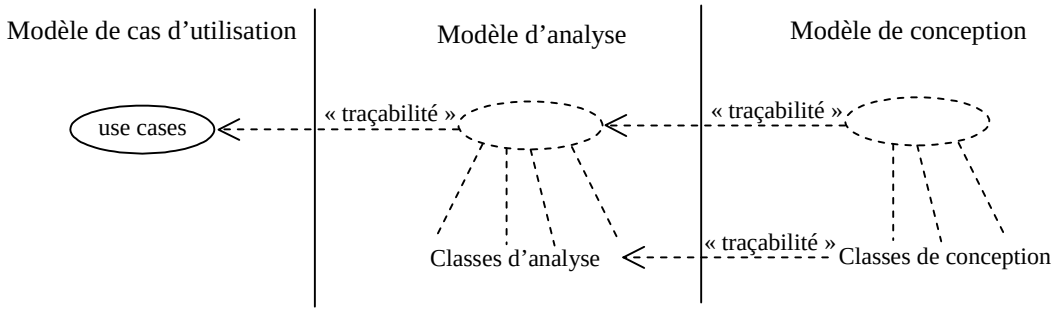


Du modèle d'analyse au modèle de conception

Le modèle de conception est conçu en vue de l'implémentation. Il est ainsi plus proche du modèle d'implémentation. Toutefois, il est d'abord crée à partir du modèle d'analyse avant d'être adapté à l'environnement d'implémentation choisi. A l'instar du modèle d'analyse, le modèle de conception définit les relations entre les classificateurs et les collaborations réalisant les cas d'utilisation. Les éléments définis dans le modèle de conception sont les équivalents des éléments plus conceptuels

spécifiés dans le modèle d'analyse. Les éléments du modèle de conception sont plus « physique » (adaptés à l'environnement d'implémentation) alors que ceux du modèle d'analyse sont plus « conceptuel ».

La réalisation des cas d'utilisation dans les différents modèles obéissent à plusieurs objectifs. Les différentes classes participants à la réalisation d'un cas d'utilisation dans le modèle d'analyse donnent lieu à des classes de conception plus sophistiquées, adaptées à l'environnement d'implémentation. Toutefois, la traçabilité permet de remonter des classes de conception du modèle de conception vers les classes d'analyse du modèle d'analyse. On doit ainsi savoir quelles classes d'analyse participent à la réalisation de quel cas d'utilisation, et quelles classes de conception correspondent à quelle classe d'analyse pour la réalisation de quel cas d'utilisation.

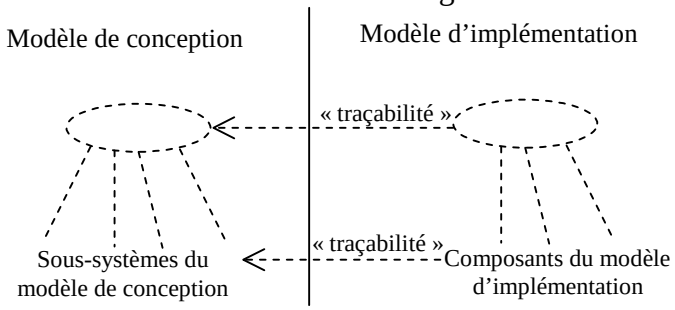


Remarques : Généralement, une classe du modèle d'analyse peut s'éclater en plusieurs classes du modèle de conception.

Du modèle de conception au modèle d'implémentation

Dans le modèle d'implémentation sont développés tous les éléments (composants, code sources, scripts shell, bibliothèques, éléments de BD, …) nécessaires à la production du système exécutable. Ce sont généralement les sous-systèmes du modèle de conception qui sont implémentés en composants avec des interfaces bien définies. Ces composants favorisent le déploiement dans les nœuds du modèle de déploiement.

L'implémentation ne se borne pas au développement du code nécessaire à la création du système exécutable. Les développeurs chargés d'implémenter les composants doivent également les tester unité par unité avant de les soumettre aux tests d'intégration et aux tests système.



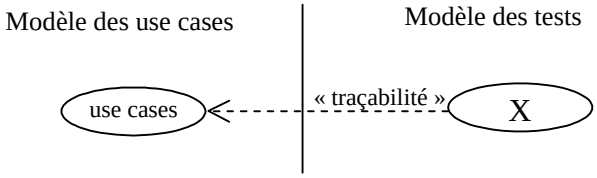
Tests des cas d'utilisation

Les tests permettent de vérifier que le système implémente correctement sa spécification. Le modèle de test est composé des cas de test et de procédures de test que l'on exécute ensuite pour se rassurer que le système fonctionne comme initialement prévu. Les anomalies détectées suite au test sont ensuite analysées afin de localiser le problème. Enfin les problèmes sont hiérarchisés et corrigés par ordre d'importance.

L'identification des cas d'utilisation dès le début du processus permet de planifier très tôt les activités de test et de suggérer des cas de test utiles. Ces cas de test peuvent ensuite être améliorer au cours de la conception , lorsqu'on en sait un peu plus sur la façon dont le système exécutera les cas d'utilisation.

Les cas d'utilisation peuvent être testés soit du point de vue d'un acteur traitant le système comme une boîte noire, soit d'un point de vue propre à la conception ; le cas de test est alors élaboré pour vérifier que les instances des classes participant à la réalisation du cas d'utilisation font bien ce qu'elles doivent faire.

Traçabilité des cas de test : Un cas de test est représenté par le symbole du cas d'utilisation revêtu d'une croix.



IV.4. Processus centré sur l'architecture

IV.4.1. Intérêt de l'architecture

Le rôle de l'architecture logicielle est comparable à celui que joue l'architecture dans la construction d'un bâtiment. Un bâtiment ne peut être construit sans plan (même conçu dans la tête pour des petits projets de construction). Du point de vue du client, un bâtiment forme en général une seule unité. Pourtant il peut être intéressant pour l'architecte de créer des dessins du bâtiment de différents points de vue. En principe, peu détaillés, ces dessins sont compréhensibles par le client. Mais, la phase de construction d'un bâtiment implique plusieurs professionnels : maçons, charpentiers, ouvriers, plombiers, électriciens, … Chacun d'eux a besoin de dessins techniques du bâtiment plus détaillés et spécialisés qui, tous doivent être cohérents les uns avec les autres (le circuit électrique et celui d'adduction d'eau ne doivent pas se trouver au même emplacement physique). L'architecte est un expert chargé d'intégrer tous les aspects du bâtiment, mais il n'est spécialiste d'aucun des domaines. Ce sont les concepteurs qui lui apportent tous les détails. Une fois que les dessins architecturaux (les différentes vues détaillées) sont prêts, les professionnels les utiliseront pour produire une image fidèle du bâtiment.

De même qu'un bâtiment, un logiciel est vu comme une entité unique par les clients. Même si l'architecte logiciel et les développeurs préfèrent le présenter selon différents points de vue pour mieux en maîtriser la réalisation. Un logiciel vaste et complexe nécessite une architecture qui pourra conduire tous les intervenants du projet à une vision commune. Cette architecture permet de :

- **Comprendre le système** : l'architecture permet aux intervenants de comprendre avec suffisamment de détails ce qui doit être fait et de favoriser ainsi leur participation. En effet, les systèmes à développer ont un comportement complexe, ils fonctionnent dans des environnements complexes (évolutions technologiques) et sont de plus en plus exigeants (systèmes distribués, embarqués, …). Cela devient plus complexe lorsqu'on sait que ces facteurs changent constamment.
- **Organiser le développement** : l'architecture permet de faciliter la coordination des efforts de développement et de limiter les dépendances entre groupes de développeurs (sous-systèmes). En effet, la définition explicite des interfaces d'une architecture permet à chaque sous-système d'évoluer de façon indépendante.
- **Favoriser la réutilisation** : l'architecture définit des composants appropriés pour réaliser le modèle des cas d'utilisation. Ces composants peuvent être développés, et si les développeurs en trouvent d'autres obéissant à la configuration du système exigée, il devront trouver moyen de les relier pour satisfaire les cas d'utilisation. Cela réduit considérablement les délais et les coûts de construction.
- **Faire évoluer le système** : l'architecture permet au système d'être facilement modifiable. En effet, les systèmes évoluent même au cours de leur développement. On doit pouvoir modifier ou ajouter de nouvelles fonctions sans se soucier des répercutions inattendues que pourraient avoir

ces changements ou sans constater d'effet spectaculaire sur la conception et l'implémentation existantes.

IV.4.2. Architecture et cas d'utilisation

Le processus de développement ne se résume pas en un enchaînement d'activités guidés par les seuls cas d'utilisation. Les cas d'utilisation une fois réalisés doivent trouver leur place dans l'architecture. Les différentes vues du système à construire sont ressorties dans cette architecture, qui doit prévoir la réalisation de tous les cas d'utilisation.

L'architecture est influencée par les cas d'utilisation car le souci est d'obtenir une architecture adaptée à l'implémentation des cas d'utilisation. Pour le faire, on part des cas d'utilisation les plus significatifs et indispensables, ceux qui constituent le cœur même des fonctions du système. Les cas d'utilisation significatifs sont ceux qui permettent de réduire les risques les plus importants, ceux qui comptent le plus pour les utilisateurs et ceux qui aident à couvrir les principales fonctionnalités.

Toutefois, l'architecture ne subit pas la seule influence des cas d'utilisation, elle doit prendre en compte les contraintes de réalisation (la plate-forme sur laquelle sera exécuté le système, le SGBD, le SE, les protocoles de communication réseau - distribution, middleware, …). L'expérience acquise sur les travaux antérieurs se révèle également utile pour la mise au point de l'architecture.

Les cas d'utilisation subissent aussi l'influence de l'architecture déjà en place. Il est donc indispensable de construire d'abord une architecture provisoire à partir d'une bonne compréhension du domaine, sans toutefois considérer les détails des cas d'utilisation. Puis on choisit quelques cas d'utilisation et l'on adapte l'architecture pour qu'elle prenne en charge les cas d'utilisation considérés. On prend ensuite d'autres cas d'utilisation, en fonction desquels on améliore encore l'architecture, et ainsi de suite. A chaque itération, on sélectionne et on implémente un ensemble de cas d'utilisation afin de valider et si nécessaire d'améliorer l'architecture. Les cas d'utilisation contribuent, par conséquent à l'amélioration progressive de l'architecture au fur et à mesure qu'on s'achemine vers le système complet.

Les cas d'utilisation et l'architecture doivent évolués en parallèle et cette dernière doit avoir la capacité de réaction aux futurs changements et la réutilisation.

IV.4.3. Construction de l'architecture

Architecture et patterns d'architecture

L'architecture est mise au point par itérations successives, essentiellement pendant la phase d'élaboration. La phase d'élaboration débauche sur une architecture de référence. L'on doit définir et décider des différents standards à respecter, des logiciels systèmes et middleware à utiliser, des systèmes existants à réutiliser et des besoins en matière de distribution. Après tout cela, il est produit les premières versions des différents modèles (cas d'utilisation, analyse, conception, …). Cet ensemble de modèles (que comportera le système à la fin de la phase de construction) constitue l'architecture de référence (la version interne du système telle qu'elle existe à la fin de la phase d'élaboration). Elle donne un squelette du système et est obtenu à partir des cas d'utilisation significatifs. A travers elle, les différents acteurs ont un aperçu du système et peuvent déjà y réagir.

Chaque modèle faisant partie de l'architecture de référence représente la version du modèle mise au point à la fin de la phase d'élaboration. Chacun doit émerger vers la version complète destinée au client. Les différents modèles ne sont pas développés indépendamment les uns des autres. Par exemple une modification dans le modèle de déploiement visant à satisfaire les exigences de performance est susceptible de conduire à des remaniements dans le modèle des cas d'utilisation, si ces changements nécessitent de modifier certains cas d'utilisation : dépendances de traçabilité entre différents modèles.

Toutefois, l'architecture de référence ne se limite pas aux éléments de modèles. Elle comprend également une description de l'architecture. En réalité, cette description s'élabore au fur et à mesure (et parfois même en amont) des activités qui donnent naissance aux versions des modèles faisant partie de l'architecture de référence. Le rôle de la description de l'architecture consiste à guider l'équipe de

développement tout au long de la durée de vie du système et non seulement pour les itérations du cycle en cours. Cette description doit être stable et est un standard à suivre par les développeurs présents et ceux à venir.

La mise au point d'une architecture pour un système particulier revient à créer quelque chose de nouveau. D'autre part il existe de nombreuses « solutions » génériques d'architecture : les patterns d'architecture. Il en existe aussi pour la conception : les patterns de conception. Un pattern est définit comme « une solution à un problème récurrent de conception ». Les patterns d'architecture les plus utilisés sont :

- Le pattern « Broker » : mécanisme générique de gestion de la distribution des objets. Les objets distribués communiquent par l'intermédiaire d'un « broker » de façon transparente (l'objet appelant n'a pas besoin de savoir si l'objet appelé est distant ou non).
- Les pattern client-serveur (3-tiers), Peer-to-peer : facilitent la conception du système au dessus du matériel. Ils définissent une structure pour le modèle de déploiement et suggèrent le mode d'affectation des composants sur les nœuds.
- Le pattern « Couches » (Layers) : définit l'organisation du système en couches, selon laquelle les composants d'une couche ne peuvent faire référence qu'aux composants de la couche située immédiatement au-dessous.

Remarque : On peut utiliser plusieurs patterns d'architecture au sein d'un même système et il peut arriver qu'un pattern prédomine sur les autres.

Description de l'architecture

L'architecture de référence survit sous la forme d'une description de l'architecture. Cette description peut revêtir diverses formes et contient des extraits (ou vues) des modèles de l'architecture de référence, qui seront actualisés au gré de l'évolution du système et de l'enrichissement des modèles dans les phases plus tardives (après la phase d'élaboration).

La description de l'architecture sera actualisée tout au long du processus de développement, afin de refléter les changements et ajouts pertinents sur le plan architectural. Pendant que les différents modèles de l'architecture de référence sont modifiées en réaction aux divers changements, la description de l'architecture n'est pas nécessaire d'être étoffée, elle doit simplement être actualisée pour rester pertinente.

La description de l'architecture doit également :

- intégrer les besoins significatifs sur le plan architectural qui ne sont pas décrit dans les cas d'utilisation. Par exemple la répartition, les terminaux/équipements en jeu, …
- contenir une brève description de la plate-forme, des systèmes existants et protocoles à utiliser, … Exemple de Java RMI pour la distribution des objets, .NET, EJB ou Fractal pour le modèle de composants, …
- décrire les outils mettant en œuvre les mécanismes génériques. Par exemple les mécanismes de stockage et récupération à travers les SGBD, les mécanismes de journalisation, de reprise après panne, de substitution de nœuds, de répartition de charges (entre serveurs ou processus), …
- décrire tous les patterns d'architecture auxquels on aura eu recours.

La description de l'architecture doit susciter les réactions des participants au développement. Ces réactions doivent faire l'objet de discussions, être analysées et finalement tranchées.

La description de l'architecture doit être suffisamment détaillée (présenter de façon compréhensible tout ce qui est susceptible d'intéresser l'un ou l'autre des participants pour effectuer son travail), sans toutefois prétendre à l'exhaustivité. C'est une feuille de route et non une spécification complète du système. Il ne faut pas égarer les participants dans les détails inutiles sur le plan architectural. Généralement, moins de 10% des cas d'utilisation et classes sont pertinents pour l'architecture.

Les auteurs de UP estiment que la taille d'une description architecturale pour des systèmes à une seule application est variable (pas une règle absolue) et doit comprendre de 50 à 100 pages.

L'architecte crée l'architecture en compagnie de quelques développeurs. Les développeurs savent ce qu'ils ont à mettre en œuvre et c'est à l'architecte de choisir les patterns d'architecture, les systèmes

existants et organiser les dépendances entre sous-systèmes et façon à éviter un fort couplage entre eux (les changements intervenants dans un sous-système ne doivent pas se répercuter sur plusieurs autres sous-systèmes). Le véritable objectif étant de répondre aux besoins de l'application en utilisant les meilleurs moyens disponibles dans l'état actuel de la technologie, l'architecte doit avoir connaissance (en plus de l'expérience acquise des projets antérieurs) :

- du domaine dans lequel il travaille : il doit être suffisamment informé pour collaborer avec tous les intervenants et non pas seulement avec les développeurs ;
- du développement logiciel : il doit connaître jusqu'à l'écriture du code car il doit communiquer l'architecture aux développeurs et comprendre leurs réactions.

L'architecte occupe donc un poste délicat dans l'équipe de développement. Il crée l'architecture, assure sa maintenance et veille à son application (vérification et validation). Il peut être nécessaire d'avoir recours un plusieurs architectes pour développer une architecture dans le cadre d'un projet. Il est important de consacrer du temps pour produire une architecture stable car cela guide les autres phases de développement et la durée totale de développement décroit nettement lorsqu'une architecture stable est développée.

IV.5. Processus itératif et incrémental

IV.5.1. C'est quoi itératif et incrémental ?

Un projet logiciel s'étend sur plusieurs mois, voire sur une année ou plus. Pour être efficace, le processus logiciel doit œuvré pour une stratégie de développement par petite étapes facile à gérer :

- On planifie un peu ;
- On spécifie, on conçoit et on implémente un peu ;
- On intègre, on teste et on exécute le peu implémenté.

L'ensemble de ces points constituent une itération destinée à réaliser une partie du projet (mini-projet). Le processus de développement parcourt alors chaque phase par une série d'itérations, chacune donnant lieu à un incrément. L'ensemble des itérations d'une phase concourent à atteindre les objectifs de cette phase. Pour cela, il est indispensable de contrôler les itérations. Une itération s'achève par un jalon qui se définit par un ensemble d'artefacts.

Artefact : information créée, modifiée et utilisée par les travailleurs (une personne ou une équipe) dans la réalisation de leur activité. Il peut être un modèle, un élément de modèle, un document, … Bref un travailleur réalise des activités et produit un ensemble d'artefacts.

Jalons : Ce sont des points de synchronisation où sont atteints des objectifs précis, où sont contrôlés (vérifiés et validés) le développement et où sont prise des décisions importantes.

Jalon majeur : jalon de fin de phase. Prise de décision de poursuivre ou non la phase suivante du processus de développement.

Jalon mineur : jalon intermédiaire entre deux jalons majeurs. Ça peut être la fin d'itération.

En gros, les jalons permettent :

- Aux chefs projets de prendre un certain nombre de décisions cruciales avant de passer à la phase suivante du développement ;
- Aux développeurs et autres responsables de surveiller eux-mêmes la progression du travail ;
- Evaluer les besoins en terme de délai et de personnel, de contrôler l'avancement par rapport aux projections.

Qu'est ce qu'une itération ?

Une itération est un déroulement plus ou moins complet des principaux enchaînements d'activités, aboutissant à une version livrée en interne. L'enchainement principal d'activités est constitué de besoins, analyse, conception, implémentation et test. Cet enchaînement définit un canevas pour la description d'enchainements d'activités d'itération. Les itérations parcourent les cinq activités principales en débutant

par la planification et s'achèvent par une évaluation : chaque type d'itération réutilise à sa façon les descriptions des enchaînements principaux d'activités. En effet, les défis auxquels sont confrontés les développeurs diffèrent dans chaque phase. Les premières itérations s'attachent à la compréhension du problème et de la technologie.

- Dans la phase de création, les principaux objectifs des itérations consistent à délimiter la portée et le rôle du produit, à réduire les risques majeurs (les plus sérieux) et à produire une première étude de rentabilité démontrant la viabilité commerciale du projet. Le jalon majeur est l'établissement des objectifs du cycle de vie du projet.
- Les principaux objectifs de la phase d'élaboration consistent à créer l'architecture de référence, à saisir l'essentiel des besoins et à réduire les risques de moindre gravité. A l'issu de cette phase, on est en mesure d'estimer les coûts, de prévoir le calendrier de développement et de planifier en détail la phase de construction. On peut donc faire une offre.
- La phase de construction a pour principaux objectifs le développement du système complet et de s'assurer que le produit peut commencer à être utilisé par les clients. Le jalon est que le produit atteigne une capacité opérationnelle initiale.
- Les principaux objectifs de la phase de transition consistent à s'assurer que l'on dispose d'un produit prêt à être livré à l'ensemble des utilisateurs, qui sont formés à ce moment à l'utilisation du logiciel.

Dans les premières phases, un incrément ne constitue pas nécessairement un additif. Dans les phases suivantes, les incréments ne sont généralement que des additifs successifs qui finissent par constituer la version du produit destinée au client (version externe).

Chaque itération fait l'objet d'une évaluation à son terme. L'un des objectifs de cette évaluation consiste à déterminer si de nouveaux besoins sont apparus ou si des besoins existants ont subi des modifications susceptibles d'affecter les itérations ultérieures. Avant de terminer une itération, il faut se rassurer qu'aucune partie du système qui fonctionnait dans les itérations précédentes n'a été endommagée. D'où l'importance capitale des tests de non-régression dans le développement itératif et incrémental car chaque itération produit un résultat qui s'ajoute à l'incrément précédent et entraîne un certain nombre de changements.

Test de non-régression : Fait de tester de nouveau une partie ou la totalité d'une construction déjà testée lors des constructions précédentes. Elles servent essentiellement à vérifier qu'une « ancienne fonction » d' « anciennes constructions » fonctionne toujours lorsqu'est ajoutée une « nouvelle fonction » dans une « nouvelle construction ».

Si toutes les itérations parcourent tous les enchaînements d'activités des besoins, de l'analyse, de la conception, de l'implémentation et des tests, chacune insiste sur différents points selon la phase abordée (voir les courbes indicatives du cycle de vie). Au cours des phases de création et d'élaboration, les efforts se portent essentiellement sur l'expression des besoins, ainsi que sur l'analyse et la conception préliminaire, tandis que l'on insiste surtout, dans les phases de construction et de transition, sur la conception détaillée, l'implémentation et les tests.

Planifier et séquencer les itérations

Les itérations doivent être sélectionnées et menées à bien de façon planifiée. Une itération prend en compte un certain nombre de cas d'utilisation. A chaque itération les développeurs identifient et spécifient les cas d'utilisation pertinents, créent une conception en se laissant guidé par l'architecture choisie, implémentent cette conception sous forme de composants et vérifient que ceux-ci sont conformes aux cas d'utilisations. Dès qu'une itération répond aux objectifs qui lui sont fixés, le développement peut passer à l'itération suivante.

Dans l'approche en cascade, tout le projet est planifié à l'avance et cette planification repose sur une grande part d'incertitude. En itératif, le projet n'est pas tout planifié en détail au cours de la phase de création. Pendant les deux premières phases, il existe un plan de travail qui n'entre pas dans les détails : la phase de construction et de transition ne peuvent pas être planifiée sans la mise en place d'une base solide à la fin de la phase d'élaboration. En principe (sauf au début du projet), la planification prend en compte

les résultats des itérations précédentes. A la fin de la phase d'élaboration, on dispose d'une base permettant de planifier le reste du projet et d'avancer un plan détaillé pour chaque itération de la phase de construction. Le plan de la première itération sera très clair. Celui des itérations suivantes comportera moins de détails, sera sujet à d'éventuelles modifications et reposera sur les résultats des itérations précédentes. De même, on devra disposer d'un plan de la phase de transition qui sera susceptible de changer selon les résultats des itérations de la phase de construction.

Cette planification permet de mener un développement itératif contrôlé. Elle tente d'ordonner les itérations de façon à tracer une voie directe entre itérations. L'objectif étant de trouver une séquence d'itérations qui permette d'aller de l'avant, sans jamais avoir à revenir aux résultats d'une itération antérieure pour reprendre le modèle en fonction des connaissances issues d'une itération plus tardive.

Les itérations successives exploitent les artefacts de développement dans l'état où les a laissé l'itération précédente et les enrichissent progressivement. Une itération n'est donc pas une entité totalement indépendante : c'est une étape d'un projet. Les qualifier de « mini-projets » c'est parce qu'elles ne satisfont pas entièrement les besoins initiaux des utilisateurs. En outre, chacun de ces mini-projets s'apparente au modèle en cascade, car il met en œuvre les activités de ce type de modèle.

Les itérations peuvent se chevaucher dans le sens où une itération débute alors que la précédente est sur le point de s'achever. Cependant, il ne faut pas que ces chevauchements aillent très loin, car une itération constitue toujours le fondement de l'itération suivante. L'ordre dans lequel sont prévues les itérations dépend largement des facteurs techniques. L'objectif étant d'organiser de sorte que les décisions les plus importantes (celles permettant de gérer les risques techniques c'est-à-dire concernant les nouvelles technologies, les cas d'utilisation, l'architecture) soient prises très tôt.

Les itérations contribuent à planifier, organiser, surveiller et contrôler le déroulement du projet. Elles présentent chacune leurs propres besoins en termes de ressources humaines, financement, calendrier, … C'est à travers elles que les développeurs recherchent l'équilibre entre les cas d'utilisation et l'architecture.

IV.5.2. Intérêt du développement itératif et incrémental

Le développement itératif et incrémental permet d'atteindre les jalons majeurs et mineurs qui permettent de maîtriser le développement. Il aide à :

- **réduire les risques** : Plus le risque est identifié tardivement, plus il est de nature à menacer tout le projet. Les risques graves sont identifiés dans les deux premières phases (création et élaboration) et gérés le plus tôt possible au moyen des itérations. Tous les risques peu sérieux sont traités par ordre d'importance dès le début de la phase de construction.
- **obtenir une architecture robuste** : L'obtention d'une architecture robuste est le résultat des itérations menées dans les premières phases du processus. On stabilise ainsi l'architecture à un niveau de référence très tôt dans le cycle de vie, lorsque les coûts sont encore faibles et que les délais de livraison restent un horizon lointain.
- **gérer les exigences de changement** : il est plus productif de faire évoluer un produit à travers une série de constructions. Une **construction** est une version opérationnelle d'une partie d'un système faisant la démonstration d'un sous-ensemble des fonctions du système. Les itérations permettent d'obtenir une série de constructions pour approcher progressivement le résultat prévu. Le fait de disposer d'un système en état de fonctionnement partiel dès les premières phases permet aux utilisateurs et aux intervenants de suggérer des améliorations et de signaler les besoins ayant pu être négligés : les ajouts, modifications des exigences et leur intégration sont aisément réalisable sans impact (budget et calendrier), car ne représentent qu'un changement incrémental. Un **incrément** est une partie modeste du système constituant l'écart entre deux constructions successives. Dans le modèle en cascade, les utilisateurs ne voient aucun système en état de fonctionnement avant l'intégration et les tests : un moindre changement entraîne les rallonges budgétaires et temporelles.
- **parvenir à une intégration continue** : Etant donnée la livraison fréquente des constructions, les itérations produisent des résultats indiquant très précisément l'état d'avancement du projet. Cela

(l'implémentation qui débute très tôt) permet de mettre à jour très tôt les problèmes. Par contre, à l'absence de l'intégration continue, il est probablement possible que la tâche la plus ardue reste à venir. En cascade par exemple, les développeurs ne débutent pas l'implémentation avant d'avoir finalisé l'expression des besoins, l'analyse et la conception : les problèmes demeurent, à un niveau très bas jusqu'à l'intégration et les tests qui les révèlent en bloc. Cette intégration ayant lieu proche de la date de livraison et relevant une panoplie de problèmes, l'ampleur de ces derniers et l'inévitable précipitation qui règne à ce stade signifient que la plupart des corrections ne seront pas sérieusement préparées. Ce qui provoque nécessairement une livraison au-delà de la date prévue. En plus, le produit risque d'être difficile à maintenir.

- **accéder très tôt à la connaissance** (assistance à l'apprentissage et à l'acquisition d'expérience) : Après une ou deux itérations, tous les membres de l'équipe se font une idée assez claire de la nature des différents enchaînements d'activités. Ils en apprennent tôt et toute la suite leur sera mieux abordable car ce sont les mêmes activités qui seront menées à différentes phases. L'intégration de nouveaux collaborateurs est assez aisée. Ils parcourront toutes les activités comme s'ils intégraient le projet à son début. S'ils commettent une erreur, celle-ci ne met pas en péril la progression à long terme du projet, puisqu'elle apparaît dès la tentative de construction suivante. Le problème d'organisation se gère progressivement car l'équipe de développement s'étoffe au fur et à mesure que le projet avance : l'équipe d'origine est optimisé et ce noyau de l'équipe peut guider les nouveaux venus au rythme de leur arrivée.

En développant par phase et par itération, les développeurs ont de meilleurs chances de satisfaire les besoins réels des clients, de réduire les risques, d'observer le niveau de progression, de réduire les difficultés tardives (qui raccourci le délai). Cette approche est aussi bénéfique aux utilisateurs qui peuvent prendre la mesure de véritables progrès en prenant acte des itérations achevées.

IV.5.3. La réduction des risques

Un **risque** est un facteur, un phénomène ou un élément constituant un danger (une exposition) pouvant conduire à une perte ou à un dommage. Il met en danger, voire compromet totalement la réussite d'un projet : retard de calendrier, dépassement de budget, annulation du projet. L'approche itérative est guidée par la réduction des risques. Les cas d'utilisation sont identifiés, hiérarchisés et menés en fonction des risques et de leur ordre d'importance. L'objectif étant d'identifier et de traiter les risques le plus tôt possible. Toutefois, il est nécessaire de mener l'exploration des risques dans les itérations tout au long du cycle de développement jusqu'au codage et aux tests. En effet, certains risques relèvent des performances, de la fiabilité, de la disponibilité et de la portabilité. Beaucoup de ces risques n'apparaissent pas tant que le logiciel mettant en œuvre les fonctions sous-jacentes n'a pas été implémenté et testé.

Il est important de noter que les **risques techniques** peuvent être atténués par les cas d'utilisation (si le cas d'utilisation est réalisé avec tous ces besoins fonctionnels et non fonctionnels). Ils sont regroupés en catégorie et peuvent être liés :

- Aux nouvelles technologies : par exemple le problème de dépendance aux techniques qui ne sont pas encore au point (ou mûres) : reconnaissance du langage naturel, technologies du web, nouveau protocole réseau, …
- A l'architecture : ce sont les plus importants. La difficulté de prise en compte des changements (adaptation, évolution), d'intégration des composants réutilisables, … peuvent surgir si on n'a pas obtenu au terme de la phase d'élaboration une architecture robuste.
- A la construction du système attendu : cette catégorie de risque souligne l'importance de l'identification des besoins qui revient à trouver les « bons » cas d'utilisation. Sinon le système ne sera pas capable de satisfaire ses missions et d'assister les utilisateurs qui l'emploient. Il est fondamental de déterminer très tôt quelles sont les fonctions les plus importantes et de s'assurer qu'elles sont rapidement implémentées. L'ordre dans lequel sont traités les cas d'utilisation, au moment de leur sélection est fonction du degré et du type de risque qu'ils présentent. Par exemple l'exécution même d'un cas d'utilisation peut entraîner des risques.
- Aux performances : il peut être nécessaire de prévoir qu'un cas d'utilisation aura un temps de

réponse bien définit, …

Il est nécessaire que les intervenants du projet examinent le système à réaliser, dressent les listes des problèmes susceptibles de se produire et programment des réunions d'identification des risques. Non pas pour résoudre les problèmes mais pour les identifier et établir l'ordre dans lequel ils seront étudiés plus en détails et résolus.

Risques techniques : ceux liés à des artefacts d'ingénierie et à des aspects tels que les technologies d'implémentation, l'architecture ou les performances.

Risques non techniques : ceux liés à des artefacts de gestion et à des aspects tels que les ressources disponibles (humaines), leurs compétences ou les dates de livraison de leurs travaux.

Les risques non techniques sont ceux à identifier et écarter par les responsables :

- Implémenter certaines parties du système dans un langage qu'on découvre seulement ;
- Le calendrier proposé par le client semble trop serré et ne pourra être tenu que si chaque étape se déroule sans moindre problème ;
- Le respect du calendrier proposé est conditionné par la livraison aux dates prévues, de sous-systèmes (composants) développés par des sous-traitants ;
- Il est possible que le client ne puisse pas toujours donner son approbation dans les délais fixés.

Une fois les risques identifiés et hiérarchisés, l'équipe doit décider du traitement à appliquer à chacun d'entre eux. En général, quatre attitudes peuvent être adoptées :

- Evitement : certains risques peuvent être évités en révisant la planification du projet ou en modifiant les exigences ;
- Confinement : la portée de certains risques peut être réduite de façon à n'affecter qu'une petite partie du projet ou du système ;
- Surveillance : certains risques sont rebelles à toute tentative de réduction. Il ne reste qu'à les surveiller et regarder s'ils se matérialisent. Dans ce cas, l'équipe doit appliquer ses **plans de secours** (plan décrivant la conduite à adopter au cas où certains risques se matérialiseraient). Si c'est un risque « tueur de projet », à cette étape, l'investissement temporel et financier reste encore timide et l'équipe doit se réjouir d'avoir détecté un tel risque avant d'impliquer tous les développeurs dans le projet.

Certaines solutions exigent généralement une révision du planning, tandis que d'autres peuvent nécessiter la construction d'un élément permettant de mettre à jour ce risque. Dans tous les cas, on ne peut traiter tous les risques à la fois. D'où la hiérarchisation des itérations qui guide la réduction des risques.

Chapitre V . Les principaux enchaînements d'activités

Une fois compris les concepts qui sous-tendent le Processus Unifié, nous allons décrire chacun des enchaînements d'activités.

V.1. Compréhension du contexte du système

Il est indispensable de comprendre le contexte du système à l'aide d'un modèle du domaine. Ce modèle saisit les classes les plus pertinentes dans le domaine du système. Ces classes sont l'ensemble de ce qui existe ou des évènements qui se produisent dans l'environnement au sein duquel fonctionne le système. On les retrouve en général en interviewant les experts du domaine. Le modèle du domaine est décrit par les **diagrammes de classes**. Il illustre pour les clients, les utilisateurs et les développeurs les classes du domaine et la façon dont elles sont liées par des relations d'association.

La modélisation du domaine est effectuée par les analystes compétents en matière de modélisation à qui on associe les experts du domaine. L'objectif étant de comprendre et de décrire les classes essentielles dans le contexte du domaine contribuant à une meilleure compréhension du contexte du système et du problème qu'est censé résoudre le système par rapport à son contexte.

Remarques : Pour ne pas encombrer le modèle du domaine, certaines classes moins pertinentes peuvent être décrites dans un glossaire. En gros, le glossaire et le modèle du domaine offre aux intervenants un vocabulaire commun, indispensable au partage de connaissances. Cela évite les confusions et les ambiguïtés.

Certains éléments du domaine peuvent avoir une représentation directe dans le système. Il ne faut pas se préoccuper dans le modèle du domaine à spécifier les détails de cette représentation.

Il ne faut pas se mettre à aller en détails dans la gestion des comptes des clients (un client a plusieurs comptes ou un compte appartient à plusieurs clients, …).

Le glossaire et le modèle du domaine sont très utiles au moment de l'élaboration du modèle des cas d'utilisation et du modèle d'analyse.

Une fois le domaine du système compris à travers le modèle du domaine, les paragraphes qui suivent parcourent les enchaînements d'activités principaux des besoins aux tests.

V.2. Expression des besoins

V.2.1. Présentation

Le principal objectif de l'expression des besoins est l'élaboration d'un modèle du système à construire. L'emploi des cas d'utilisation constitue un excellent moyen de procéder à la création de ce modèle car les besoins fonctionnels sont naturellement structurés comme des cas d'utilisation. Par ailleurs, les besoins non fonctionnels liés à un cas d'utilisation sont traités dans le cadre de ce dernier. Les autres besoins non fonctionnels (communs à une partie ou à la totalité des cas d'utilisation) sont traités dans un document à part et désigné sous le noms d'exigences supplémentaires.

Exemples d'exigence supplémentaires :

Exigence de plate-forme matérielle : PC Pentium, …

Contraintes de plate-forme logicielle : SE clients (Win NT), SE serveurs (Linux, Solaris)

Contraintes de formats de fichiers : prise en charge des noms de fichiers longs, …

Autres exigences : sécurité (la transaction doit être sûre et seules les personnes autorisées doivent accéder aux informations), disponibilité (le service de facturation ne doit pas être indisponible plus d'une heure par mois), …

Les besoins se capturent principalement au cours des phases de création et d'élaboration (voir

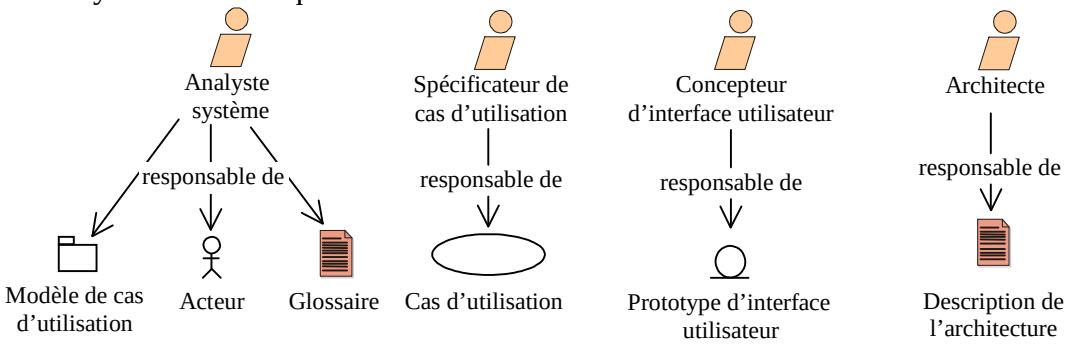
cycle de vie UP).

La description de l'enchaînement d'activités des besoins s'articule autour :

- Des artefacts crées dans l'enchaînement d'activités des besoins ;
- Des travailleurs prenant part à cet enchaînement d'activités ;
- D'une vision plus détaillée de chaque activité de l'enchaînement d'expression des besoins.

Le figure ci-dessous donne les différents travailleurs ainsi que les artefacts dont ils sont responsables.

Remarque : Les symboles spéciaux sont utilisés pour la plupart des artefacts : ceux représentant des documents sont désignés par un symbole de document, tandis que les modèles et éléments de modèles utilisent les symboles UML qui les identifient.



V.2.2. Les artefacts

a) ***Modèle des cas d'utilisation :*** C'est le principal artefact de l'expression des besoins. Il permet de parvenir à un accord entre le client et les développeurs sur les besoins c'est-à-dire les conditions que le système doit respecter et les possibilités qu'il doit offrir. Il sert de contrat et fournit les entrées pour l'analyse, la conception et les tests. Un modèle de cas d'utilisation contient les acteurs, les cas d'utilisation et les relations qu'ils entretiennent les uns avec les autres. Si ce modèle est important en taille, il faut y introduire les paquetages afin de mieux le gérer.

b) ***Acteur :*** Le modèle de cas d'utilisation décrit l'utilité du système pour chaque type d'utilisateur. Les acteurs sont les rôles des utilisateurs ou systèmes externes avec lequel dialogue le système. Une fois tous les acteurs du système identifiés, on a identifié l'environnement externe du système.

Remarque : Attention d'identifier de faux acteurs (par exemple ceux qui n'interagissent pas directement avec le système, mais par un intermédiaire).

A chaque fois qu'un acteur dialogue avec le système, c'est l'instance correspondante de l'acteur qui joue le rôle de cet acteur. L'instance d'un acteur est un utilisateur (ou un système) spécifique dialoguant avec le système.

c) ***Cas d'utilisation :*** Chaque usage que les acteurs font du système est représenté par un cas d'utilisation. Un cas d'utilisation spécifie une séquence d'actions que le système peut effectuer en dialoguant avec les acteurs.

d) ***Description de l'architecture :*** La description architecturale contient une vue architecturale du modèle des cas d'utilisation. Cette vue doit inclure les cas d'utilisation significatifs pour l'architecture, c'est-à-dire ceux stratégiques ou impliquant certaines exigences fondamentales qui doivent être prises en charge au début du cycle de vie du logiciel.

e) ***Glossaire :*** Il permet de définir les termes qu'utilisent les analystes (et les autres développeurs) pour décrire le système. Ceci permet aux intervenants de parvenir à un consensus sur la définition des divers concepts et notions, en limitant les risques de malentendu.

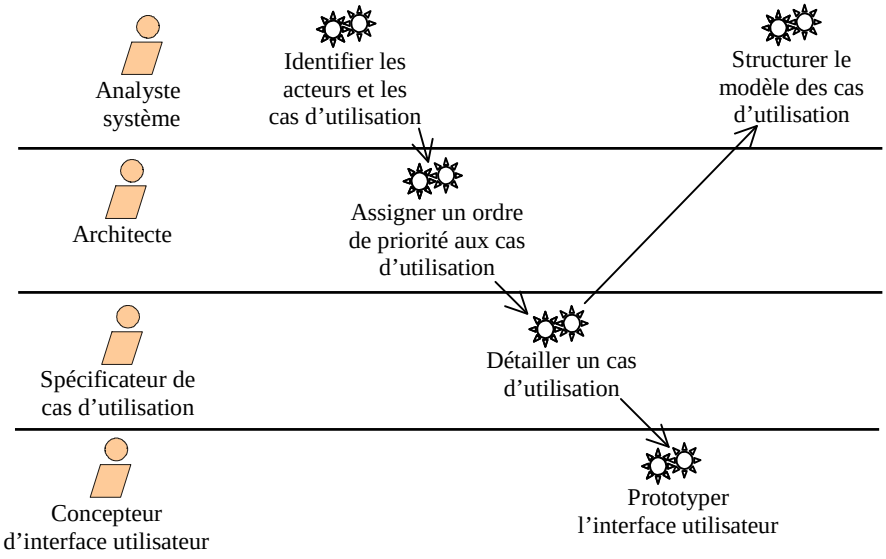
f) ***Prototype d'interface utilisateur :*** Les prototypes d'interface utilisateur contribuent au cours de l'expression des besoins, à faciliter la compréhension et la spécification des interactions entre acteurs et le système.

V.2.3. Les travailleurs

a) ***Analyste système*** : Il est le responsable de l'ensemble des besoins modélisés en tant que cas d'utilisation. Il est chargé de délimiter le système, d'identifier les acteurs et les cas d'utilisation et de s'assurer que le modèle de cas d'utilisation est à la fois complet et cohérent. Il peut donc avoir recours au glossaire pour le respect de la cohérence. Bien qu'étant responsable de l'ensemble des cas d'utilisation, il n'est pas responsable de chaque cas d'utilisation particulier, car cela est confié aux spécificateurs de cas d'utilisation.
b) ***Spécificateur de cas d'utilisation*** : Il est responsable de la formulation d'un ou de plusieurs cas d'utilisation. Il doit travailler en étroite collaboration avec les utilisateurs réels des cas d'utilisation dont il a la charge.
c) ***Concepteur d'interface utilisateur*** : Il est chargé de dessiner l'aspect visuel de l'interface utilisateur d'un ou de plusieurs acteurs. Cette tâche peut nécessiter la création de prototype d'interface utilisateur pour certains cas d'utilisation.
d) ***Architecte*** : Il a la responsabilité de décrire la vue architecturale du modèle des cas d'utilisation.

V.2.4. Enchaînement d'activités

L'enchaînement d'activités des besoins peut se résumer par le « diagramme d'activités » ci-dessous.



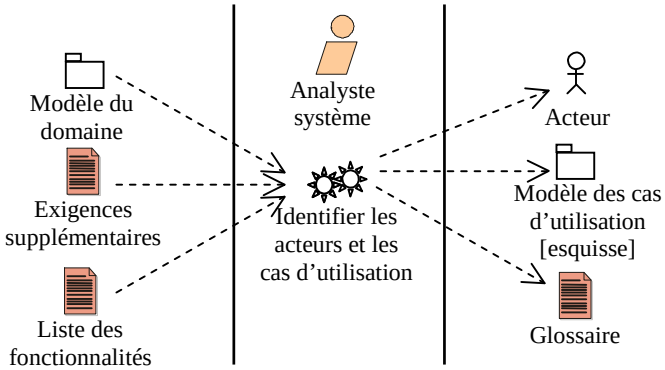
Ce diagramme matérialise la répartition des activités entre les divers intervenants. Chaque activité est représenté par un engrenage et placé dans la zone du travailleur chargé de l'effectuer. L'exécution de ces activités conduit à la création, puis à la modification de divers artefacts. L'enchaînement d'activités est ordonné de sorte que chacune d'elle produise un résultat qui servira de point de départ à l'activité suivante.

Remarque : L'enchaînement ci-dessus ne présente qu'un flux logique. Dans la réalité, il n'est pas indispensable de mener ces activités de façon séquentielle pour aboutir au même résultat. On pourrait par exemple commencer par identifier certains cas d'utilisation (activité « identifier les acteurs et les cas d'utilisation »), puis concevoir l'interface utilisateur (activité « prototyper l'interface utilisateur ») avant de s'apercevoir qu'il convient d'ajouter un autre cas d'utilisation (retour sur l'activité « identifier les acteurs et les cas d'utilisation »), et ainsi de suite.

Une activité peut ainsi être reprise plusieurs fois, chacun de ces passages n'aboutissant qu'à la réalisation partielle de l'activité. Toutefois, les activités doivent suivre un enchaînement logique et l'exécution d'une activité doit utiliser les résultats issus de l'activité précédente comme point de départ.

Les activités effectués sont les suivantes :

1. ***Identifier les acteurs et les cas d'utilisation*** : Cette activité permet de délimiter le système par rapport à son environnement, de faire apparaître les éléments (acteurs) interagissant avec le système, les fonctionnalités (cas d'utilisation) attendues du système, de saisir et de définir dans un glossaire les termes essentiels à la création de descriptions détaillées des fonctionnalités du système.

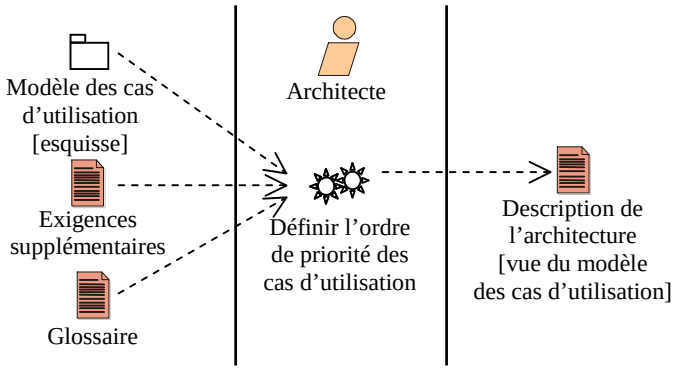


- ***Identifier les acteurs*** : Lorsqu'il existe un modèle de domaine, l'identification des acteurs est presque directe. Dans le cas contraire, l'analyste système doit identifier, avec l'aide du client les utilisateurs du système et tenter de les répartir en différentes catégories représentant les acteurs. Deux critères sont applicables pour identifier les acteurs pertinents : (1) il doit être possible d'identifier au moins un utilisateur susceptible d'incarner l'acteur proposé ; (2) éviter le chevauchement entre les rôles (si plusieurs acteurs ont à peu près le même rôle, il est préférable de regrouper ces rôles en un ensemble unique pris en charge par un seul acteur, soit de trouver un autre acteur plus général auquel sont attribués les rôles communs à ces différents acteurs).
- ***Identifier les cas d'utilisation :*** Elle peut se faire par le biais des rencontres entre client et ses utilisateurs et l'entretien entre l'analyste système et ces derniers. L'analyste système examine les acteurs un par un et propose les cas d'utilisation de chacun. L'acteur a le plus souvent besoin des cas d'utilisation pour effectuer les tâches consistant à créer, modifier, supprimer, étudier les objets métier. Il en a également besoin pour informer le système de certains évènements externes ou d'être informer par le système de la survenue d'un évènement donné. D'autres acteurs peuvent assurer la surveillance, l'administration ou la maintenance du système. Il convient de bien délimiter la **portée des cas d'utilisation** en gardant à l'esprit qu'un cas d'utilisation doit être facile à modifier, à tester et à gérer en tant qu'« unité individuelle ». Un cas d'utilisation doit rendre service à l'acteur, livrer un résultat observable et satisfaisant aux attentes de ce dernier et lui permettre d'atteindre un objectif. Le nom d'un cas d'utilisation commence généralement par un verbe et doit refléter la nature de l'interaction entre l'acteur et le système.
- ***Décrire brièvement chaque cas d'utilisation :*** Il faut accompagner chaque cas d'utilisation des explications (les notes et ce que le système a besoin de faire lors des interactions avec ses acteurs).
- ***Décrire le modèle de cas d'utilisation comme un tout :*** l'élaboration des diagrammes et les descriptions permet d'expliquer le modèle de cas d'utilisation comme un tout et d'examiner les relations que les cas d'utilisation entretiennent entre eux d'une part, et avec les acteurs d'autres part. La mise au point d'un glossaire offre un moyen pratique de rester cohérent lors de la description parallèle de plusieurs cas d'utilisation.

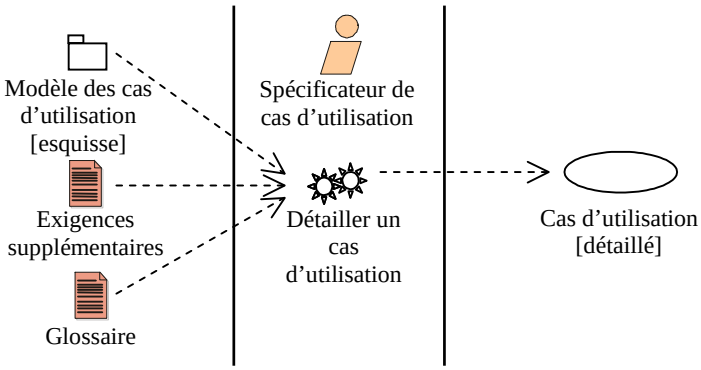
Remarque : A la fin de cette activité, les personnes extérieures à l'équipe de développement (utilisateurs, clients, …) sont invités à déterminer si – tous les besoins fonctionnels ont été exprimés sous forme de cas d'utilisation ; - la séquence d'actions de chaque cas d'utilisation est correcte, complète et compréhensible ; - des cas d'utilisation sans véritable intérêt ont été identifiés (=> les réexaminer).

2. ***Définir un ordre de priorité pour les cas d'utilisation*** : Il faut identifier quelles sont les cas d'utilisation qu'il faut développer (analyser, concevoir, implémenter, tester) dès les premières itérations et ceux qui peuvent attendre les itérations plus tardives. Les résultats sont exprimés sous

la forme d'une vue architecturale du modèle des cas d'utilisation. Elle sert d'entrée à la planification des développements à mener au cours de chaque itération.



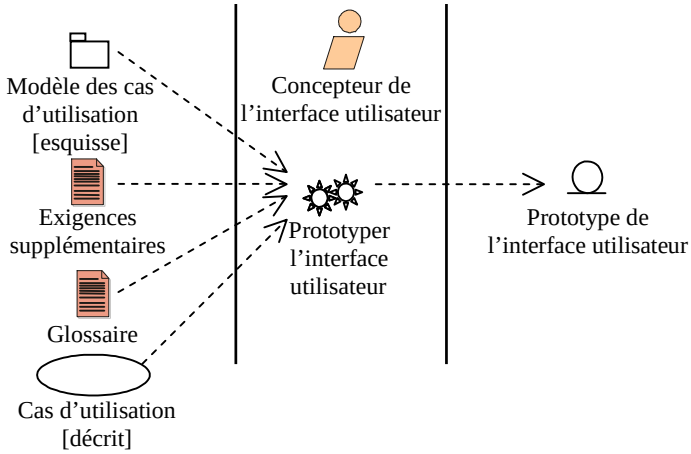
3. ***Détailler un cas d'utilisation*** : il faut décrire étape par étape et détailler la séquence d'actions de chaque cas d'utilisation, en précisant les modalités de début et de fin de chacun. On le fait en collaboration avec les utilisateurs des cas d'utilisation.



- ***Structurer la description d'un cas d'utilisation*** : décrire le cas d'utilisation selon un modèle et aboutir à une description précise, mais parfaitement lisible. Faire ressortir les alternatives si possible.
- ***Formaliser les description des cas d'utilisation*** : décrire le cas d'utilisation avec une technique de modélisation pouvant facilité sa compréhension. On peut décrire un aperçu de la dynamique de réalisation du cas d'utilisation (sorte de diagramme d'états-transition, activité ou interaction) pour faire ressortir les interactions entre des différents éléments participant à la réalisation du cas d'utilisation.

Une description textuelle est très avantageuse et peut se compléter au fur et à mesure.

4. ***Prototyper l'interface utilisateur*** : l'objectif est de concevoir les prototypes d'interface utilisateur permettant à l'utilisateur d'exécuter les cas d'utilisation.



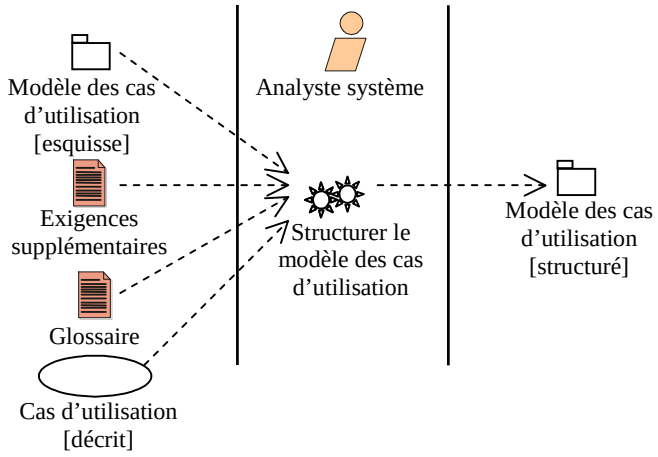
- ***Créer une conception de l'interface utilisateur logique*** : identifier pour chaque acteur et chacun de ses cas d'utilisation les différents éléments (attributs, ceux venant du glossaire,

…) d'interface utilisateur qu'il utilise et manipule (ce qu'il doit fournir au système et ce que le système peut lui renvoyer). Il faut également savoir les liens entre ces éléments, comment ils seront utilisés, manipulés et présentés (leur apparence).

- ***Créer une conception et un prototype de l'interface utilisateur physique :*** ajouter aux éléments ci-dessus identifiés les éléments supplémentaires (conteneurs, fenêtres, …) pour constituer des interfaces complètes. Il faut dessiner manuellement, puis « développer » des prototypes (respect des normes en matière d'IHM). L'effort consacré au prototypage doit être proportionnelle à la valeur de retour attendue (on ne développe des prototypes d'IHM que s'il y a beaucoup à gagner en terme d'utilisabilité : un prototype pour les acteurs les plus importants et esquisser sur papier les autres).

Le prototype permet d'éviter des erreurs pouvant être détectées tard, de révéler les omissions dans les descriptions de cas d'utilisation et d'y remédier avant que le cas d'utilisation n'entre en phase de conception.

5. ***Structurer le modèle de cas d'utilisation*** : l'objectif est de structurer l'ensemble des cas d'utilisation afin de les rendre plus compréhensibles et plus simples à utiliser (dégager les descriptions générales et partagées des fonctionnalités qui pourront être utilisées par des descriptions plus spécifiques).



- ***Identifier les descriptions partagées de fonctionnalités*** : rechercher dans la description de chaque cas d'utilisation les actions complètes ou partielles communes ou partagées par plusieurs cas d'utilisation. Pour éviter les redondances, celles-ci doivent être extraites et décrites sous la forme d'un cas d'utilisation à part susceptible d'être réutilisé par les cas d'utilisation originaux. Cela peut conduire à des relations de généralisation entre les cas identifiés (communs) et les cas originaux.
- ***Identifier les descriptions supplémentaires et facultatives des fonctionnalités*** : faire de même que ci-dessus avec les séquences d'actions supplémentaires et facultatives. Ces actions étendent (souvent sous certaines conditions) les cas d'utilisation originaux.
- ***Identifier d'autres relations entre cas d'utilisation*** : les inclusions (extensions inversées sans conditions) peuvent être aussi identifiées.

La structuration du modèle des cas d'utilisation implique des relations entre le cas. Toutefois, il faut éviter de faire de la décomposition fonctionnelle et trouver le juste milieu (un cas d'utilisation ne doit être ni trop englobant, ni trop fin).

Le modèle des cas d'utilisation comporte essentiellement une description générale de l'ensemble des cas d'utilisation, un ensemble de diagrammes (cas d'utilisation, semblant d'états-transition, activité ou interaction), une description détaillée de chaque cas d'utilisation. On y ajoute aussi en ensemble d'esquisse et de prototype d'interface utilisateur et une spécification des exigences supplémentaires génériques non spécifique à un cas d'utilisation.

V.3. Analyse

V.3.1. Présentation

L'objectif de l'expression des besoins est d'utiliser un langage compréhensible par le client pour décrire les besoins du système afin qu'il (le client) puisse apprécier/confirmer la compréhension de son problème. En outre, les cas d'utilisation constituant un véritable outil d'expression des besoins, il s'agit essentiellement d'utiliser le langage naturel dans la description des cas d'utilisation. Dans ce cas, il faut être très vigilent dans l'utilisation des notations telles que le diagramme d'états-transition, d'activité, d'interaction, … Par conséquent, l'utilisation du langage naturel entraîne une perte de puissance d'expression et il est fort probable que subsistent un grand nombre de problèmes qui auraient dû être précisés par des notations beaucoup plus techniques.

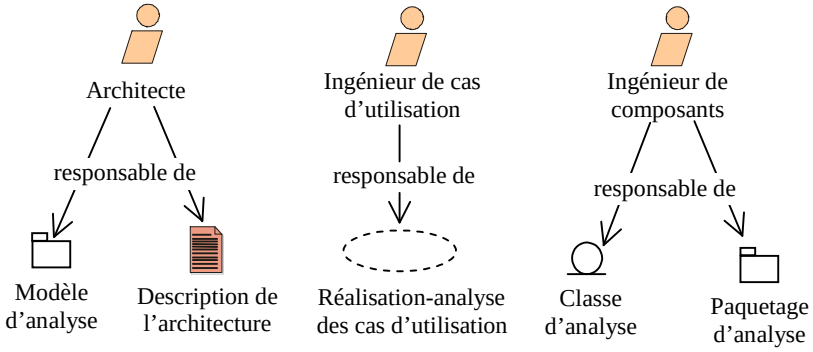
L'objectif de l'analyse est donc de résoudre ces problèmes restés en suspens en procédant à une analyse plus approfondie des besoins et des exigences en utilisant un langage beaucoup plus technique.

En bref, ci-dessous une petite comparaison entre le modèle de cas d'utilisation et le modèle d'analyse.

Modèle des cas d'utilisation	**Modèle d'analyse**
Formulé dans le « langage du client »	Formulé dans le langage du développeur
Vue externe du système	Vue interne du système
Structuré par les cas d'utilisation	Structuré par les classes, les paquetages stéréotypés
Sert principalement de contrat entre le client et les développeurs sur ce que le système doit faire et ne doit pas faire	Sert principalement aux développeurs pour comprendre la conception et l'implémentation
Exprime les caractéristiques du système	Exprime la réalisation des caractéristiques

Le modèle d'analyse constitue le point de convergence des premières itérations de la phase d'élaboration (confère schéma cycle de vie).

Les travailleurs et artefacts impliqués dans l'analyse sont représentés dans la figure ci-dessous.



V.3.2. Artefacts

a) ***Modèle d'analyse*** : il aide à préciser les besoins et exigences (plus grande puissance d'expression et un formalisme plus poussé), permet d'envisager les aspects internes du système sous forme d'objets.

b) ***Classe d'analyse*** : elle représente une abstraction d'une ou de plusieurs classes et/ou sous-systèmes dans la conception du système. Elle se concentre sur les besoins fonctionnels et renvoie la prise en compte des besoins non fonctionnels à la conception et l'implémentation. Les classes d'analyse sont de granularité plus large que leurs équivalents en conception et implémentation. Les attributs d'une classe d'analyse demeurent à un niveau assez élevé et sont souvent reconnaissable à partir du domaine du problème, tandis que ceux des classes de

conception et implémentation présentent des types issus de langages de programmation. Les attributs de classes d'analyse peuvent se transformer en classes au cours de la conception et l'implémentation. Les relations entre classes d'analyse comptent assez peu. Par exemple, une généralisation entre classes d'analyse, pourtant impossible de s'en servir en conception car le langage cible ne la supporte pas. Les classes d'analyse sont définies par trois stéréotypes de classe.

Classes frontières : 

Elle est utilisée pour modéliser l'interaction (réception/présentation des informations et des requêtes échangées) entre le système et ses acteurs. Les classes frontières modélisent les parties du système qui dépendent de ses acteurs en recueillant et clarifiant les exigences pesant sur les frontières du système. Ainsi, la modification d'une interface utilisateur ou d'une interface de communication se contente généralement à la mise à jour de une ou plusieurs classes frontières. Elles représentent souvent des abstractions de fenêtres, formulaire, … Chaque classe frontière doit être liée à au moins un acteur, et vice versa.

Classes entités : 

Elle sert à modéliser les informations persistantes ainsi que le comportement associé. Généralement, les classes entité sont directement dérivées des classes entité du domaine correspondante du modèle de domaine. Toutefois, les classes entités fournissent une représentation des informations utiles aux développeurs lors de la conception et de l'implémentation du système, tandis que les classes du domaine expriment les informations du contexte du système qui ne sont absolument pas gérées par le système. Les classes entités font souvent apparaître une structure de données logique et contribuent à une meilleure compréhension des informations dont dépend le système.

Classes de contrôle : 

Les classes de contrôle représentent la coordination, le séquencement, les transactions et le contrôle d'autres objets. Elles servent souvent à modéliser le contrôle associé à un cas d'utilisation spécifique. Elles permettent aussi de représenter les calculs complexes, tels que la logique métier, ne pouvant être liée à aucune information spécifique à une classe entité. Elles modélisent ainsi la dynamique du système en coordonnant les actions et les séquences principales, et délèguent le travail à d'autres objets (les objets frontières et entités).

c) ***Réalisation-analyse de cas d'utilisation*** : elle décrit la façon dont un cas d'utilisation est réalisé et exécuté en terme de classes d'analyse et d'interaction entre les objets de ces classes. Elle présente une description des diagrammes de classes qui décrivent les classes d'analyse, de diagrammes d'interaction (collaboration par défaut ou si possible de séquence) entre objets d'analyse pour la réalisation du cas d'utilisation. La séquence d'un cas d'utilisation commence lorsqu'un acteur invoque ce cas d'utilisation en envoyant un message quelconque au système. C'est l'objet frontière qui recevra ce message, il envoie à son tour le message à un autre objet, de sorte que les objets concernés dialoguent pour réaliser le cas d'utilisation. Une réalisation de cas d'utilisation doit réaliser de façon adéquate le comportement du cas d'utilisation correspondant dans le modèle de cas d'utilisation, et uniquement ce comportement.

d) ***Paquetage d'analyse*** : il offre un moyen d'organiser les artefacts du modèle d'analyse en parties gérables. Il peut se composer de classes d'analyse et d'autres paquetages d'analyse (hiérarchisation). Les paquetages d'analyse doivent être cohérents (contenir des éléments fortement liés).

e) ***Description de l'architecture (vue du modèle d'analyse)*** : elle contient les classes d'analyse clés, les paquetages d'analyse et les réalisations des cas d'utilisation qui réalisent certaines fonctionnalités importantes et stratégiques (celles impliquant un grand nombre de classes d'analyse et ayant une large couverture et peuvent traverser plusieurs paquetages d'analyse). Cette vue est conserver autant que possible au moment de la conception et de l'implémentation de l'architecture.

V.3.3. Travailleurs

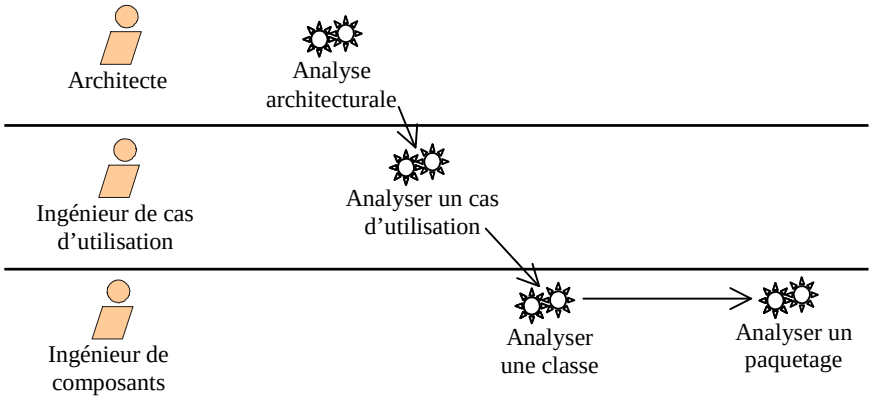
a) ***Architecte*** : il s'assure de l'exactitude et de la cohérence du modèle d'analyse, de l'existence des parties significatives du modèle d'analyse dans l'architecture (vue architecturale du modèle d'analyse). Toutefois, il n'est pas responsable du développement des divers artefacts du modèle d'analyse.

b) ***Ingénieur de cas d'utilisation*** : il est responsable de l'intégrité d'une ou de plusieurs réalisations de cas d'utilisation et doit s'assurer que celles-ci satisfont aux exigences qui leur sont imposées. Il s'assure que toutes les descriptions textuelles et les diagrammes décrivant cette réalisation sont lisibles et remplissent leur mission. Toutefois, il n'est pas responsable des classes d'analyse.

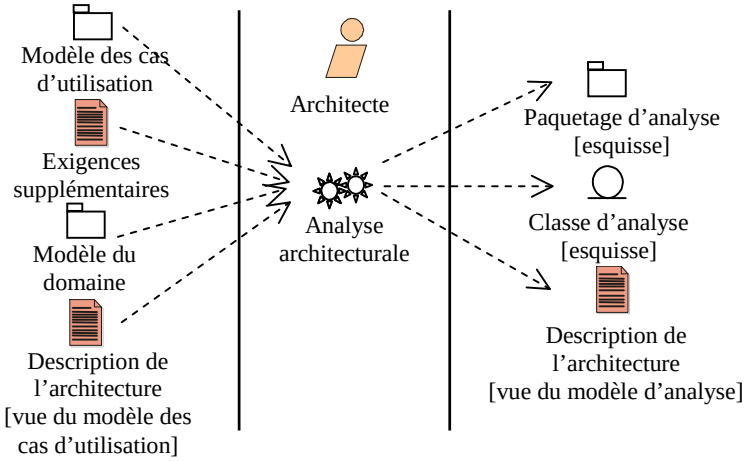
c) ***Ingénieur de composants*** : il définit les responsabilités, attributs, relations et exigences supplémentaires des classes d'analyse et assure leur maintenance. Il le fait en veillant que chaque classe satisfasse aux exigences dans les réalisations des différents cas d'utilisation auxquels elle participe. Il s'assure également que le contenu des paquetages d'analyse et leurs dépendances sont justes et corrects.

V.3.4. Enchaînement d'activités

L'enchaînement d'activités d'analyse peut se résumer par le « diagramme d'activités » ci-dessous.



1. ***Analyse architecturale*** : l'objectif est de délimiter les contours du modèle d'analyse et de l'architecture en identifiant les paquetages d'analyse, les classes d'analyse manifestes et les exigences particulières communes.

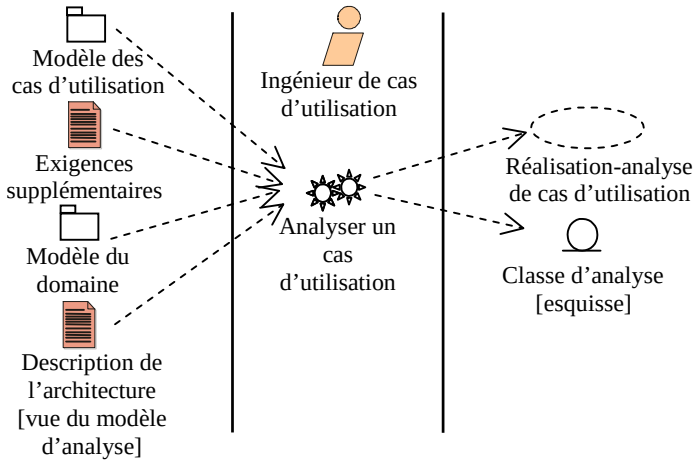


- ***Identification des paquetages d'analyse*** : les fonctionnalités étant exprimées sous forme de cas d'utilisation, il faut attribuer un certains nombre de cas d'utilisation à un paquetage, puis réaliser la fonctionnalité correspondante au sein de ce paquetage. Le regroupement peut se faire selon le processus métier pris en charge, l'acteur ou les relations entre cas

d'utilisation. Selon ces critères, les cas d'utilisation peuvent chevaucher entre paquetages d'analyse (plusieurs paquetages d'analyse se partagent le même élément). L'élément partagé peut donc être extrait de ces paquetages et mis dans un paquetage commun dont tous les autres dépendront. Il faut obtenir des paquetages à forte cohésion interne (les éléments au sein d'un paquetage doivent être liés), mais à faible couplage (minimiser le nombre de relations entre éléments de paquetages différents). En vue de clarifier les dépendances, un découpage en couche du modèle d'analyse est utile, en plaçant les paquetages spécifiques à l'application dans une couche de haut niveau et les paquetages plus généraux dans une couche inférieure.

- ***Identification des classes entités manifestes*** : il faut faire une identification préliminaire des classes entités les plus importantes (celles prenant part à la réalisation des cas d'utilisation et significatives pour l'architecture). Elles peuvent être issues des classes du domaine.
- ***Identification des exigences particulières communes*** : Il faut identifier les exigences qui surgissent au cours de l'analyse afin de pouvoir les gérer dans les activités de conception et d'implémentation. Celles-ci sont en général des restrictions et contraintes de persistance, distribution et concurrence, sécurité, tolérance aux pannes, gestion des transactions, …

2. ***Analyser un cas d'utilisation*** : l'objectif est d'identifier les classes d'analyse dont les objets sont nécessaires à l'exécution du cas d'utilisation, de repartir le comportement du cas d'utilisation entre les objets d'analyse en interaction et de formuler les exigences particulières sur la réalisation du cas d'utilisation. C'est en fait le raffinement des cas d'utilisation.



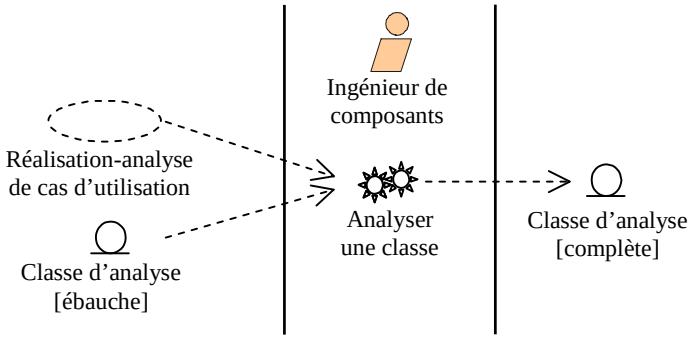
- ***Identification les classes d'analyse*** : identifier les classes entités, frontières et de contrôle nécessaires à la réalisation du cas d'utilisation, ainsi que leurs responsabilités, les attributs et relations. Pour les classes entités, il convient d'étudier en détail la description du cas d'utilisation et identifier les informations nécessaires à sa réalisation, tout en distinguant celles à formuler en tant qu'attributs de celles qu'il vaut mieux relier à des classes frontières ou de contrôle. Les classes frontières centrales pour chaque acteur sont celles qui représentent la fenêtre principale de l'interface utilisateur avec laquelle dialoguera l'acteur. Il faut les minimiser au maximum (réduire le nombre de fenêtre principales avec lesquelles l'utilisateur aura besoin de communiquer) car elles seront des agrégats des classes frontières primitives. Il faut identifier une classe de contrôle responsable de la coordination et du contrôle de la réalisation du cas d'utilisation. Elle peut ne pas exister (si l'acteur assure une part importante du contrôle, le contrôle est encapsulé dans une classe frontière) ou il peut en avoir plusieurs (si le contrôle est complexe, il peut être reparti dans plusieurs classes de contrôle).

  Les classes d'analyse identifiées doivent être réunies dans un diagramme de classes pour montrer les relations existantes dans la réalisation du cas d'utilisation.
- ***Description des interactions entre objets d'analyse*** : il faut utiliser les diagrammes de collaboration pour décrire les interactions entre les instances des acteurs, les objets

d'analyse et leurs liens. Ces diagrammes peuvent être complétés par des description textuelles si plusieurs diagrammes réalisent le cas d'utilisation.

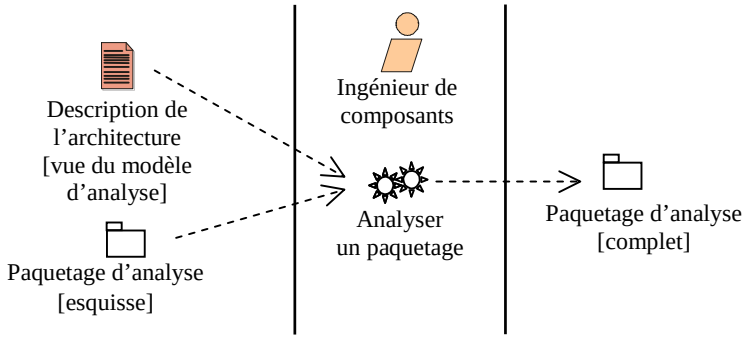
- ***Expression des exigences particulières*** : il faut se référer aux exigences particulières communes identifiées dans l'activité précédente pour formuler celles pesant sur la réalisation du cas d'utilisation.

3. ***Analyser une classe*** : l'objectif est de définir les responsabilités de la classe, d'identifier les attributs et relations de la classe d'analyse et de formuler les exigences particulières pesant sur la réalisation de la classe d'analyse.



- ***Identification des responsabilités*** : elle se fait à partir de tous les rôles que la classe d'analyse joue dans les différentes réalisations de cas d'utilisation. Ces rôles sont recensés en étudiant les diagrammes de classes et d'interaction (collaboration) des diverses réalisations.
- ***Identification des attributs*** : Généralement, les attributs de classes entités sont triviales. Ceux des classes frontières sont des informations manipulées par les acteurs (comme les champs de texte, …), alors que ceux des classes de contrôle représentent des valeurs accumulées ou dérivées au cours de la réalisation du cas d'utilisation.
- ***Identification des associations et des agrégations*** : il faut étudier, affiner et réduire autant que possible les associations entre classes d'analyse. On définit alors les agrégations, multiplicités, les noms des rôles, les associations récursives, les rôles ordonnés, qualifiés et associations n-aires.
- ***Identification des généralisations*** : Dans l'objectif de faciliter la compréhension du modèle d'analyse, il faut représenter les comportements communs et partagés entre classes d'analyse par les généralisations.
- ***Expression des exigences particulières*** : il faut se référer aux exigences particulières communes identifiées dans l'activité précédente pour formuler complètement celles concernant le classe d'analyse.

4. ***Analyser un paquetage*** : l'objectif est de s'assurer que le paquetage d'analyse remplit sa mission (réaliser certains cas d'utilisation) et de décrire les dépendances entre les différents paquetages. Il faut pour cela veiller que le paquetage contient les classes qui conviennent, que les dépendances entre le paquetage et les autres sont celles dont les classes lui sont associées.



Le modèle d'analyse affine et structure les besoins et les exigences. Il comprend les paquetages d'analyse, les classes d'analyse (leurs responsabilités, attributs, relations et exigences particulières), les

réalisations-analyse de cas d'utilisation et la vue architecturale du modèle d'analyse.

Il est possible que le modèle d'analyse ne soit pas conservé tel quel et nécessite quelques modifications lors de la conception et l'implémentation. En effet, la conception exige de prendre en compte la plate-forme d'implémentation (langage de programmation, SE, frameworks, systèmes existants, …).

V.4. Conception

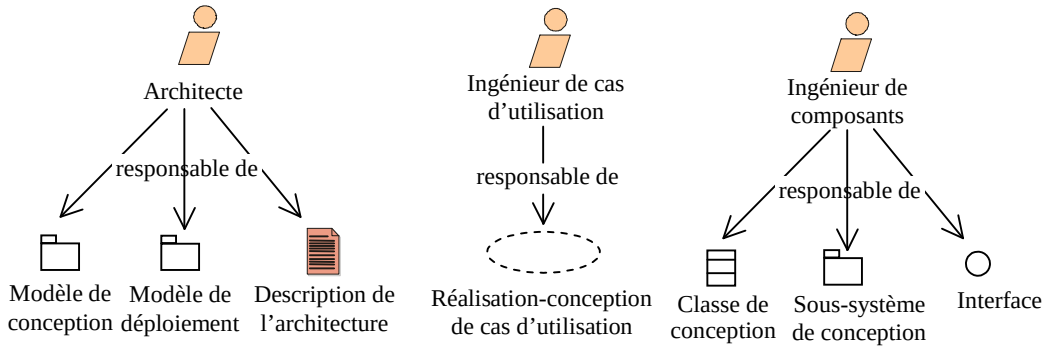
V.4.1. Présentation

Le modèle d'analyse apporte une compréhension détaillée des besoins et des exigences, qui servira de base à la conception. La conception a pour objectifs de constituer une base pour les activités d'implémentation en approfondissant la compréhension des questions concernant les exigences non fonctionnelles (liées au langage de programmation, réutilisation des composants, système d'exploitation, technologie de distribution, concurrence, transaction, …), de décomposer le travail d'implémentation en portion devant faciliter le développement parallèle, …

Modèle d'analyse	Modèle de conception
Evite les problèmes d'implémentation	Plan d'élaboration et de construction de l'implémentation
Générique à la conception	Non générique, mais spécifique à une implémentation
Trois stéréotypes de classe d'analyse (« entité », « de contrôle » et « frontière »)	Autant de stéréotype de classe que le permet le langage d'implémentation
Moins formel	Plus formel
Moins coûteux à développer	Plus coûteux à développer
S'attache dynamiquement peu à la séquence	S'attache dynamiquement plus à la séquence
Risque de ne pas être conservé tout au long du cycle de vie	Doit être conservé tout au long du cycle de vie du logiciel

La conception attire toutes les attentions durant les itérations de la fin de la phase d'élaboration et celles du début de la phase de construction (confère cycle de vie). Elle permet la mise en place d'une architecture saine et stable et crée un plan d'élaboration et de construction du modèle d'implémentation.

Les travailleurs et artefacts impliqués dans la conception.



V.4.2. Artefacts

a) ***Modèle de conception*** : il représente la version abstraite de l'implémentation du système. Il est composé des sous-systèmes de conception, des classes de conception, des réalisations-conception de cas d'utilisation, …

b) ***Classe de conception*** : elle est une abstraction d'une classe équivalente de l'implémentation. Sa spécification utilise le même langage que le langage de

programmation. Il y est spécifié la visibilité des attributs et des opérations de la classe de conception, les relations liant celles-ci, … tous ces éléments ayant une correspondance directe dans l'implémentation. Pour cela, leurs spécifications dépend des possibilités offertes par le langage de programmation cible. Il est cependant conseillé de confier au même développeur la conception et l'implémentation d'une classe.

c) ***Réalisation-conception de cas d'utilisation*** : elle décrit la réalisation et l'exécution d'un cas d'utilisation spécifique par les classes de conception. Elle offre la traçabilité directe avec une réalisation-analyse de cas d'utilisation (avec les classes d'analyse). Elle comporte les diagrammes de classes montrant les classes de conception, les sous-systèmes participants à la réalisation du cas d'utilisation et leurs relations. Elle comporte aussi les diagrammes d'interaction (surtout de séquence) décrivant la réalisation de la séquence d'actions précise du cas d'utilisation en terme d'interaction entre les objets de conception. Ces diagrammes peuvent être accompagnés des descriptions textuelles les éclairant et les complétant avec des exigences d'implémentation.

d) ***Sous-système de conception*** : ils offrent un moyen d'agencer les artefacts du modèle de conception en portions plus gérables. Un sous-système peut être composé de classes de conception, de réalisation-conception de cas d'utilisation, d'interfaces et d'autres sous-systèmes (récursivité). Les éléments qui composent un sous-système doivent être fortement liés et les sous-systèmes doivent être faiblement couplés (minimiser les dépendances).

e) ***Interface*** : elle permet de spécifier les opérations fournies par les classes et les sous-systèmes de conception. Les interfaces offrent un moyen de séparer la spécification des fonctions de leur implémentation. Une classe de conception fournissant une interface doit procurer les méthodes qui réalisent les opérations. De même un sous-système fournissant une interface doit contenir les classes de conception et les autres sous-systèmes qui fournissent cette interface.

f) ***Description de l'architecture (vue du modèle de conception)*** : elle contient la vue architecturale du modèle de conception, c'est-à-dire les éléments les plus significatifs (sur le plan architectural) du modèle de conception. Elle comprend également les processus fournissant les mécanismes de concurrence et de synchronisation du système, y compris les exigences de performance, …

g) ***Modèle de déploiement*** : il décrit la distribution physique du système en montrant la répartition des fonctions sur les différents nœuds de calcul du réseau.

h) ***Description de l'architecture (vue du modèle de déploiement)*** : elle contient la vue architecturale du modèle de déploiement, prenant en compte les nœuds formant la topologie matérielle sur laquelle s'exécute le système, vue centrée sur la distribution, la livraison et l'installation des parties constituant le système.

V.4.3. Travailleurs

a) ***Architecte*** : il s'assure que les modèles de conception et de déploiement sont corrects, cohérents et lisibles. Qu'ils réalisent les fonctions décrites dans le modèle de cas d'utilisation, les exigences supplémentaires et le modèle d'analyse. Il s'assure de l'existence des parties significatives (sur le plan architectural) de ces modèles dans l'architecture. Il n'est pas responsable du développement des artefacts du modèle de conception.

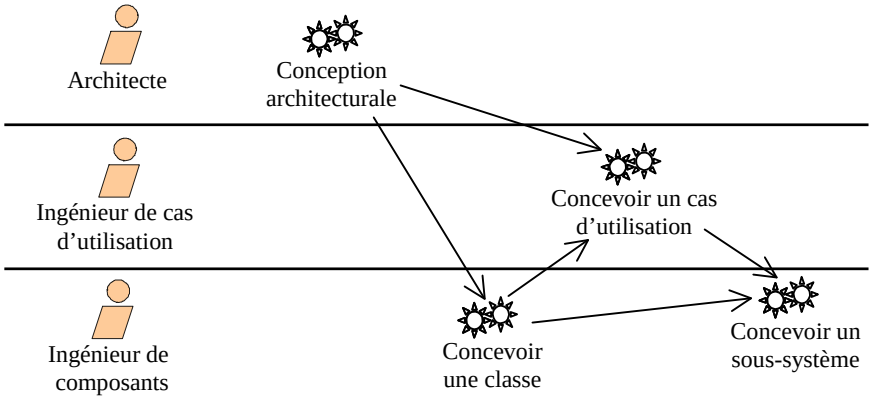
b) ***Ingénieur de cas d'utilisation*** : il rend conforme et lisible les différentes descriptions textuelles et les diagrammes décrivant la réalisation-conception de cas d'utilisation. Il s'assure que ces réalisations satisfont correctement le comportement de la réalisation-analyse de cas d'utilisation, ainsi que le comportement du cas d'utilisation correspondant dans le modèle de cas d'utilisation.

c) ***Ingénieur de composants*** : il définit et actualise les méthodes, attributs, relations et

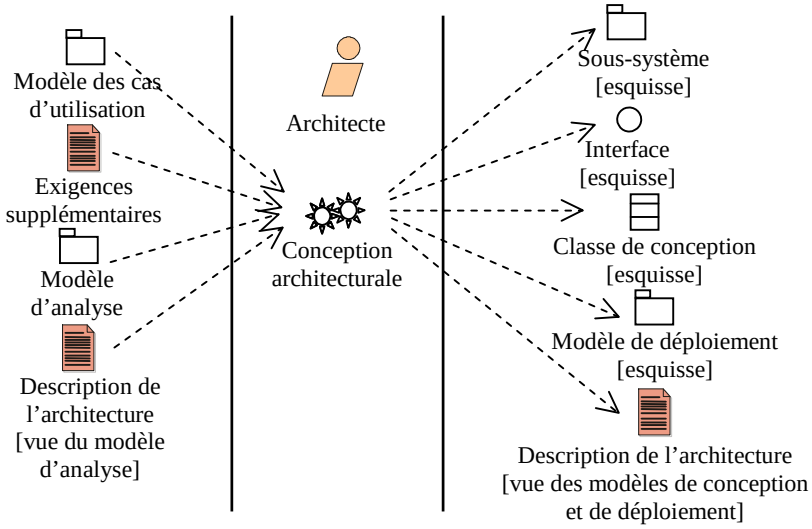
exigences d'implémentation d'une ou de plusieurs classes de conception, et s'assure que chacune satisfait ses exigences formulées dans les réalisations du cas d'utilisation auxquelles elle prend part. il peut également veiller à la cohérence des sous-systèmes en s'assurant que leur contenu est juste et que leurs dépendances vis-à-vis d'autres sous-systèmes sont correctes (fortement liées et faiblement couplées).

V.4.4. Enchaînement d'activités

L'enchaînement d'activités de conception peut se résumer par le « diagramme d'activités » ci-dessous.



1. ***Conception architecturale*** : l'objectif est de tracer les grandes lignes des modèles de conception et de déploiement et de leur vue architecturale, en identifiant les nœuds et leur configuration réseau, les sous-systèmes et leurs interfaces, les classes de conception significatives sur le plan architectural, les mécanismes de conception génériques prenant en charge les exigences particulières communes.

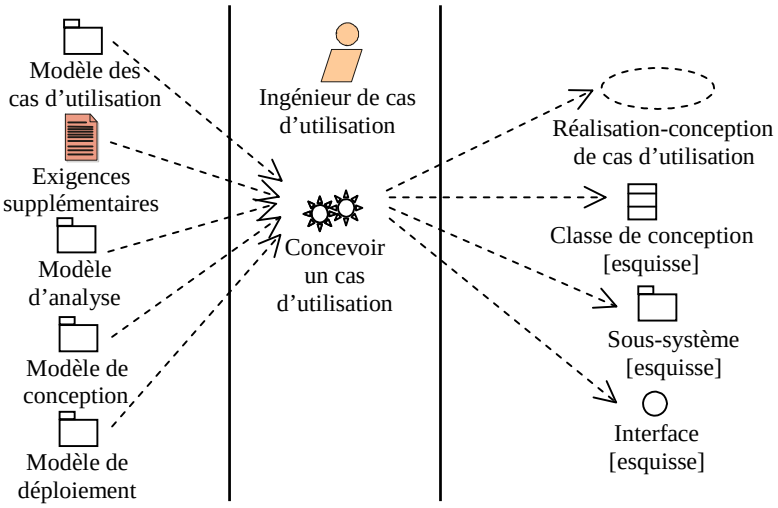


- ***Identification des nœuds et des configurations réseau*** : il est fréquent que les configurations réseau physiques suivent un modèle à trois niveaux (IU, logique métier/applicative, données). L'objectif est de répertorier les nœuds impliqués, leurs capacités en puissance de traitement (mémoire, CPU, …), les types de connexions (caractéristiques, disponibilité, qualité, …) entre les nœuds et les protocoles de communications utilisés, les autres besoins (copie de secours de données, migration de processus, …). Cette connaissance des limites et possibilités des nœuds permettra à l'architecte d'intégrer les technologies ou les services appropriés.
- ***Identification des sous-systèmes et de leurs interfaces*** : l'objectif est de répertorier l'ensemble des sous-systèmes (ceux à développer et ceux à réutiliser). Pour cela, il faut (1) identifier les sous-systèmes d'applications (ceux de la couche spécifique à

l'application et de la couche générale aux applications), (2) identifier les sous-systèmes de middleware et de logiciel système sur lesquels on doit s'appuyer (Attention d'être totalement dépendant de ces sous-systèmes), (3) définir les dépendances entre les sous-systèmes (inter-couches et intra-couches) et (4) identifier les interfaces des sous-systèmes.

- ***Identification des classes de conception significatives sur le plan architectural*** : l'objectif est d'identifier les classes de conception les plus significatives (celles qui participent à la réalisation des cas d'utilisation significatifs) à partir des classes d'analyse (traçabilité). Il faut aussi décider quelles classes seront actives c'est-à-dire celles qui seront (associées) des threads ou des processus et qui seront affectées à des nœuds du modèle de déploiement.
- ***Identification des mécanismes génériques de conception*** : il s'agit d'étudier les exigences particulières identifiées sur les réalisations-analyse de cas d'utilisation et les classes d'analyse et de décider de la manière dont elles seront prises en charge en fonction des technologies d'implémentation disponibles. Cela peut aboutir à des classes de conception et sous-systèmes additionnels (par exemple pour la détection et récupération des erreurs).

2. ***Concevoir un cas d'utilisation*** : cette activité a pour objectif d'identifier les classes de conception et/ou sous-systèmes nécessaires à la réalisation du cas d'utilisation, distribuer le comportement du cas d'utilisation entre les objets de conception en interaction et/ou les sous-systèmes participants, de définir les exigences pesant sur les opérations des classes de conception et/ou des sous-systèmes et leurs interfaces et de formuler les exigences d'implémentation pour le cas d'utilisation en question.

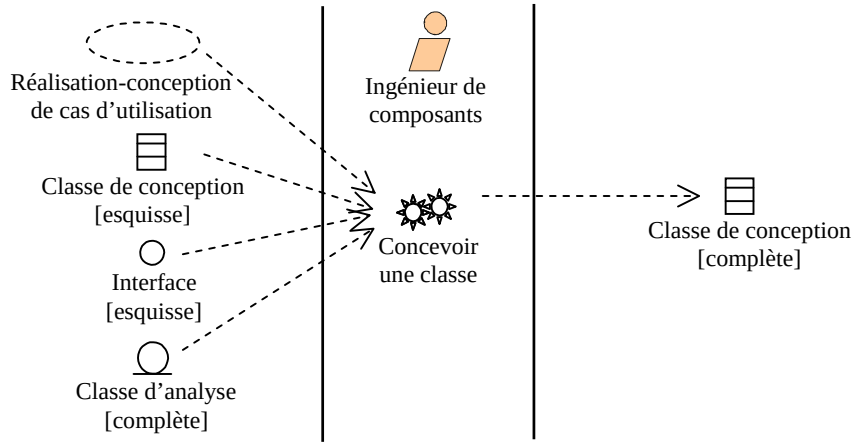


- ***Identification des classes de conception participantes*** : il faut étudier les classes d'analyse prenant part à la réalisation-analyse de cas d'utilisation pour en identifier les classes de conception correspondantes, étudier les exigences particulières et identifier aussi les classes de conception réalisant ces exigences.
  L'ensemble des classes de conception doit être réuni dans un diagramme de classes qui exposera les relations entre ces classes.
- ***Description des interactions entre objets de conception*** : il faut décrire les interactions entre objets de l'esquisse de classes identifié. On utilise le diagramme de séquence pour faire ressortir les instances des acteurs participants, les objets de conception et leurs échanges de message. Chaque classe de conception doit avoir au moins un objet de conception participant au diagramme de séquence.
- ***Identification des sous-systèmes participants et de leurs interfaces*** : l'utilisation de l'approche descendante encourage la décomposition du système en sous-système avant le dévoilement du contenu de chaque sous-système. Dans ce cas, les sous-systèmes (les classes de conception et leur regroupement en sous-systèmes) doivent être identifiés à

partir de l'étude des classes d'analyse et des exigences particulières de la réalisation-analyse de cas d'utilisation.

- ***Description des interactions entre sous-systèmes*** : les interactions sont décrites à partir des diagrammes de séquences matérialisant cette fois-ci des échanges de messages entre sous-systèmes (les lignes de vie représentent les sous-systèmes et non les objets de conception).
- ***Formulation des exigences d'implémentation*** : il faut formuler toutes les exigences non fonctionnelles identifiées, mais qui devront être gérées dans l'implémentation.

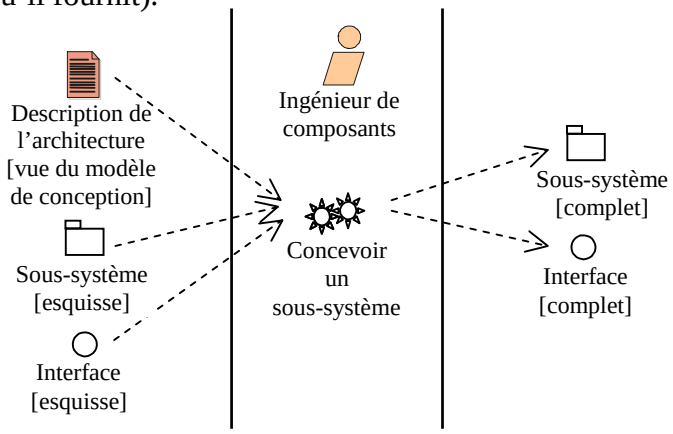
3. ***Concevoir une classe*** : l'objectif est de créer une classe de conception remplissant son rôle dans les réalisations de cas d'utilisation.



- ***Ebauche de la classe de conception*** : il faut esquisser quelques classes de conception à partir des classes d'analyse et des interfaces déjà identifiées. Pour les classes frontières, les classes de conception dépendront de la technologie d'interface spécifique utilisée. Les classes de conception correspondantes aux classes entité imposent de recourir à une technologie de BD spécifique. Pour les classes de contrôle, il faut tenir compte des aspects non fonctionnels (distribution, transaction, …). Dans tous les cas, les classes de conception doivent se voir attribuer des dépendances de traçabilité par rapport aux classes d'analyse correspondantes.
- ***Identification des opérations*** : il faut, en fonction des responsabilités de chaque classe, des exigences particulières à la classe, de ou des interfaces fournies par la classe, des réalisations-conception de cas d'utilisation auxquels participent la classe identifier les opérations qui doivent être fournies par la classe et les décrire dans la syntaxe du langage de programmation cible (visibilité, paramètres, …).
- ***Identification des attributs*** : les attributs sont souvent impliqués et nécessaires aux opérations de la classe. Ils doivent être décrits dans la syntaxe du langage cible. Ils sont en général les attributs des classes d'analyse auxquelles la classe de conception remonte (il peut avoir naissance d'autres).
- ***Identification des associations et des agrégations*** : il faut étudier les échanges de messages (interactions : diagramme de séquence) entre les objets des classes de conception pour déterminer les associations entre classes de conception.
- ***Identification des généralisations*** : Comme la sémantique du langage de programmation cible doit être utilisée, si ce dernier ne supporte pas la généralisation, on peut utiliser les associations et/ou agrégations pour assurer la délégation.
- ***Description des méthodes*** : on peut utiliser les méthodes pour spécifier la façon dont les opérations seront réalisées. Il faut alors définir les algorithmes (ou guides) en langage naturel, pseudo-code ou autre.
- ***Description des états*** : Dans le cas où le comportement (à la réception d'un message) de certains objets dépend de leur état, il est intéressant d'utiliser un diagramme d'états-transitions pour décrire les différentes transitions de ces objets.

- ***Gestion des exigences particulières*** : il faut gérer les exigences qui n'ont pas été prises en compte dans les étapes précédentes, surtout en remontant à classe d'analyse à laquelle dérive la classe de conception.

4. ***Concevoir un sous-système*** : l'objectif est de s'assurer que le sous-système est aussi indépendant que possible des autres sous-systèmes, qu'il fournit les interfaces voulues, qu'il remplit sa mission (offre une réalisation satisfaisante des opérations définies par les interfaces qu'il fournit).



- ***Actualisation des dépendances entre sous-systèmes*** : il faut veiller à ce que le sous-système dépende modérément des interfaces fournies d'autres sous-systèmes que de dépendre des sous-systèmes eux-mêmes (dépendances avec les éléments internes). Dans la mesure du possible, on peut envisager de déplacer des classes qui seraient trop dépendantes d'autres sous-systèmes de ces sous-systèmes.
- ***Actualisation des interfaces fournies par le sous-système*** : il faut s'assurer et actualiser si possible les interfaces du sous-système si celles-ci ne prennent pas en charge tous les rôles du sous-système. Les raffinements sont apportés par l'ingénieur se composants à ce qu'à fait l'architecte.
- ***Actualisation du contenu d'un sous-système*** : le contenu du sous-système peut être amélioré par l'ingénieur de composants au fur et à mesure qu'évolue le modèle de conception.

En définitive, le modèle de conception comprend les sous-systèmes de conception et leurs dépendances, interfaces et contenu, les classes de conception (avec attributs, opérations, relations, exigences d'implémentation), les réalisations-conception de cas d'utilisation, la vue architecturale du modèle. La conception se traduit également par le modèle de déploiement, qui décrit toutes les configurations réseau (les nœuds, leurs caractéristiques, les connexions, …) sur lesquelles le système doit être distribué.

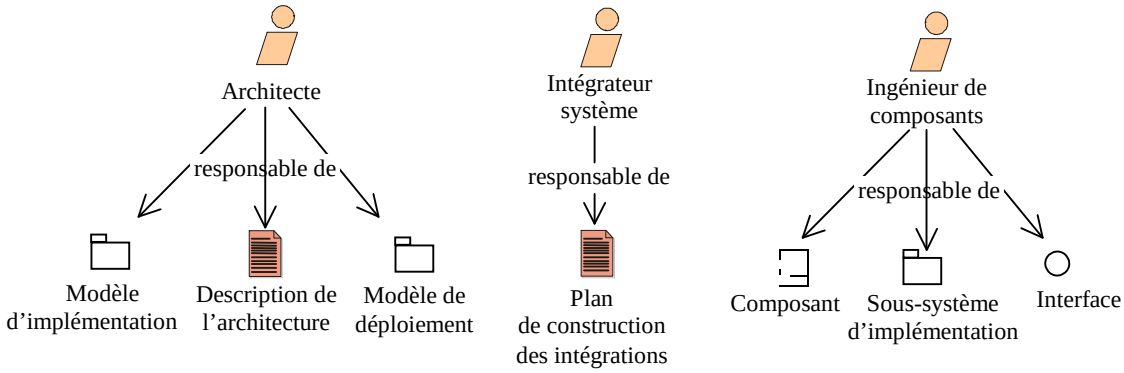
V.5. Implémentation

V.5.1. Présentation

Les objectifs de l'implémentation sont d'implémenter (sous forme de composants – code source, scripts, binaires, …) les classes et sous-systèmes identifiés au cours de la conception, de tester les composants unité par unité et les intégrer en les compilant et en les reliant les uns aux autres, de distribuer le système en faisant correspondre les composants exécutables à des nœuds du modèle de déploiement, de planifier les intégrations nécessaires à chaque itération.

Etant au cœur des itérations de construction, l'implémentation intervient aussi lors de l'élaboration pour la création de l'architecture exécutable de référence et au cours de la transition pour la prise en compte des anomalies apparues tardivement (cf. cycle de vie).

Les travailleurs et artefacts impliqués dans l'implémentation sont les suivants :



V.5.2. Artefacts

a) ***Modèle d'implémentation*** : il décrit la façon dont les éléments du modèle de conception sont implémentés. Il présente aussi l'agencement des composants en fonction des mécanismes de structuration et de modularisation disponibles dans l'environnement d'implémentation et le(s) langage(s) d'implémentation.
b) ***Composant*** : c'est un empaquetage d'éléments de modèle. Il peut être stéréotypé « exécutable » (un programme pouvant être exécuté sur un nœud), « fichier » (contenant le code source ou des données), « bibliothèque » , « table », « document ». Un composant peut implémenter plusieurs éléments de modèle et entretient des relations de traçabilité avec les éléments de modèle qu'il implémente (en fournissant les mêmes interfaces). Il peut aussi exister des dépendances de compilation entre composants.
c) ***Sous-système d'implémentation*** : il offre un moyen de répartir les artefacts du modèle d'implémentation en entités plus gérables. Il peut être constitué de composants, d'interfaces, et d'autres sous-systèmes. Le mécanisme de regroupement en sous-système dépend de l'environnement d'implémentation (paquetage en java, projet en .Net, …). Il doit avoir une traçabilité un-à-un avec le sous-système de conception correspondant.
d) ***Interface*** : Un composant qui fournit une interface doit implémenter toutes les opérations définies par cette interface. De même pour les sous-systèmes, mais ça peut être un composant du sous-système ou un autre sous-système du sous-système qui fournit l'interface.
e) ***Description de l'architecture (vue du modèle d'implémentation)*** : elle contient une vue architecturale du modèle d'implémentation c'est-à-dire les artefacts les plus significatifs sur le plan architectural (composants clés remontant à des classes de conception significatives).
f) ***Plan de construction des intégrations*** : il décrit la séquence de constructions requise dans une itération. Pour chaque construction, il décrit les fonctions (cas d'utilisation) devant être implémentées par la construction, les parties du modèle d'implémentation affectées par la construction.

V.5.3. Travailleurs

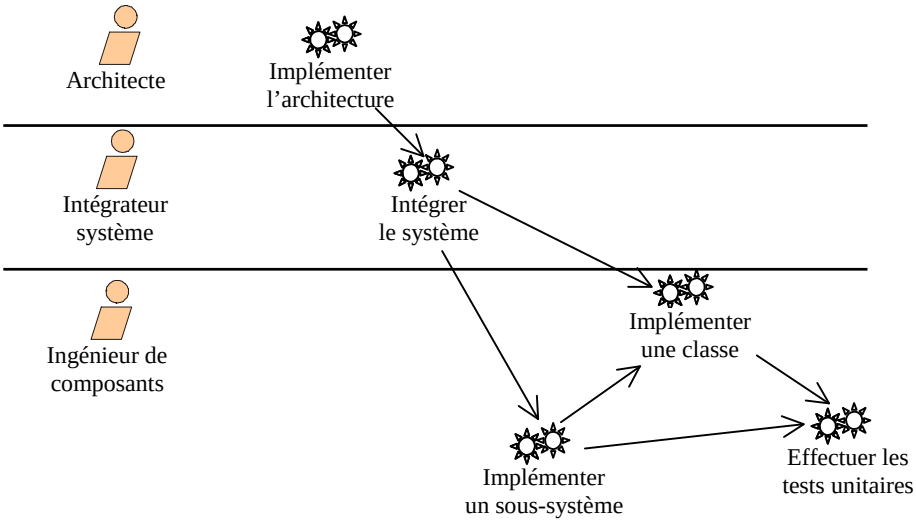
a) ***Architecte*** : il s'assure de l'exactitude, de la cohérence et de la lisibilité globale du modèle d'implémentation. Il veille à ce que le modèle d'implémentation implémente les fonctions décrites par le modèle de conception, ainsi que toute exigence supplémentaire. Il s'assure de l'existence des parties significatives (sur le plan architectural) du modèle d'implémentation dans l'architecture. Il n'est pas responsable du développement des divers artefacts du modèle d'implémentation.
b) ***Ingénieur de composants*** : il définit et actualise le code source d'un ou de plusieurs composants, en s'assurant que chaque composant implémente les fonctions telles qu'elles ont été spécifiées dans la conception. Il veille à l'intégrité d'un ou de plusieurs sous-

systèmes d'implémentation (traçabilité un-à-un avec les sous-systèmes de conception). Il s'assure que le contenu des sous-systèmes d'implémentation est juste et que leur dépendance vis-à-vis d'autres sous-systèmes est exacte et qu'ils implémentent correctement les interfaces fournies.

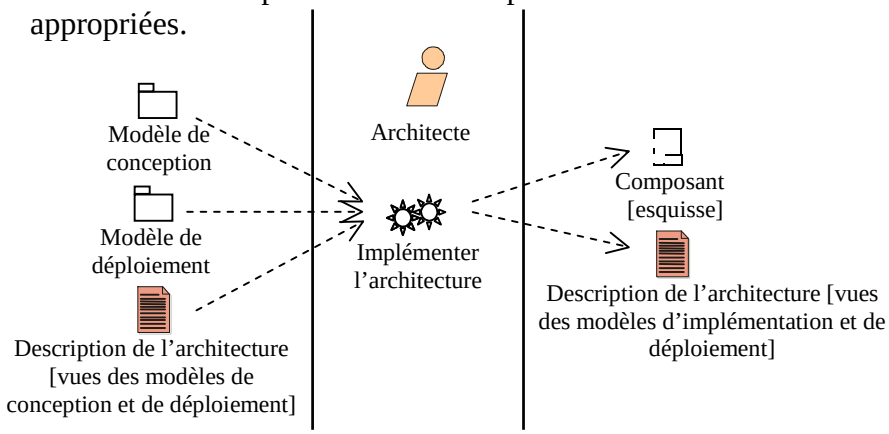
c) ***Intégrateur système*** : il planifie la séquence de constructions nécessaires à chaque itération et l'intégration de chaque construction une fois ses diverses parties implémentées.

V.5.4. Enchaînement d'activités

L'enchaînement d'activités d'implémentation est le suivant :

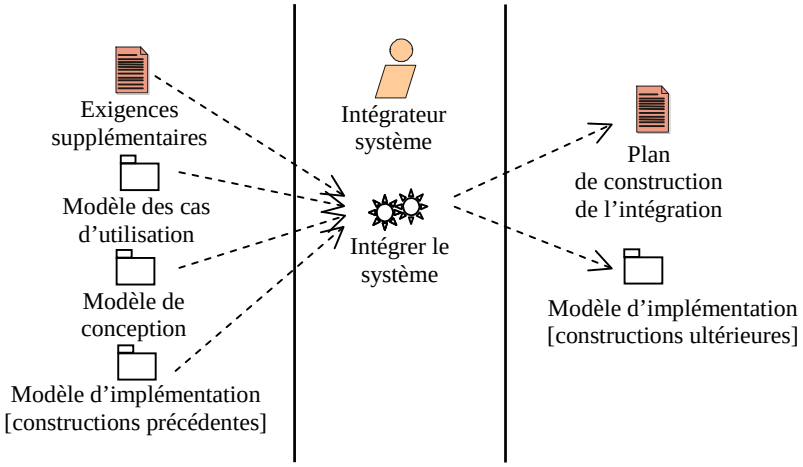


1. ***Implémenter l'architecture*** : l'objectif est de délimiter le modèle d'implémentation à son architecture en identifiant les composants significatifs sur le plan architectural, puis en assurant la correspondance des composants avec les nœuds dans les configurations réseau appropriées.



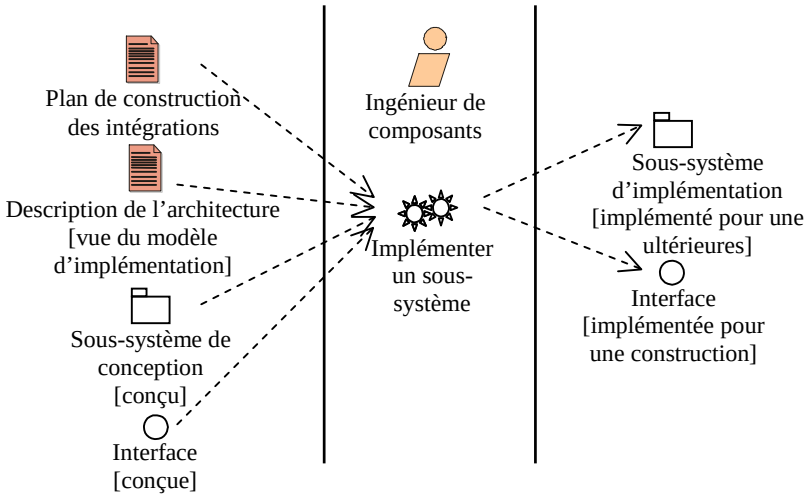
- ***Identification des composants significatifs sur le plan architectural*** : il faut identifier les composants clés sur le plan architectural. Le regroupement de l'implémentation des classes dans les fichiers sources permet d'identifier aisément certains composants fichiers. Il faut identifier juste les composants significatifs (une simple esquisse de composants significatifs).
- ***Identification des composants exécutables et correspondance avec les nœuds*** : les classes actives déterminées pendant la conception doivent se voir affecter chacune un composant exécutable devant être déployé sur un nœud. On peut détecter d'autres composants fichiers et/ou binaires nécessaires à la création de composants exécutables.

2. ***Intégrer le système*** : l'objectif est la création d'un plan de construction de l'intégration et l'intégration de chaque construction avant qu'elle ne soit soumise au test d'intégration.

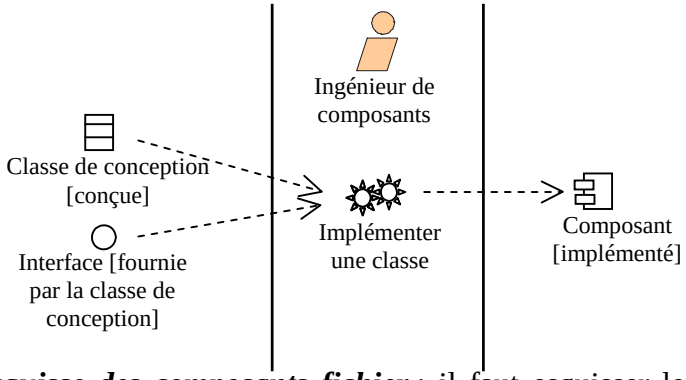


- ***Planification d'une construction ultérieure*** : il faut planifier le contenu d'une construction, partant d'une construction existante ou de rien. Une construction doit ajouter des fonctions par rapport à la construction précédente à travers l'implémentation des cas d'utilisation. Elle ne doit pas comprendre trop de composants nouveaux ou affinés. Les premières constructions doivent partir des couches inférieures (les couches middlewares et logiciel systèmes), tandis que les suivantes remontent en direction de la couche générale aux applications, puis à celle spécifique à l'application. Il faut bien identifier les véritables besoins devant être implémentés par une construction et laisser les autres besoins aux constructions ultérieures.
- ***Intégration d'une construction*** : il faut recueillir les versions appropriées des sous-systèmes d'implémentation et des composants, les compiler et les lier en une construction. Cette dernière doit être ensuite soumise aux tests.

3. ***Implémenter un sous-système*** : l'objectif est de s'assurer que le sous-système remplit son rôle c'est-à-dire les besoins à implémenter le sont correctement par les composants de ce sous-système. Le contenu du sous-système peut connaître des améliorations par rapport à ce qui a été définit par l'architecte, en fonction de l'évolution du modèle d'implémentation. Les dépendances de traçabilité doivent existées entre les sous-systèmes de conception et ceux d'implémentation. Chaque interface fournie par le sous-système de conception correspondant et nécessaire à la construction en cours doit être fournit par le sous-système d'implémentation.

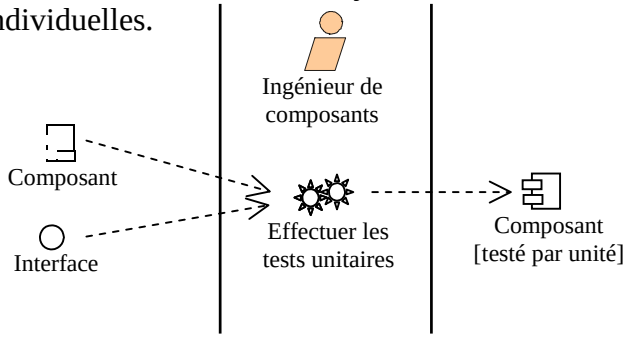


4. ***Implémenter une classe*** : l'objectif est d'implémenter une classe de conception dans un composant fichier.



- ***Esquisse des composants fichier*** : il faut esquisser le composant fichier qui contiendra le code source. On peut implémenter plusieurs classes de conception dans le même composant fichier. Toutefois, les conventions du langage utilisé aura un impact sur l'élaboration des composants fichiers.
- ***Génération de code à partir d'une classe de conception*** : il faut générer le code source à partir de la classe de conception et des relations qu'elles entretiennent. En effet, une grande partie des éléments (attributs, opérations, …) de la classe de conception sont exprimés dans la syntaxe du langage de programmation. Toutefois, seule la signature des opérations est générés et elles doivent être implémentées. Cela dépend toujours du langage de programmation cible.
- ***Implémentation des opérations*** : chaque opération définie par la classe de conception doit être implémentée, à moins qu'elle ne soit abstraite et implémentée par les descendants de la classe. Cela suppose de choisir un algorithme approprié même s'il peut être déjà spécifié en langage naturel ou pseudo-code pendant le conception. La méthode est l'implémentation d'une opération de la classe.
- ***Faire en sorte que les composants fournissent les interfaces appropriées*** : le composant résultant de l'implémentation de la classe doit fournir les mêmes interfaces que la ou les classes de conception qu'il implémente.

5. ***Effectuer les tests unitaires*** : l'objectif est de tester les composants implémentés comme unités individuelles.



- ***Réalisation des tests des spécifications*** : il faut vérifier le comportement (observable de l'extérieur) du composant sans s'intéresser à la façon dont il est implémenté (test boîte noir). Pour cela, on examine les sorties que le composant produit à partir des entrées de données et en fonction de l'état de départ spécifique. Généralement, la gamme des combinaisons entrées, sorties et états de départ est souvent considérable (difficile de les tester un par un). On repartit alors ces entrées, sorties et états de départ en classes d'équivalence. Une **classe d'équivalence** est un ensemble de valeurs d'entrée, de sortie et d'état de départ pour lesquelles un composant est censé se comporter de manière identique. Il faut alors tester le composant pour chaque combinaison de classes d'équivalence.
- ***Réaliser les tests de structure*** : il faut vérifier le fonctionnement interne du composant (test boîte blanche). Tout le code source doit être testé (chaque instruction doit être exécutée au moins une fois). Il faut surtout veiller à tester les chemins les plus intéressants à travers le code : ceux qui sont le plus couramment empruntés, les plus stratégiques et ceux qui sont

associés à un niveau de risque élevé.

Le modèle d'implémentation doit donc contenir les sous-systèmes d'implémentation et leurs dépendances, interface et contenus, les composants (testés un à un) et leurs dépendances, la vue architecturale du modèle d'implémentation (les éléments significatifs de ce modèle sur le plan architectural).

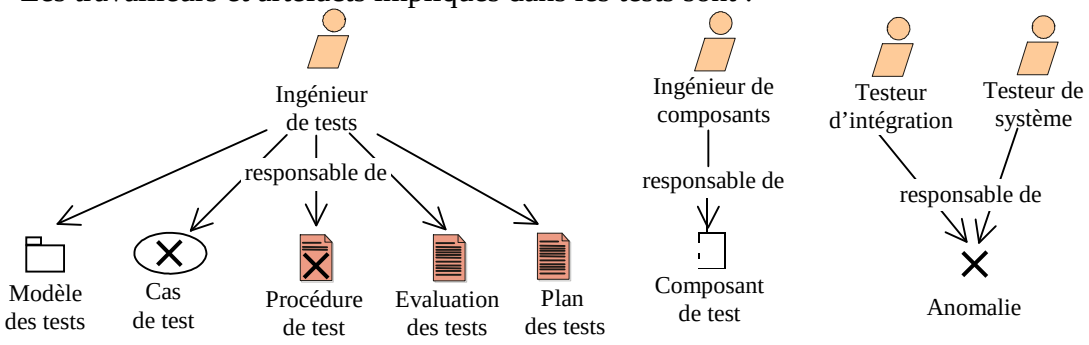
V.6. Tests

V.6.1. Présentation

L'objectif des tests est de vérifier les résultats de l'implémentation en testant chaque construction (internes, intermédiaires et versions finales livrables à l'extérieur). Il faut alors planifier les tests nécessaires pour chaque itération, concevoir et implémenter les tests en créant les cas de test spécifiant ce qui doit être testé, si possible développer les composants de test exécutables destinés à automatiser les tests. Les constructions faisant apparaître des anomalies doivent être retestées, éventuellement renvoyées à d'autres enchaînements d'activités (conception, implémentation) afin que les anomalies puissent être corrigées.

Les tests mobilisent l'attention pendant la phase d'élaboration lors de l'architecture de référence et pendant la construction lors de l'implémentation du système. Pendant le phase de transition, l'attention se porte principalement sur la correction des anomalies détectées au cours des premières utilisations et des tests de non-régression.

Les travailleurs et artefacts impliqués dans les tests sont :



V.6.2. Artefacts

a) ***Modèle de test*** : il décrit les conditions et les modalités de test d'intégration et système des composants exécutables du modèle d'implémentation.

b) ***Cas de test*** : il spécifie une manière de tester le système, en précisant ce qui doit être testé, avec quelles entrées, le(s) résultat(s) escompté(s) et sous quelles conditions. Certains cas de test peuvent être très proches et ne différer que par une seule valeur d'entrée ou de résultat (pour vérifier différentes possibilités – scénario de cas d'utilisation par exemple). Les cas de test peuvent concerner les cas d'utilisation (boîte noire : vérifier que le cas d'utilisation réalise effectivement ses fonctions), l'installation du système (vérifier que le système peut être installé sur la plate-forme du client et qu'il fonctionne bien une fois installé), la configuration (vérifier que le système fonctionne correctement dans diverses configurations – réseau, …), robustesse (provoquer l'échec du système en le faisant fonctionner dans des conditions anormales afin de mettre à jour ses points faibles), …

c) ***Procédure de test*** : elle spécifie les instructions pour l'exécution d'un cas de test. Il est utile de réutiliser une même procédure de test pour plusieurs cas de test et de réutiliser plusieurs procédures pour un seul cas de test.

d) ***Composant de test*** : il automatise tout ou parties d'une ou de plusieurs procédures de test. Il est utilisé pour tester les composants du modèle d'implémentation en fournissant les entrées

de test, en contrôlant et en surveillant l'exécution du composant testé et, éventuellement en rendant compte des résultats du test. Si plusieurs composants de test exercent des interactions complexes entre eux ou avec des composants du modèle d'implémentation (ceux testés), ils peuvent nécessiter un modèle de conception de test pour les modéliser afin d'en fournir une vue de haut niveau.

e) ***Plan de test*** : il décrit les stratégies de test, leur calendrier et les ressources à utiliser. Il définit le type et les objectifs des tests à effectuer pour chaque itération.

f) ***Anomalie*** : elle désigne un symptôme d'échec ou un problème découvert. Elle désigne tout ce que les développeurs doivent considérer comme problème (symptôme) à suivre et à résoudre.

g) ***Évaluation des tests*** : elle présente l'évaluation des résultats de l'ensemble des tests. Elle précise la couverture des cas de test, la couverture du code et l'état des anomalies.

V.6.3. Travailleurs

a) ***Ingénieur de test*** : il doit veiller à ce que le modèle de test remplisse son rôle. Il est chargé de planifier les tests en leur fixant des objectifs et en définissant le calendrier de leur déroulement, … Toutefois, il n'exécute pas les tests, mais s'occupe de leur préparation et de leur évaluation.

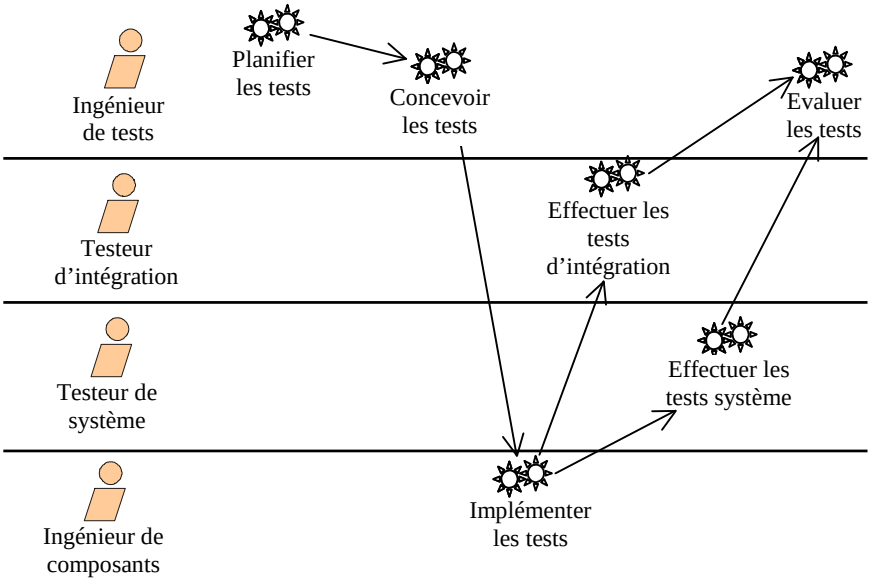
b) ***Ingénieur de composants*** : il est responsable des composants de test qui automatisent certaines procédures de test.

c) ***Testeur d'intégration*** : il effectue les tests d'intégration nécessaires pour chaque construction produite au cours de l'implémentation. Ces tests permettent de s'assurer que les composants intégrés à une construction fonctionnent correctement ensemble.

d) ***Testeur système*** : il est chargé de vérifier les interactions entre le système et les acteurs. Pour cela, il doit effectuer les tests exigés pour le résultat de chaque itération. D'autres types de test (voir les cas de test) peuvent s'appliquer à l'ensemble du système.

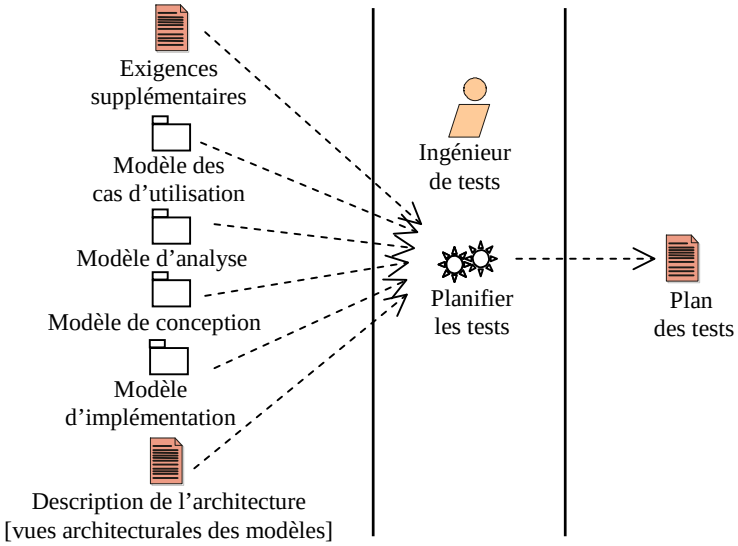
V.6.4. Enchaînement d'activités

L'enchaînement d'activités de test est le suivant :

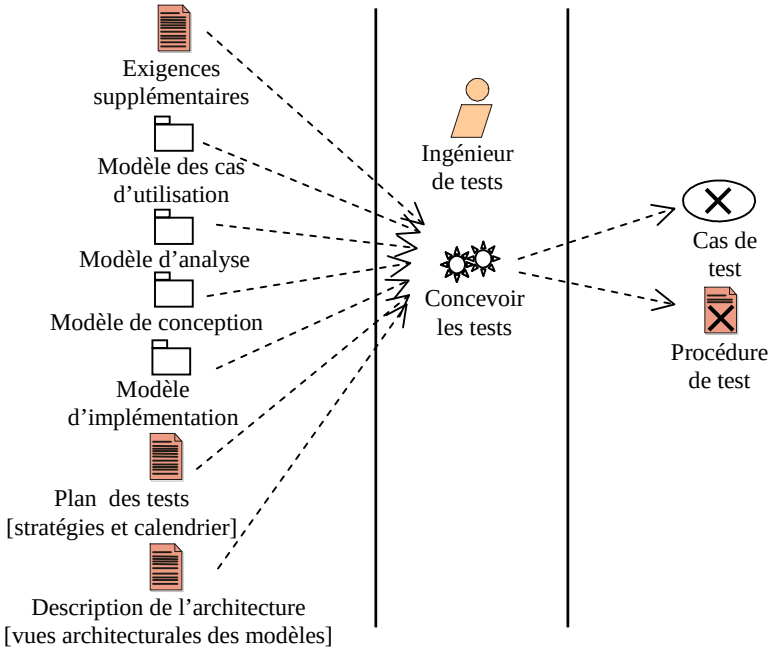


1. ***Planifier les tests*** : l'objectif est de planifier les tests en décrivant la stratégie de test, en estimant les exigences imposées par les tests (ressources humaines et systèmes nécessaires, …), en fixant le calendrier des tests. Les différents modèles conçus et les exigences supplémentaires aident les ingénieurs de tests à définir la stratégie pour chaque itération (type de test, quand et comment les exécuter, comment déterminer s'ils ont été concluants,

…). Toutefois, les cas et procédures de test prioritaires sont ceux destinés à tester les cas d'utilisation les plus importants et les exigences associées aux risques les plus sérieux.



2. ***Concevoir les tests*** : l'objectif est d'identifier et de décrire les cas de test pour chaque itération, d'identifier et de structurer les procédures de test indiquant les conditions de réalisation des cas de test.

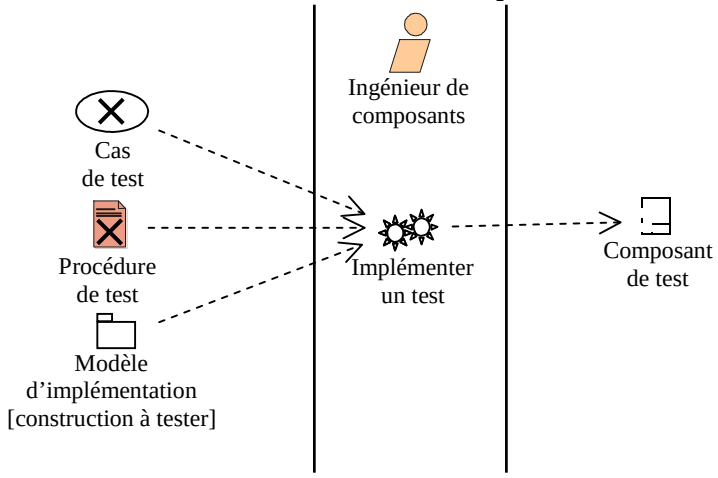


- ***Conception des cas de test d'intégration*** : les cas de test d'intégration servent à vérifier l'interaction des composants les uns avec les autres après leur intégration au sein d'une construction. Pour cela, il faut créer un ensemble de cas de test permettant d'atteindre l'objectif du plan des tests. On considère les diagrammes d'interaction et on recherche les combinaisons d'entrées, de sorties et d'état de départ du système formant les réalisations de cas d'utilisation mettant à contribution les classes (=> les composants) figurant dans les diagrammes.
- ***Conception des cas de test système*** : les tests système visent à vérifier que le système fonctionne correctement dans son ensemble. Les cas de test système doivent privilégier les combinaisons de cas d'utilisation susceptibles d'être exécutés en parallèle, de s'influencer les uns les autres, impliquant de nombreux processus, utilisant fréquemment les ressources systèmes (mémoire, processeur, …). Il faut concevoir les cas de test système sous différentes conditions (différentes configurations matérielles, diverses tailles de BD, …),

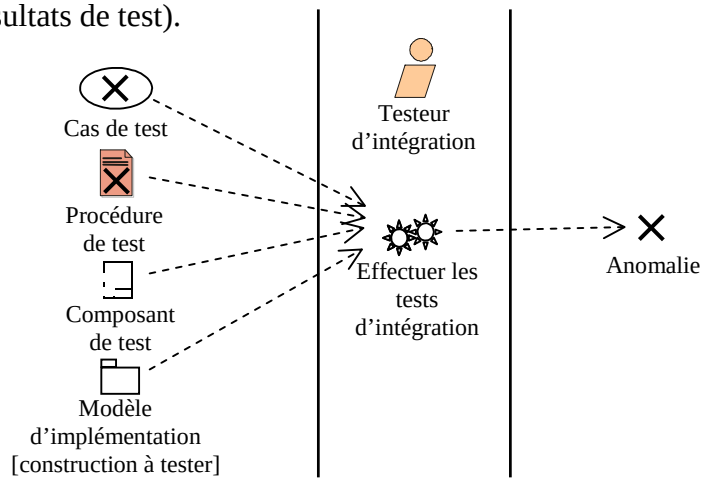
inspiré des exigences particulières.

- ***Conception des cas de test de non-régression*** : les cas de test de non-régression peuvent être certains de ceux issus des constructions précédentes. Les cas de test doivent être assez souples pour réagir correctement aux changements affectant le système à tester.
- ***Identification et structuration des procédures de test*** : il faut identifier et structurer les instructions pour l'exécution des cas de test.

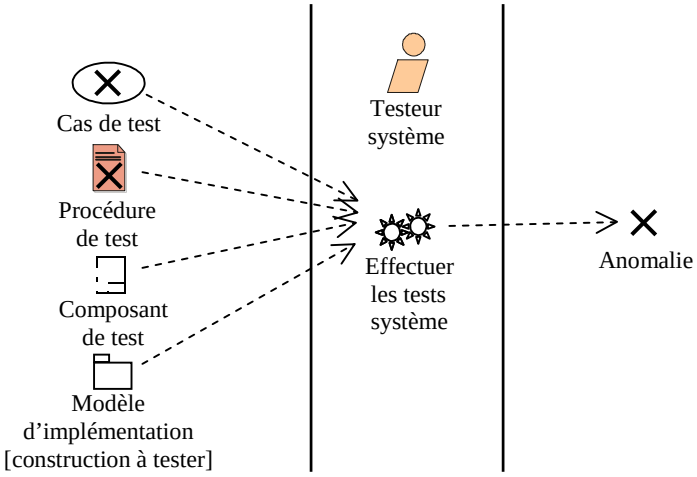
3. ***Implémenter les tests*** : l'objectif est d'automatiser les procédures de test en créant des composants de test. Ces composants peuvent provenir d'un outil d'automatisation des tests (on spécifie les actions de la procédure de test et l'outil génère un composant de test) ou développé explicitement (les actions de la procédure de test serviront de spécification du composant à développer). Le composant de test obtenu doit fournir sous une forme agréable (mode de visualisation) les résultats de test afin de faciliter leur interprétation. Il n'est pas possible d'automatiser entièrement toutes les procédures de test.



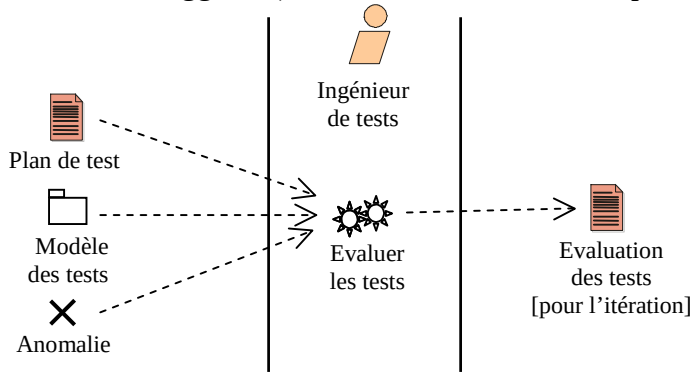
4. ***Effectuer les tests d'intégration*** : l'objectif est de soumettre au test chaque construction crée au cours d'une itération. Il faut en particulier exécuter manuellement les procédures de test pour chaque cas de test ou exécuter les composants de test qui automatisent ces procédures. Ensuite, comparer les résultats des tests aux résultats escomptés et examiner les résultats des tests qui s'éloignent de ceux prévus. Il faut également rendre compte des anomalies détectées à l'ingénieur de composants concerné et à l'ingénieur de test (pour évaluation des résultats de test).



5. ***Effectuer les tests système*** : on le fait de façon analogue aux tests d'intégration. Toutefois, ils débutent lorsque les tests d'intégration indiquent que le système satisfait aux objectifs de l'intégration fixés par le plan des tests (par exemple, 95% des cas de test d'intégration s'exécutent avec un résultat prévisible).



6. ***Evaluer les tests*** : l'objectif est d'évaluer les tests menés au sein d'une itération. Les résultats des tests sont évalués par rapport aux objectifs fixés dans le plan de test. Les ingénieurs de tests préparent également les métriques leur permettant de déterminer le niveau de qualité du logiciel et les tests restant à effectuer. Il faut surtout mesurer la complétude des test (degré de couverture des cas de test et celui des composants/code testés), la fiabilité (analyser les tendances qui se dégage des anomalies détectées, le ratio des tests exécutés avec succès). A partir de ces études, plusieurs actions peuvent être entreprises (1) exécuter les tests supplémentaires (2) assouplir les critères d'évaluation des tests (si la barre était très haute pour juger la qualité) et (3) isoler les parties ayant un niveau de qualité acceptable et les présenter comme résultat de l'itération en cours. Les autres parties doivent être révisées et testées à nouveau. L'évaluation des tests peut être décrite (complétude, fiabilité, actions suggérées) dans un document : description de l'évaluation des tests.



Le modèle de test décrit la manière dont est testé le système. Il contient les cas de test, les procédures de test et les composants de test. Le modèle de test se traduit également par le plan de test, les évaluation des tests effectués et les anomalies susceptibles de guider les autres enchaînements d'activités principaux (conception, implémentation).

V.7. Conclusion

En décrivant les enchaînements d'activités principaux l'un après l'autre, on a l'impression que le processus global de développement logiciel ne traverse cette séquence d'enchaînement d'activités qu'une seule fois entre le début et la fin du projet (comme le cycle de vie en cascade). Même si l'on traverse les cinq enchaînements d'activités de façon séquentielle, ce parcours se produit à chaque itération et non pas une seule fois pour tout le projet : on parcoure autant de fois que d'itérations réparties sur les 4 phases du processus. Toutefois, il faut reconnaître qu'il est probable que l'on n'aille pas jusqu'aux derniers enchaînements d'activités (implémentation, test) dans les premières itérations (phase de création). En bref, **le parcours des enchaînements d'activités est soumis aux particularités de chaque itération**.

Aussi, les enchaînements d'activités principaux, bien que décrits séparément dialoguent les uns avec les autres en produisant les artefacts et en utilisant ceux produits par les autres. L'association des divers enchaînements d'activités est faite selon le stade du cycle de vie où l'on se trouve.