الجمهورية الجزائرية الديمقراطية الشعبية

République Algérienne Démocratique et Populaire

Ministère de l'enseignement supérieur et de la recherche scientifique

Université de 8 Mai 1945 – Guelma -

Faculté des Mathématiques, d'Informatique et des Sciences de la matière

Département d'Informatique

Mémoire de Fin d'études Master

Filière : Informatique

Option : Master Académique

Thème :

La séparation avancée des préoccupations dans un cycle itératif

Encadré Par :

TADJER Houda

Présenté par :

GUECHI Nassiba

ABDAOUI Nabila

Juin 2015

REMERCIEMENT

Nous tenons tout d'abord à exprimer notre très grande gratitude au ***Dieu****, tout puissant qui nous a donné la force et la volonté d'élaborer ce travail.*

Et qui nous a doué des parents qui sont dignes de tout remerciement c'est grâce â leurs prières et patience que nous y arrivons â accomplir ce travail.

*On tient à remercier notre encadreur « **Mme Tadjer H** », qui a accepté de nous diriger, qu'il trouve ici toutes expressions de remerciement pour son assistance et ses orientations durant toute la période du travail.*

Nos vifs remerciements vont également aux ***membres du jury*** *pour l'intérêt qu'ils ont porté à notre projet en acceptant d'examiner notre travail et de l'enrichir par leurs propositions.*

Touts les enseignants du département d'informatique*, qui ont assisté à nos débuts en informatique, pour leurs précieux conseils.*

Nos collègues de fin de cycle*, qui nous ont donné leurs encouragements toute la durée de travail. Spécialement Asma.*

Nos familles, *qui durant nos études, nous ont toujours donné la possibilité de faire ce que nous voulions et qui ont toujours cru en nous.*

MERCI

DEDICACE

Je souhaite dédier ce mémoire de fin d'études :

A mes très chers parents *qui ont toujours pris soin de moi, et n'ont jamais cessé deCroire en moi. Des mots ne pourront jamais exprimer ce que j'ai dans le cœur.*
Papa, Maman, je vous aime.

A mon très cher mari Walid *qui est toujours avec moi, et je lui dis :*

Quand je t'ai connu, j'ai trouvé l'homme de ma vie, et la lumière de mon chemin. Ma vie à tes cotés est remplie de belles surprises.

Tes sacrifices, ton soutien moral et matériel, ta gentillesse sans égal, ton profond attachement m'ont permis de réussir mes études.

A mes frères *khamissi, Kader et son épouse Adila, à ma sœur Ahlem, Amel qui m'a toujours soutenu.*

A mes chère sœur *aussi Selma, Mounira, Boutayna et frère Aymen, Houssem, Mouhsen, et surtout ma deuxième mère Souad.*

A toute mes chère : *mes cousins et Cousines, en souvenir de toutes les joies et forces qui unissent notre chère famille.*

A tous mes amis pour votre sincère amitié :

Soumia, Saida, Rima, Imen, Selma, et surtout Ma cher binôme ***« noussaiba»*** *qui a toujours partagé cette passion et Cet amour pour l'informatique,*
Je ne peux trouver les mots justes et sincères pour vous exprimer mon affection et mes pensées, vous êtes pour moi des sœurs et des amis sur qui je peux compter.

A tous les étudiants de ma classe.

NABILA

DEDICACE

Je dédie ce mémoire de fin d'études

A

Mon très cher père et ma très chère mère

En témoignage de ma reconnaissance envers le soutien, les sacrifies et tous les

Efforts qu'ils ont faits pour mon éducation ainsi que ma formation

A

Mes chers frères, et mes chères sœurs

Pour leur affection, compréhension et patience

A

Tous mes amis pour votre sincère amitié :

Rima, Hala, Soumia, Saida, Dounia, Aicha, Ilhem, Asma

A tous les étudiants de ma classe.

A

Ma très chère ami Nabila

A

Tous ceux qui ont une relation de proche ou de loin

Avec la réalisation du présent rapport.

NOUSSAIBA

Résumé

L'évolution des systèmes industriels complexes actuels est souvent inéluctable et présente des enjeux économiques et qualitatifs considérables. Une des solutions envisagées consiste à utiliser la programmation orientée aspect. Ce nouveau paradigme de programmation cherche, en effet, à améliorer la séparation des préoccupations en « modularisant » les éléments transversaux des systèmes sous forme d'unités séparées appelées « aspects ». Cette séparation améliore la réutilisabilité et facilite la maintenance des systèmes, selon le paradigme de séparation des préoccupations, un système est un ensemble de préoccupations fonctionnelles et non fonctionnelles (les aspects).

Le travail présenté dans ce mémoire consiste en un enrichissement du développement agile, en particulier l'approche « Extreme Programming », par l'approche orientée aspect. En prenant en compte la nature transversale de certains besoins dans la phase d'analyse des besoins elle-même, la combinaison des deux approches améliore la maîtrise du changement des logiciels durant les itérations du développement agile, dans le sens d'une réduction de l'effort requis dans les phases de conception, du codage et du test des nouveaux besoins. Ensuite nous mettons en place un mécanisme de traçabilité qui nous permet de maintenir les différentes traces du système dans le temps.

Mots clés: Extreme programming, Histoire utilisateur, Ingénierie des besoins, Séparation des préoccupations, Aspect, Traçabilité.

Sommaire

Liste des figures

Liste des tableaux

Liste des abréviations et acronymes

XP	Extreme Programming
DSDM	Dynamic Software Development Method
ASD	Adaptive Software Development
FDD	Feature Driven Development
AOSD	Aspect Oriented Software Development
AOP	Aspect-Oriented Programming
AORE	Aspcet Oriented Requirements Engineering
SGBD	Système de Gestion de Base de Données
IDE	Integrated Development Environment
SQL	Structured Query Language
SDK	Software Development Kit
T-SQL	Transact- Structured Query Language
TDS	Tabular Data Stream
SUN	Stanford University Network
IDE	Integrated Development Environment
JAC	Java Aspect Components

INTRODUCTION GÉNÉRALE

Les systèmes informatiques sont de plus en plus grands. Cette évolution et en particulier l'ampleur des systèmes ont forcé le développement de techniques et pratiques architecturales visant à réduire les imposants problèmes de maintenance. L'ajout de fonctionnalités ainsi que la modification des systèmes ont, en effet, donné lieu à de nombreuses histoires effarantes de dépassements des coûts, des logiciels impossibles à faire évoluer dans des délais raisonnables ou encore à des taux de défectuosités atteignant des sommets.

Cette nouvelle réalité n'a pas seulement forcé les communautés à proposer des solutions architecturales mais également à se pencher sur le processus même de fabrication des logiciels. Processus destinés à produire des logiciels plus facilement maintenables et répondant plus adéquatement aux besoins, souvent on évolution, des clients.

Les méthodes agiles font face à l'évolution des besoins et sont devenues, grâce à leur pragmatisme, des approches privilégiées de développement des systèmes. Ces approches sont construites pour livrer les produits de haute qualité dans les délais et avec un budget prévisible. Le succès de l'application des approches de développement agiles pour la construction de systèmes facilement changeables dépend de la manière avec laquelle les besoins évolutifs sont découverts et structurés par les développeurs des logiciels.

D'un autre côté, la programmation orientée aspect a émergé suite à différents travaux de recherche, dont l'objectif principal était d'améliorer la modularité des applications afin de faciliter leur évolution et leur réutilisation. Les approches orientées aspects sont aujourd'hui disponibles à chaque phase du développement d'un logiciel: analyse des exigences, conception, ou encore implémentation.

La programmation orienté aspect de même que les processus Agiles sont des innovations qui ont gagné une certaine popularité durant dans la dernière décennie. Provenant de deux disciplines différentes du génie logiciel et se situant à des niveaux distincts, ces deux concepts ne sont pas conflictuels et ont, bien au contraire, un fort potentiel d'être utilisés conjointement. En ce sens, Agile peut offrir un cadre s'appliquant au cycle de vie du logiciel alors que la programmation orienté aspect offre une solution architecturale à un niveau plus technique.

De ce fait, ces deux concepts fournir une bonne réalisation des logiciels. Notre objectif consiste à réunir la séparation des préoccupations, l'ingénierie des besoins et l'approche XP dans le but de parvenir à une synergie qui améliore cette approche de développement par la

séparation des préoccupations. Cette intégration permet d'éliminer l'enchevêtrement et l'éparpillement dès la phase des besoins ce qui résulte en une bonne modularité dans les étapes ultérieures. Cela réduit l'effort de compréhension et de modification et par conséquent diminue le temps nécessaire à la conception et l'implémentation. Dans notre travail, nous cherchons comment intégrer les aspects dans l'approche Extreme Programming.

Dans ce mémoire, nous adopterons une organisation comportant quatre différents chapitres.

Dans le premier chapitre, nous allons décrire les différents modèles de développement existant dans la littérature. Ensuite nous discutons la technique de traçabilité, nous montrons ses étapes et nous terminons ce chapitre par la présentation des avantages de la traçabilité.

Dans le deuxième chapitre, nous allons présenter le concept de la séparation des préoccupations. Nous allons faire un aperçu sur les avantages, ainsi les limites de la programmation orientée objet. Ensuite, nous introduisons les concepts de développement orienté aspect. Nous terminons ce chapitre par la présentation l'ingénierie des besoins orientés aspects.

Le troisième chapitre sera consacré entièrement à la présentation de notre contribution: l'approche d'intégration qui est constitué d'un ensemble d'étapes et le modèle de traçabilité.

Le quatrième chapitre est destiné pour la présentation de la mise en œuvre de notre proposition.

CHAPITRE I : MODÈLES DE DÉVELOPPEMENT ET TRAÇABILITÉ

I.1 Introduction

Dans ce chapitre, nous présentons, dans une première section, le cycle de vie d'un logiciel. Ensuite nous décrivons quelques modèles de développement puis nous discutons la technique de traçabilité, nous décrivons ses différentes étapes et ensuite nous passons aux différentes représentations des traces.

I.2 Cycle de vie d'un logiciel

Le cycle de vie du logiciel regroupe toutes les étapes de l'existence d'un logiciel, de son apparition à sa disparition. Ce cycle de vie, tel que défini dans [1], se compose de plusieurs étapes afin de séparer les rôles impliqués dans la construction de logiciels. La définition d'un découpage en étapes permet d'isoler les étapes de construction du logiciel, afin de vérifier et valider chacune des étapes indépendamment les unes des autres, et d'assurer une certaine qualité du logiciel tout au long du cycle de vie.

En général, un cycle de vie comprend les étapes suivantes [2] :

- Analyse et définition des besoins
- Spécification
- Conception détaillé
- Implémentation (codage ou programmation)
- Tests unitaires
- Intégration
- Qualification
- Documentation
- Maintenance

I.3 Les modèles de développement

Cette section présente très brièvement différents modèles que l'on trouve dans la littérature. Il existe deux types de modèles: les modèles linéaires comme le modèle en cascade, le modèle en V et les modèles itératifs comme le modèle en spiral et le modèle de développement agile.

I.3.1 Le modèle en cascade

Le modèle de développement en cascade est probablement le processus le plus simple qui puisse exister. Il s'agit d'une démarche séquentielle de quelques étapes. C'est un modèle très

simple à appliquer et il est encore couramment utilisé [3]. Rappelons que selon ce modèle, l'analyse des besoins consiste en une phase de durée limitée durant laquelle tous les besoins et les spécifications servant au développement sont identifiés. Ce modèle a l'avantage de faciliter la planification d'un projet et l'estimation des coûts puisque le déroulement de toutes les étapes est défini dès le début du projet.

I.3.2 Le modèle en V

Ce modèle de cycle de vie est développé par MC Dermid et Ripkin en 1984. Ce modèle est une amélioration du modèle en cascade, chaque phase du projet à une phase de test qui lui est associé. Les phases de la partie montante, doivent renvoyer de l'information sur les phases en vis-à-vis lorsque des défauts sont détectés afin d'améliorer le logiciel. Le modèle en V met en évidence la nécessité d'anticiper et de préparer dans les étapes descendantes les "attendus " des futures étapes montantes : ainsi les attendues des testes de validation sont définis lors des spécifications, les attendus des tests unitaires sont définis lors de la conception [2].

I.3.3 Le modèle en spiral

Proposé par B. Boehm en 1988, ce modèle de cycle de vie tient compte de la possibilité de réévaluer les risques en cours de développement, il emprunte au prototypage incrémental mais lui adjoint une dimension relevant de la prise de décision managériale et non purement technique. Il couvre l'ensemble du cycle de développement d'un produit. Il met l'accent sur l'activité d'analyse des risques.

Le Modèle en spirale d'après [4] permet:

a) La détermination des objectifs du cycle, des alternatives pour les atteindre et des contraintes ;
b) L'analyse des risques, évaluation des alternatives à partir de maquettage et/ou prototypage;
c) Le développement et vérification de la solution retenue, un modèle « classique » (cascade ou en V) peut être utilisé ici ;
d) Revue des résultats et vérification du cycle suivant.

I.3.3.1 Analyse des risques

La mise en œuvre demande des compétences managériales et devrait être limitée aux projets innovants à cause de l'importance que ce modèle accorde à l'analyse des risques. Citons, par exemple :

- ✓ Risques humains
- ✓ Risques processus
- ✓ Risques technologiques

I.3.3.2 Conditions d'application

Le modèle en spirale s'applique essentiellement en interne, lorsque les clients et les fournisseurs font partie de la même entreprise, si l'analyse de risque démontre que le projet doit être continué, une équipe peut être réaffectée au projet. Alors que dans une relation client-fournisseur ordinaire, il y a eu signature de contrat et donc l'effort doit être estimé à l'avance. Le modèle en spirale ne peut donc s'appliquer. Ou bien il doit être adapté en signant des contrats partiels pour chaque itération [1].

I.3.4 Modèle de développement agile

I.3.4.1 Qu'est-ce qu'une méthode agile

Definition1 : d'après [5] Une méthode agile est une approche itérative et incrémentale, qui est menée dans un esprit collaboratif, avec juste ce qu'il faut de formalisme. Elle génère un produit de haute qualité tout en prenant en compte l'évolution des besoins des clients.

Definition2 : Le développement Agile est une philosophie. Il est une façon de penser de développement de logiciels. La description canonique de cette façon de penser est le manifeste Agile, une collection de 4 valeurs et 12 principes [6].

Les adeptes du développement agile le désignent plus souvent comme suit: « Agile = Itératif + Incrémental + Adaptatif », voici en figure I.1 un modèle générique de la vision Agile.

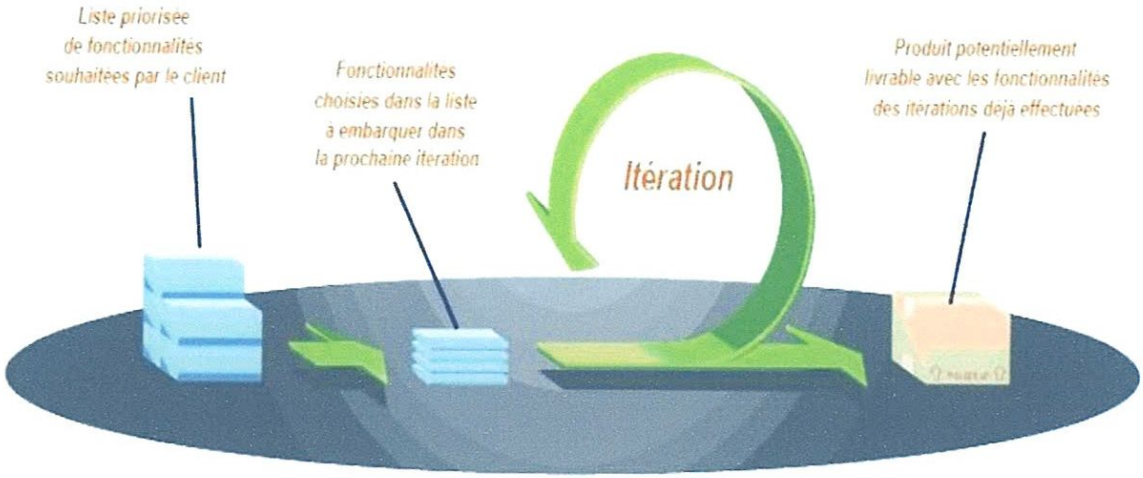

Figure I.1: *Vue globale de cycle de vie Agile [7]*

Ce cycle de vie permet de livrer de façon successive les fonctionnalités souhaitées par le client. Chaque itération permet de fournir un produit livrable, utilisable, comprenant les fonctionnalités des itérations effectuées.

I.3.4.2 Le manifeste agile

En 2001, dix-sept figures éminentes du développement logiciel se sont réunies dans la station de ski de Snowbird, dans l'état de l'Utah (états unis d'Amérique), pour débattre, mettre en évidence les fondements unificateurs et offrir un cadre à de nouvelles tendances de développement dont le succès commençait à faire beaucoup d'écho. De cette réunion émergea le manifeste agile, considéré comme la définition canonique du développement agile et de ses principes sous-jacents.

Le manifeste propose quatre valeurs directrices, du développement agile, qui donnent précédence à certains pratiques et artéfacts plutôt que d'autres:

- L'équipe et les interactions plutôt que les processus et les outils.
- Produire un logiciel entièrement testé et qui fonctionne plutôt qu'une documentation compréhensible.
- Collaborer avec le client plutôt que négocier un contrat.
- Réagir au changement plutôt que suivre une planification.

A la base de ces valeurs, un ensemble de 12 principes sont identifiés. Le but de ces principes est d'aider à déterminer si un développeur suit une méthodologie agile ou non [8]. Les méthodologies agiles devraient se conformer aux principes suivants:

- La priorité est de satisfaire le client par des livraisons rapides et continues de logiciels utiles.
- Intégrer les changements aux besoins même s'ils arrivent tard dans le processus de développement. Les méthodes Agiles intègrent rapidement les changements de façon à offrir un avantage compétitif au client.
- Livrer fréquemment des logiciels opérationnels, de quelques semaines à quelques mois en visant des délais courts.
- Les clients et les développeurs doivent travailler main dans la main quotidiennement tout au long du projet.
- Élaborer des projets autour d'individus motivés. Leur procurer l'environnement et le support nécessaire et leur faire confiance pour réaliser le travail.
- La façon la plus efficace de transmettre l'information à une équipe et entre les membres est par des conversations en face à face. Le logiciel opérationnel est la principale mesure du progrès.
- Agile favorise le développement à rythme "normal".
- Les gestionnaires, développeurs et utilisateurs devraient être en mesure de maintenir un rythme constant et ce, indéfiniment.
- Porter une attention continue à l'excellence technique et à la bonne conception améliore l'agilité.
- La simplicité, maximiser la quantité de travail à ne pas faire est essentielle. Il faut se focaliser sur le plus important.
- Les meilleures architectures, exigences et conceptions prennent naissance dans des équipes qui se gèrent elles-mêmes.
- Régulièrement, l'équipe fait une réflexion sur les façons de devenir plus efficace, s'ajuste et modifie son comportement en conséquence.

Parmi les approches agiles qui sont actuellement utilisées, On distingue :

- Extreme Programming (XP) [9]
- Dynamic Software Development Method (DSDM) [10]
- Adaptive Software Development (ASD) [8]

- Crystal Methodologies [11]
- SCRUM [12]
- Feature Driven development (FDD) [13]

Dans ce qui suit on va présenter la méthode XP sur laquelle se base notre travail.

I.3.4.3 L'extreme programming (XP)

La méthode Extreme Programming (XP) est née suite à des problèmes causés par le cycle de développement longs des méthodes de développement traditionnelles [14]. Elle a débuté par de simples opportunités de faire un travail rapide [15], et après un nombre d'essais réussis dans la pratique, la méthodologie XP a été théorisée en concordance avec les principes clés et les pratiques utilisées [9].

Actuellement, XP est la méthode la plus connue parmi les méthodes Agiles.

La réussite de la méthode XP se base sur 4 principes [16] :

- La communication.
- La simplicité.
- Le feedback.
- Le courage.

Ces 4 principes de base ont conduite à 12 pratiques que nous citons ci-après. Nous maintenons les appellations d'origines qui sont très utilisées dans le domaine [17, 18, 19, 20].

I.3.4.3.1 Les pratiques de l'XP

Dans cette section, nous présentons les 12 pratiques de l'approche XP.

1. **Jeu du Planning**: Planifier la prochaine version à livrer en tenant compte des priorités.
2. **Petites livraisons**: Les incréments des versions dites de développement sont minimes.
3. **Utilisation de métaphores**: L'utilisation de métaphores montrant comment le système complet devrait fonctionner guide le développement.
4. **Conception simple** : Une analyse, simple, sinon sommaire, est effectuée et les programmeurs commencent à travailler en utilisant celle-ci.

5. **Tests de recette**: Les programmeurs écrivent continuellement des tests qui doivent fonctionner entièrement et parfaitement avant de continuer le développement. Le client écrit des tests qui lui permettent de démontrer que la fonctionnalité est couverte.
6. **Programmation par paire**: Les programmeurs travaillent par paire sur chaque poste.
7. **Refactoring**: Méthode de travail qui consiste à améliorer la conception en restructurant le code déjà programmé.
8. **Intégration continue**: L'intégration des divers composants ou bibliothèques doit être continue et validée à chaque livraison de version de production ou de développement.
9. **Appropriation collective du code**: L'appartenance du code est collective, chaque paire de programmeurs est responsable du code et peut l'améliorer à n'importe quel moment.
10. **Rythme soutenable**: La cadence de travail doit être soutenue (par exemple : 40 heures/semaine); les heures supplémentaires sont ponctuelles et sont considérées comme une exception et non comme une règle.
11. **Client sur site**: Inclure un client dans l'équipe de développement. Ce dernier doit être disposé et disponible pour répondre aux questions de l'équipe de développement.
12. **Convention de codage**: L'utilisation de règles de programmation, au niveau du format du code, permet à chaque programmeur de se retrouver plus facilement.

I.3.4.3.2 Cycle de vie d'un projet XP

L'approche XP fournit un modèle de cycle de vie d'un logiciel qui est utilisé comme un guide pour l'organisation de l'équipe de développement. Ce modèle est présenté dans la figure I.2.

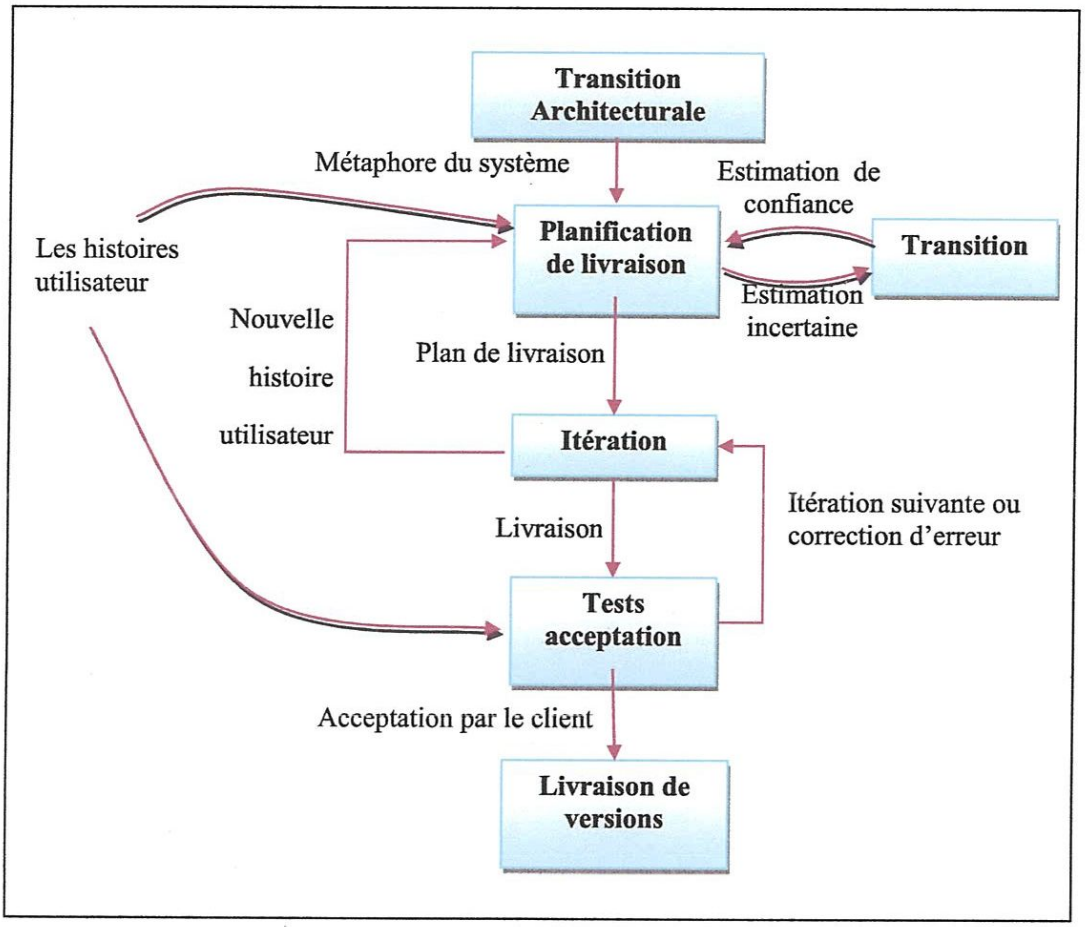

Figure I.2 : *Cycle de vie d'un projet XP [16]*

Ce qui suit est une explication des étapes qui composent ce cycle :

Les histoires utilisateur

Les histoires utilisateur sont un concept important dans le développement XP et sont habituellement le point de départ de tout processus XP. En les choisissant, le client déclenche le processus des itérations. Elles doivent être courtes et écrites par l'utilisateur lui-même en utilisant sa terminologie et non celle de l'équipe de développement. Ces histoires devraient alimenter la réunion de planification de livraison.

Planification de livraison

Une réunion de planification de livraison est utilisée pour créer le plan de livraison, qui énonce l'ensemble du projet.

Ce plan de livraison indique les histoires d'utilisateur qui seront implémentées et dans quelle livraison elles peuvent se produire. Il indique également comment plusieurs itérations

sont planifiées et quand chaque itération sera livrée. Cela se fait par la négociation entre les parties intéressées en utilisant des estimations dérivées des histoires d'utilisateur. Les estimations sont produites par les membres de l'équipe potentiellement avec une contribution de la part des clients.

Les clients mettent les priorités des histoires utilisateur et éventuellement l'apport des membres développeurs.

Itérations

C'est, au début de chaque itération que des modifications peuvent être apportées. Plus le temps de l'itération est réduit, plus l'équipe de développement peut répondre au changement qui l'affecte.

Au début de chaque itération, une réunion de planification d'une itération est tenue pour déterminer exactement ce qui va se passer au sein de cette itération.

Livraison

La réunion de planification de livraison devrait identifier les fonctionnalités significatives du système en relation avec les clients et l'état du système à diverses étapes. Ces livrables significatifs devraient être disponibles aux utilisateurs lorsqu'ils sont achevés.

I.4 Synthèse des différences fondamentales entre approche traditionnelle et approche agile

La synthèse ci-après présente, dans le tableau I.1, les différences majeures par thème entre une approche traditionnelle et une approche agile [5].

Thème	Approche traditionnelle	Approche agile
Cycle de vie	En cascade ou en V, phases séquentielles	Itératif et incrémental
Planification	Prédictive, caractérisée par des plans plus ou moins détaillés sur la base d'un périmètre et d'exigences définies et stables au début du projet	Adaptative avec plusieurs niveaux de planification (macro-et micro planification) avec ajustements si nécessaires en fonction des changements survenus.
Documentation	Produite en quantité importante comme support de communication, de validation et de contractualisation	Réduite au strict nécessaire au profit d'incréments fonctionnels opérationnels pour obtenir le feedback du client
Équipe	Une équipe avec des ressources spécialisées, dirigées par un chef de projet	Une équipe responsabilisée où l'initiative et la communication sont privilégiées, soutenue par le chef de projet
Qualité	Contrôle qualité à la fin du cycle de développement. Le client découvre le produit fini	Un contrôle qualité précoce et per manent, au niveau du produit et du processus. Le client visualise les résultats tôt et fréquemment
Changement	Résistance voire opposition au changement. Processus lourds de gestion des changements acceptés	Accueil favorable au changement inéluctable, intégré dans le processus.
Suivi de l'avancement	Mesure de la conformité aux plans initiaux. Analyse des écarts	Un seul indicateur d'avancement : le nombre de fonctionnalités implémentées et le travail restant à faire
Gestion des risques	Processus distinct, rigoureux, de gestion des risques.	Gestion des risques intégrée dans le processus global, avec responsabilisation de chacun dans l'identification et la résolution des risques. Pilotage par les risques
Mesure du succès	Respect des engagements initiaux en ter mes de coûts, de budget et de niveau de qualité	Satisfaction client par la livraison de valeur Ajoutée

Tableau I.1 : *Les différences majeures par thème entre une approche traditionnelle et une approche agile [5]*

I.5 Critères de qualité d'un produit logiciel

Le but du développement de logiciel est de produire des logiciels de qualité. La qualité d'un produit logiciel est déterminée, en génie logiciel, par plusieurs facteurs. On trouve parmi ces derniers [2]:

- **La validité:** l'aptitude du logiciel de répondre aux besoins fonctionnels définis par les cahiers de charge et les spécifications. (à remplir exactement les fonctions attendues de ce produit logiciel).
- **La vérifiabilité :** faciliter de préparation des procédures de tests.
- **Efficacité :** l'exploitation optimale de ressources matérielles.
- **Robustesse ou fiabilité:** l'assurance qu'un produit logiciel peut même fonctionner dans des conditions anormales.
- **Extensibilité ou maintenance:** c'est la facilité avec la quelle un logiciel se prête à sa maintenance (une modification ou une extension des fonctions).
- **Réutilisabilité :** aptitude d'un logiciel à être réutilisé, en tout ou en partie, dans de nouvelles applications.
- **Compatibilité:** facilité avec laquelle un logiciel peut être combiné avec d'autres logiciels.
- **Efficacité:** Utilisation optimales des ressources matérielles.
- **Portabilité:** facilité avec laquelle un logiciel peut être transféré sous différents environnements matériels et logiciels.
- **Intégrité et protection des données:** aptitude d'un logiciel à protéger son code et ses données contre des accès non autorisés.
- **Facilité d'emploi:** facilité d'apprentissage, d'utilisation, de préparation des données, d'interprétation des erreurs et de rattrapage en cas d'erreur d'utilisation.

Parfois, ces facteurs sont contradictoires, le choix d'un compromis doit s'effectué selon le cadre du projet

I.6 La traçabilité

La traçabilité fournit la capacité de suivre le cycle de vie des éléments logiciels comme les objets et les threads. En effet, les traces sont capables d'apporter des informations sémantiques riches décrivant avec détails l'exécution du système. Et ceci dans le but de suivre l'état et le comportement du système tout au long de son évolution dans différents niveaux

d'abstractions. Compte tenu de l'importance de traçabilité pour la performance de logiciel en va voir dans ce qui suit une preuve définition sur la traçabilité.

Définition 1 : La traçabilité est la capacité de décrire et de suivre le cycle de vie d'un objet et un moyen de modélisation des relations entre les entités logicielles d'une manière explicite [21]. Cette définition est adaptée pour la trace d'exécution. Elle est appliquée dans le domaine de l'analyse des systèmes dynamiques.

Définition 2: La capacité de décrire et de suivre la vie d'un besoin, dans les deux directions amont et avale d'un projet, c.-à-d., de ses origines en passant par son développement et sa spécification, à son déploiement et son utilisation dans les phases suivantes tout en considérant les périodes d'amélioration et d'itération émanant d'autres phases (conception, construction, test…) [22].

I.6.1 Etapes de traçabilité

Pour mettre en place un mécanisme de trace dans un projet de développement logiciel, on doit suivre quatre étapes principales pour qu'on puisse se contenter du mécanisme de trace comme l'indique la figure I.3 [23].

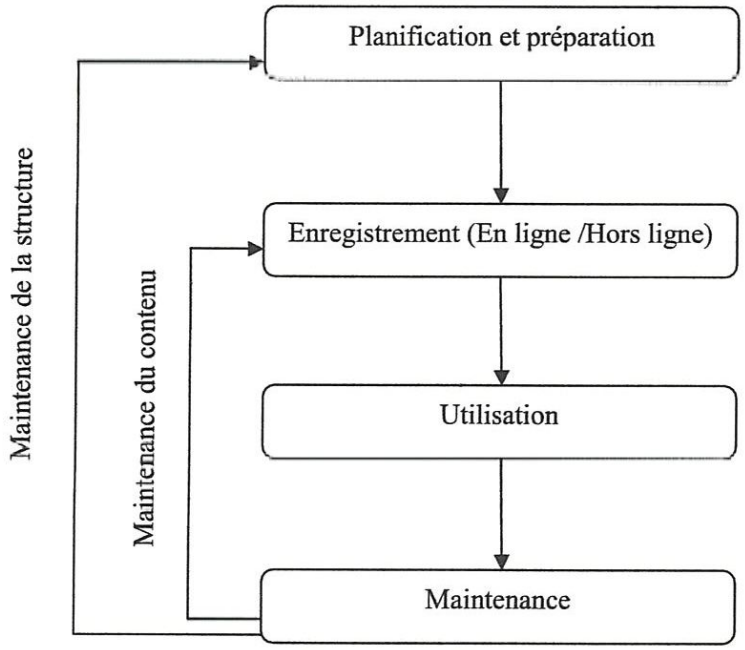

Figure I.3 : *Etape de traçabilité [21]*

1. Planification et préparation : Durant cette étape on choisit les outils et les méthodes de traçage. En effet, il existe une grande variété d'outils supportant différentes approches de traçage. A titre d'exemple, Echo est un outil utilisé avec le processus AGIL [24].

2. Enregistrement : Il s'agit de la sauvegarde des traces. Deux modes d'enregistrements sont envisagés :

- **Hors ligne** : la sauvegarde des traces est effectuée après la terminaison de l'activité actuelle de développement d'une manière manuelle ou automatique.
- **Enligne**: la sauvegarde est automatique et immédiate des traces.

3. Utilisation : Les données que décrivent les traces sont accessibles afin de rédiger des rapports ou d'extraire des informations dans un but bien déterminé.

4. Maintenance : C'est l'activité résultante d'un changement structurel du processus de développement ou une erreur ou oublie dans les données de traces. Deux cas se présentent·

- Un changement ou ajustement au niveau des outils de traçage et donc on revient à la première étape.
- Des erreurs qui doivent être corrigées au niveau de l'enregistrement.

I.6.2 Représentation des traces

Afin de pouvoir travailler avec des traces et de les utiliser, des visualisations, des représentations et des interfaces utilisateur sont nécessaires. Il existe plusieurs représentations, certains ou qui sont implémentées dans les outils actuels de traçabilité des exigences. Essentiellement, selon Wieringa [25], ils peuvent être classés dans les matrices, les références croisées et représentations graphique. Figure I.4 illustre les trois représentations.

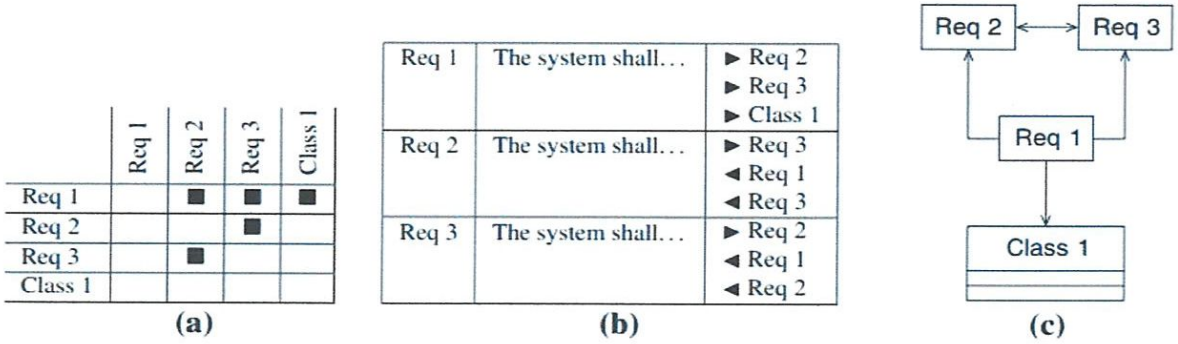

Figure I.4 : *Représentation des traces [21]*

I.6.2.1 Matrice : Une matrice de traçabilité est une grille à deux dimensions qui représente les liens de traçabilité qui existent entre les deux ensembles d'artéfacts, tels que les besoins, les éléments de conception, ...etc. Les lignes et les colonnes de la grille sont associées avec les artéfacts, les marques aux intersections représentent l'existence d'un lien. Figure I.4 (a), une boîte noire signifie, qu'il existe un lien orientée de l'artéfact de gauche à l'artéfact au dessus.

Cette forme de visualisation est celui plus traditionnel dans le domaine de la traçabilité des besoins. Alors que les premières formes de matrices de traçabilité seulement fourni un support pour un seul type de marque représentant l'existence ou l'inexistence d'un lien entre deux artéfacts, aujourd'hui les matrices de traçabilité peuvent être améliorées pour inclure des informations supplémentaires sur les artéfacts et les liens [23]. Par exemple, un artéfact dans la matrice est généralement référencé à l'aide d'un identificateur, mais les interfaces utilisateurs modernes peuvent fournir des fenêtres pop-up directement montrant une méta-information de l'artéfact ou contenu, si nécessaire. En outre, les types de lien ou autres informations pouvaient être codées à l'aide de différentes couleurs ou des symboles [26].

Même pour les utilisateurs non-techniques, les matrices de traçabilité sont faciles à comprendre et à utiliser avec des scénarios simples, telles que l'enregistrement ou la vérification de l'existence des liens uniques entre artéfacts [25]. Toutefois, pour des projets réels, les matrices de traçabilité peuvent devenir très grand et illisible [27]. Le contenu des artéfacts est habituellement non affiché dans la matrice, mais référencés à l'aide d'identificateurs uniques. Cette séparation et la nature de deux dimensions de la grille, mettre difficile de suivre des liens à travers plusieurs artéfacts récursivement. Enfin, il est presque impossible de représenter liens n-aires en une matrice de traçabilité à deux dimensions dans une manière compréhensible.

I.6.2.2 Références croisées: La deuxième représentation des traces est sous la forme de références croisées comme l'indique la figure I.4 (b). Les références croisées peuvent être intégrées et représentées en langage naturel. L'avantage de cette représentation est qu'elle peut être interprétée par n'importe quel lecteur.

Cependant, le regard des liens est très restreint puisqu'un utilisateur ne peut voir que les liens d'un seul élément à la fois.

I.6.2.3 Graphe : La troisième représentation des traces est sous la forme de graphe comme l'indique la figure I.4 (c). C'est plus clair que les deux autres.

En effet, l'interprétation des graphes est plus facile et apporte plus d'informations. Même pour les systèmes complexes, on s'assure que c'est toujours clair. Cette présentation des traces est la mieux adaptée dans le cas de l'ingénierie dirigée par les modèles. Plusieurs approches sont proposées en vue de suivre un système en cours d'exécution et qui sont basées sur le mécanisme de trace à ce niveau.

I.6.3 Les types de lien de traçabilité

Dahlstedt et Persson [28], Espinoza et al. [29], Limón et Garbajosa [30], Ramesh et Jarke [31], Spanoudakis et Zisman [32], von Knethen et Paech [33] et beaucoup d'autres ont publié des suggestions pour classer les liens de traçabilité basés sur leurs propriétés structurelles et sémantiques. Ces classifications proposées sont difficiles à évaluer et à comparer parce qu'il n'y a aucun niveau d'abstraction commun et habituellement pas de définition formelle ou sans ambiguïté des différentes catégories. En outre, les catégories elles-mêmes ne peuvent être séparées clairement : c'est difficile à dire, si un lien affine (comme exigence-affine-exigence) appartienne à une des classes d'évolutionnaire, inclusion, raffinement (de niveau) ou des liens de dépendance mentionné par von Knethen et Paech [33]. Selon le point de vue, il montre des caractéristiques de chacune des classes. Ainsi, la sémantique d'un lien est actuellement dépendante de la vue et l'interprétation des acteurs [34].

Une des plus récentes classifications étendues dans la communauté RE a été présentée par Spanoudakis et Zisman [31]. Il a été dérivé d'une analyse approfondie de la littérature et définit huit catégories de liens :

1. ***Liens de dépendance*** entre les deux artéfacts a1 et a2 décrivent que a2 repose sur l'existence de a1 ou les changements de a1 à entraîner des changements potentiels dans a2.
2. ***Liens raffinement*** sont utilisés dans les hiérarchies d'abstraction pour décrire comment les artéfacts complexes sont décomposés en plus petits, plus concrète ou des artéfacts plus faciles à gérer.
3. ***Liens d'évolution*** sont utilisés lorsqu'un artéfact remplace un autre, comme si une version plus récente remplace une ancienne version ou si un ensemble des besoins en chevauchement est retravaillé à un ensemble de besoins plus élaborés.

4. ***Liens de satisfiabilité*** indiquent un autre type d'évolution durant le développement: ils sont utilisés pour exprimer qu'un artéfact en aval est créé conforme à un artéfact en amont, comme un élément de design qui satisfait à une ou plusieurs besoins.
5. ***Liens de chevauchement*** peuvent être utilisés quand deux artéfacts décrivent des caractéristiques ou des aspects communs du système. À titre d'exemple, considérons la représentation d'un besoins en langage naturel et dans une notation formelle.
6. ***Liens de conflit*** sont utilisés pour exprimer un conflit existant entre deux artéfacts. Pour être utiles, ces liens doivent inclure des informations sur la façon de résoudre le conflit et sur quelles alternatives sont possibles sous quelles hypothèses.
7. ***Liens de rationalisation*** sont également utilisés pour transporter les informations sur les décisions, où ils sont utilisés pour documenter la raison de la création et l'évolution des artéfacts.
8. ***Liens de contribution*** décrivent les diverses relations entre les intervenants et les artéfacts.

I.6.5 Avantages de traçabilité

La gestion de la traçabilité est un enjeu important dans les projets d'ingénierie des SI. Les avantages de la traçabilité ont été signalés depuis longtemps [35] et les standards d'ingénierie la recommandent dans leurs processus [36].

La traçabilité dans un projet de développement de logiciel peut aider à assurer les autres qualités du logiciel, tels que la pertinence et la compréhensibilité. En revanche, négligeant la traçabilité peut entraîner au moins logiciel maintenable et défauts dus à des incohérences ou des omissions.

Les avantages de traçabilité sont nombreux, on peut citer quelques avantages:

- Suivi automatisé, fiable et précis de votre production,
- Assurance qualité,
- Satisfait à de nombreuses exigences réglementaires,
- Amélioration de vos processus,
- Meilleur suivi de production,
- Meilleure visibilité de la production,
- Amélioration des outils de gestion de production,
- Gain de temps, et de productivité,

- Pertes maîtrisées, et quasi-supprimées,
- Relations de confiance,

I.7 Conclusion

Dans ce chapitre, nous nous sommes intéressés à l'ingénierie des logiciels et nous avons essayé de donner une présentation de cycle de vie d'un logiciel, suivi avec quelques modèles de développement avec leur processus et particulièrement le modèle de développement agile (XP), et nous avons montré le mécanisme de traçabilité. A la fin du chapitre, nous avons cité quelques avantages de traçabilité. Dans le chapitre suivant nous nous intéressons à présenter la notion de l'orienté aspect.

CHAPITRE II : LE DÉVELOPPEMENT ORIENTÉ ASPECT

II.1 Introduction

L'implémentation des besoins non-fonctionnels pose souvent des problèmes avec les méthodes de programmation actuelles, et en se retrouve avec des problématiques dont l'implémentation est dispersée dans les différents modules fonctionnels du système. L'approches orientée aspects constituent un nouveau paradigme qui vient combler l'absence d'un support adéquat des besoins non-fonctionnels. L'objectif de ce chapitre est de présenter les différents concepts de développement par aspect.

II.2 La séparation des préoccupations

La conception des systèmes logiciels consiste à décomposer un problème donné en sous-problèmes (de petites unités ou modules) qui sont traités séparément les uns des autres (figure II.1). Le système final est ensuite construit par l'implémentation des différents modules conceptuels et leur composition. La programmation de ces unités permet de réduire la complexité du problème à implémenter, augmentant, entre autre, la flexibilité et facilitant aussi la compréhension du système [37]. C'est l'approche désormais classique de modularisation évoquée depuis les années 1970 par R. Gauthier [38] qui a été adoptée par l'ensemble des approches de développement traditionnelles (procédurale, fonctionnelle et objet).

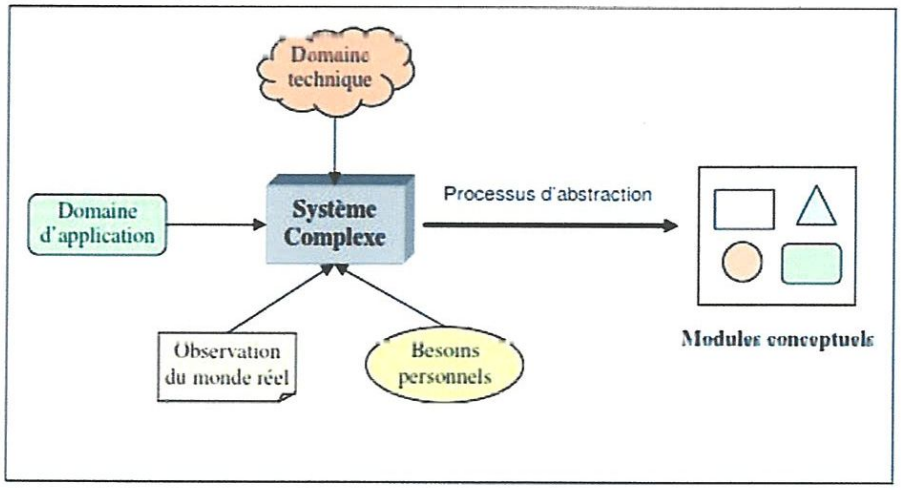

Figure II.1 : *Conceptualisation d'une application*

Aujourd'hui, toutes ces approches visent à appliquer le principe de séparation des préoccupations dans leur processus de développement [8, 9]. Une préoccupation est un concept, un but particulier ou une unité d'intérêt. Un système typique admet principalement deux types de préoccupations: fonctionnelles et non-fonctionnelles. Par exemple, l'abstraction

d'une carte de crédit admet une préoccupation fonctionnelle concevant le processus de payement et des préoccupations systémiques assurant la gestion de la connexion, l'authentification, l'intégrité de la transaction de paiement, la sécurité, etc.

En adoptant l'idée classique de l'approche de modularisation, le principe de séparation des préoccupations consiste à identifier séparément l'ensemble des préoccupations d'un système logiciel et à les adresser d'une façon relativement indépendante, aussi bien à la phase de conception qu'à la phase d'implémentation. Ce principe est reconnu essentiel au développement des logiciels: il améliore la lisibilité et la flexibilité du système permettant ainsi l'évolution, l'adaptation et la réutilisation de toute ou une partie de l'application [10].

II.2.1 L'approche orienté objet

En appliquant le principe de séparation de préoccupations, l'approche orientée objet met en avant la description d'entités du domaine d'application, définies sous la forme d'objets admettant chacun une identité, un état et un comportement particulier. Les langages à objets [8, 11] permettent en effet de décomposer le système en composants paramétrables, indépendants les uns des autres qui s'engagent à fournir chacun un ou plusieurs services au reste du système. Le système est donc conçu comme une composition d'objets collaborant entre eux.

C'est grâce aux nombreux concepts du paradigme orienté objet [12] (l'encapsulation [13], le mécanisme d'héritage, la relation de généralisation/spécialisation, le polymorphisme, etc.) que l'approche orientée objet améliore la réutilisabilité d'un système complexe.

Cependant, cette réutilisation n'est pas toujours aisée. C'est notamment le cas pour les applications dont la construction ne se limite pas à la simple définition d'un ensemble de services attendus, mais nécessite également la prise en compte de différentes propriétés non fonctionnelles. En effet, du fait des contraintes d'exécution, il est constamment nécessaire d'y intégrer différentes propriétés non fonctionnelles comme, par exemple, la gestion de la mémoire, la persistance ou dans un cadre réparti, la répartition des données, la synchronisation, etc.

La décomposition fonctionnelle adoptée par l'approche orientée objet nous donne toujours comme résultat des préoccupations transversales (*crosscutting concerns*) affectant ainsi le développement de l'application de différentes façons. Dans ce qui suit, nous présentons deux problèmes récurrents de l'approche orientée objet ainsi que leurs implications.

II.2.2 Limites de la programmation orientée objet

Une utilisation classique de la programmation orientée objet ne fournit pas de solution pour aboutir à des programmes clairs et élégants, ce type de programmation pose deux problèmes principaux: l'éparpillement et l'enchevêtrement.

II.2.2.1 L'éparpillement (*Code Scattering*)

La dispersion du code signifie que la spécification d'une propriété du système n'est pas encapsulée dans un seul module. Par exemple, dans un système de gestion de base de données, il est possible que l'optimisation des performances, la journalisation (*logging*) et la synchronisation puissent concerner toutes les classes accédant à la base de données. On voit donc que ces aspects doivent être implémentés dans plusieurs modules sans être bien circonscrits. Il s'agit très souvent de portions de code similaires à ajouter un peu partout dans les modules concernés par ces propriétés.

II.2.2.2 L'enchevêtrement (*Code Tangling*)

Chaque module du système contient la description de plusieurs propriétés ou de fonctionnalités différentes. Un problème qui découle directement du constat précédent est que certaines préoccupations non-fonctionnelles viennent recouper l'implantation des préoccupations fonctionnelles. Il en résulte la présence d'éléments de plusieurs préoccupations dans l'implantation d'un seul et même module.

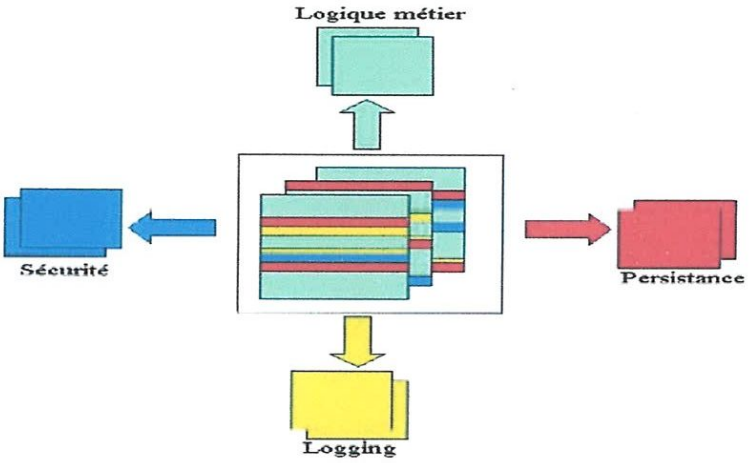

Figure II.2: *Un système vu comme un ensemble de préoccupation*

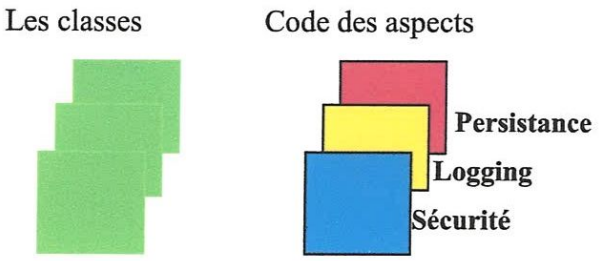

Figure II.3 : *Localisation du code d'une préoccupation transversale dans un aspect*

II.3 Les préoccupations transversales

Pour illustrer les préoccupations transversales, considérons l'exemple d'une librairie électronique [39] dont la réalisation nécessite, en plus des services attendus, la prise en compte de différentes propriétés non-fonctionnelles affectant l'application dans sa globalité. Dans cette application, les clients utilisent un réseau pour consulter la liste des ouvrages disponibles et éventuellement les commander. Plusieurs clients peuvent être connectés en même temps et plusieurs commandes peuvent être traitées en parallèle.

La conception objet de cette application conduit à définir notamment les classes: *Librairie*, *Client*, *Ouvrage*, *Commande*, *Banque* et *CarteDeCredit* qui représentent les entités du domaine d'application, fournissant les services de base de l'application (e.g. la consultation d'un ouvrage ou la passation d'une commande). Par ailleurs, nous pouvons identifier au moins trois propriétés non fonctionnelles différentes correspondant à des propriétés non-fonctionnelles de cette application : la *distribution*, la *persistance* et la *synchronisation*.

- La ***distribution*** apparaît du fait que les instances des différentes classes mises en jeu se trouvent sur des sites distants. Cette propriété non fonctionnelle regroupe entre autre la gestion des communications de site à site et la définition du protocole de communication utilisé.
- La ***persistance*** se justifie parce que des objets comme, par exemple, les ouvrages mis en vente, doivent persister à un arrêt de l'application. La sauvegarde de tels objets sur un support de masse et la gestion des mises à jour de cette sauvegarde font partie de la définition de cette propriété non fonctionnelle, quelle que soit la technique de persistance utilisée.
- La ***synchronisation*** est mise en évidence, notamment, par le besoin de synchroniser les accès concurrents aux structures des objets. C'est le cas, par exemple, lorsqu'un

client demande le prix d'un ouvrage pendant que le libraire modifie ce même prix. Cette propriété non fonctionnelle englobe la gestion de tels accès concurrents.

II.4 Les approches de séparation des préoccupations

La séparation des préoccupations est un concept présent depuis de nombreuses années dans l'ingénierie du logiciel [37, 15]. Les différentes préoccupations des concepteurs apparaissent comme les premières motivations pour organiser et décomposer une application en un ensemble d'éléments compréhensibles et facilement manipulables. La séparation en préoccupations apparaît dans les différentes étapes du cycle de vie du logiciel et est donc de différents ordres. Il peut s'agir de préoccupations d'ordre fonctionnel (séparation des fonctions de l'application), technique (séparation des propriétés du logiciel système).

Par ces séparations, le logiciel n'est plus abordé dans sa globalité, mais par parties. Cette approche réduit la complexité de conception, de réalisation, mais aussi de maintenance du logiciel et améliore la compréhension, la réutilisation et l'évolution. De nos jours, une large part des travaux sur la séparation traite de trois principales approches:

- La programmation orienté aspect [14]
- Les filtres de composition [16]
- La séparation multidimensionnelle des préoccupations [17]

II.4.1 La programmation orientée aspect (POA)

La programmation orientée aspect (Aspect Oriented Programing AOP) a été définie par Greogor Kickzales de xerox parc en 1996. AOP est une philosophie de programmation qui est essentiellement question du style. Elle résout des problèmes qui peuvent être traités dans des approches classiques mais d'une manière plus élégante [14] et est un paradigme de programmation qui permet de réduire fortement le couplage entre les différents aspects techniques d'un logiciel (préoccupations transverses). Au lieu d'avoir un appel direct à un module technique depuis le module de base métier, en programmation orientée aspect le code de base est concentré sur sa préoccupation métier, tandis qu'un aspect est spécifié de façon autonome, prenant en charge de faire appel aux modules techniques tel que l'authentification requis à un certain point d'exécution du système [18,14]. La programmation orientée aspect est une technologie transverse est n'est pas liée à un langage particulier. Elle peut être mise en œuvre aussi bien avec un langage orienté objet tel que java qu'avec un langage impératif comme C. Seul il faut avoir un tisseur d'aspect pour le langage cible.

Le langage orienté aspect le plus maturé sans doute est AspectJ [2,1], sans négliger d'indiquer d'autres langages orientés aspect importants comme Jasco, CaesarJ, AspectS, Object Teams, HyperJ, JBOSS, Compose*, DemeterJ, AspectC++,.. etc. nous referons le lecteur à [1] et à l'étude mené dans [20] pour avoir plus de détail sur les différents langages orientés aspect.

II.4.1.1 AspectJ

AspectJ est le langage orienté aspect le plus utilisé dans le développement orienté aspect. Il est une extension à java, les composantes de base sont écrites en java pur alors que les aspects sont écrit suivant la syntaxe aspectJ. AspectJ permet de définir les aspects tout en exprimant les règles de leur intégration : spécification de coupure et advice. Il dispose de son propre compilateur qui permet de tisser et combiner les aspects et les modules de base pour donner en résultat un code interprétable standard interprété par la machine virtuelle java. Le modèle de point de jonction d'AspectJ inclut les points de jonction suivants dans le flot de contrôle :

- un appel de méthode ou de constructeur
- L'exécution d'une méthode ou d'un constructeur
- L'accès en lecture ou écriture d'un champ
- L'exécution d'un bloc « catch » qui traite une exception java
- L'initialisation d'un objet ou d'une classe

Aussi, il faut noter qu'Aspectj introduit aussi un mécanisme d'introduction qui permet l'ajout de nouvelles classes et méthodes sur des classes. Une coupe est introduite grâce au mot clé pointcut, Une coupe permet de regrouper un ou plusieurs points de jonction. Comme elle Peut utiliser des caractères spéciaux appelés wildcards, qui permettent de généraliser des coupes. En effet, l'expression suivante par exemple permet de réunir tous les points de jonctions de type appel à une méthode commençant par get : pointcut allget() : call(* *..get*(..)). [18].

La figureII.4 montre un programme orienté aspect selon la syntaxe Aspectj [19,18].

- Réduction possible des erreurs de programmeur.
- Système à évolution localisée. C'est à dire que moins des modules sont affectés quand un module est altéré.
- Permettre aux développeurs de logiciels de réagir facilement à des changements imprévus, tout en favorisant la réutilisation des composants logiciels déjà conçu et testés.

II.6 Ingénierie des Besoins Orientés Aspects

L'étape d'ingénierie des besoins revêt une importance capitale de par son influence sur le reste du développement. Ce fait constitue un point de départ pour plusieurs recherches qui visent à améliorer la séparation des préoccupations au niveau des besoins.

Souvent les besoins sont éparpillés et enchevêtrés avec d'autres besoins. La solution à ce problème réside dans la séparation des besoins non fonctionnels qui ont un impact sur plusieurs autres besoins, cas de la sécurité, de la performance et dans la spécification du comment ce besoin contraint-il et affecte les autres. La figure II.5 suivante illustre l'utilisation des aspects pour éliminer l'éparpillement et l'enchevêtrement dans les besoins.

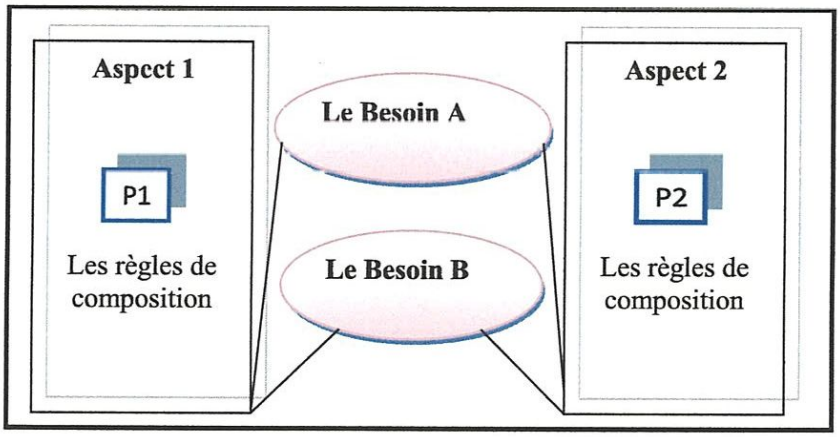

Figure II.5: *Séparation pour une bonne modularité*

Capturer les préoccupations transversales tôt dans le cycle de vie d'un logiciel permet de :

- Améliorer la qualité de produit.
- Améliorer la gestion de l'impact de changement (réduire le coût de l'adaptation, l'évolution et la maintenance).
- Améliorer la consistance des besoins.

Dans les approches de séparation des préoccupations, certaines activités sont communes et présentes dans toutes les approches, on cite :

L'identification: Le but de cette activité est d'extraire les unités de décomposition en distinguant quelles sont les préoccupations de base et quelles sont les préoccupations transversales. Dans un ensemble des besoins existant, il est peu probable que les aspects présents soient distingués clairement. Les documents des besoins peuvent décrire une multitude d'aspects. Lors de l'identification des aspects dans les besoins, on localise premièrement les termes « Aspect » [42], qui sont généralement des attributs de qualité (comme la sécurité, la consistance des données, ou le temps de réponse). L'ingénieur des besoins peut identifier ces termes en utilisant des techniques simples de recherche ou des outils faits spécialement pour l'analyse des besoins orientés aspects, tel que EA-Miner. Deuxièmement, on étudie « l'impact des besoins » qui décrit l'influence d'une préoccupation sur une autre. Et enfin, on s'intéresse à l'éparpillement des préoccupations. Ces dernières sont soit des termes, des concepts, ou des comportements qui apparaissent dans de multiples besoins. Il est possible que certaines préoccupations ne soient pas capturées comme des aspects.

La composition : pour simplifier comment les préoccupations sont composés, l'ingénieur des besoins peut composer les besoins en utilisant des techniques de composition comme celle décrites dans [43] Le but de cette activité est de composer toutes les préoccupations, c'est-à-dire fusionner les préoccupations de base et les préoccupations transversales (les aspects), pour obtenir le système final.

Résolution de conflit : Cette étape inclue l'analyse de l'aspect modulaire des préoccupations autrement dit vérifier l'influence mutuelle des différents préoccupations pour arriver à des compromis (trade-offs) et pour identifier les conflits et les incohérences.

II.7 Conclusion

Dans ce chapitre, nous avons présenté Les concepts de base de l'AOSD, pourquoi l'orienté aspect est nécessaire dans le développement logiciel et comment il contribue à l'amélioration des processus de développement modernes. Comme il est préférable d'identifier les aspects tôt dans le cycle de développement on a présenté une vue globale sur l'ingénierie des besoins orientés aspects, Dans le prochain chapitre, nous présentons notre contribution.

CHAPITRE III :

CONCEPTION

III.1 Introduction

Après avoir passé en revue les différents concepts théoriques de l'orientée aspect et les différentes notions concernant les approches de développement itératifs ainsi que la traçabilité des besoins, nous œuvrons dans ce chapitre à exploiter ces connaissances pour construire notre propre contribution.

Dans ce chapitre, nous présentons notre contribution, ensuite, nous décrivons notre approche en présentant les étapes qui la constituent et nous présentons le modèle de traçabilité proposé pour suivre un projet XP.

III.2 Notre contribution

La clé de succès de l'approche XP pour la construction de systèmes évolutifs dépend de la manière avec laquelle les besoins évolutifs sont extraits et structurés efficacement par les ingénieurs du logiciel.

L'ingénierie des besoins orienté aspect à son tour traite le problème des besoins transversaux et des concepts de l'orienté aspect pouvant être utilisés pour l'adresser.

Notre travail est divisé on deux partie: dans la première partie on s'intéresse à proposé une approche afin d'intégrer les aspects au niveau des besoins dans l'approche de développement XP, concernant la deuxième partie, on propose un modèle de traçabilité qui permet de suivre les différents artéfacts du système.

III.2.1 l'approche proposée

L'objectif à atteindre de l'approche proposée est d'intégrer les concepts de l'ingénierie des besoins orienté aspect dans le processus XP.

Par la prise en compte de la nature transversale de certains besoins des clients, la combinaison des deux approches permette d'éliminer l'enchevêtrement et l'éparpillement au niveau des besoins ce qui résulte en une bonne modularité dans les étapes ultérieures. Cela réduit l'effort de compréhension et de modification et par conséquent diminue le temps nécessaire à la conception et l'implémentation.

La figure III.1 illustre une vue globale de cette intégration.

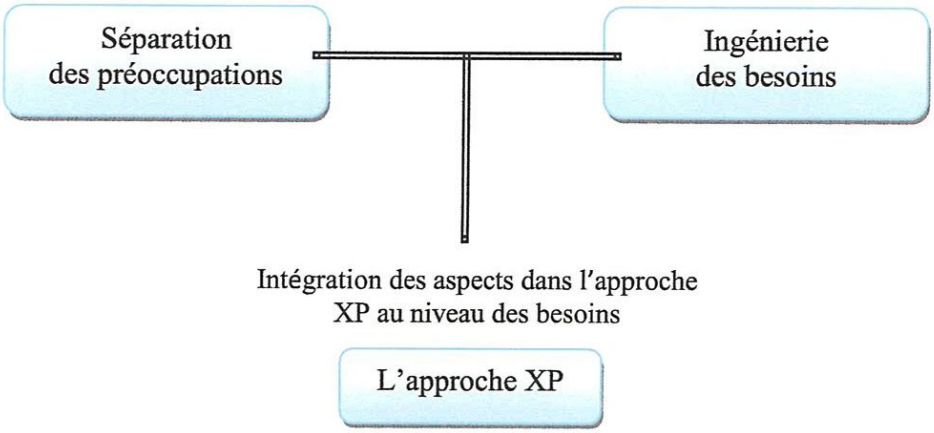

Figure III.1 *Intégration de l'orienté aspect et l'XP au niveau des besoins*

L'étape des besoins joue un rôle très important et est considérée comme la phase responsable de la défaillance des projets. Ceci est vrai aussi pour le projet XP.

Ce travail se résume à fournir une nouvelle approche qui se base sur l'approche XP et l'ingénieric des besoins orientées aspects afin de séparer les préoccupations au niveau des besoins. Il s'agit d'incorporer les concepts de l'ingénierie des besoins orientée aspects (AORE) dans le processus XP.

Notre démarche se base sur l'approche itérative. Une itération est une période fixe de temps durant laquelle les développeurs travaillent sur un projet. Avant que ces développeurs ne commencent le travail, les itérations sont créées et planifiées.

Une description générale d'une itération est nécessaire pour identifier les buts, les risques et avoir une estimation de l'effort requis pour compléter l'itération. Chaque itération consiste en l'ensemble d'histoires utilisateur qui lui sont assignées.

La figure III.2 illustre les différentes étapes de l'approche proposée.

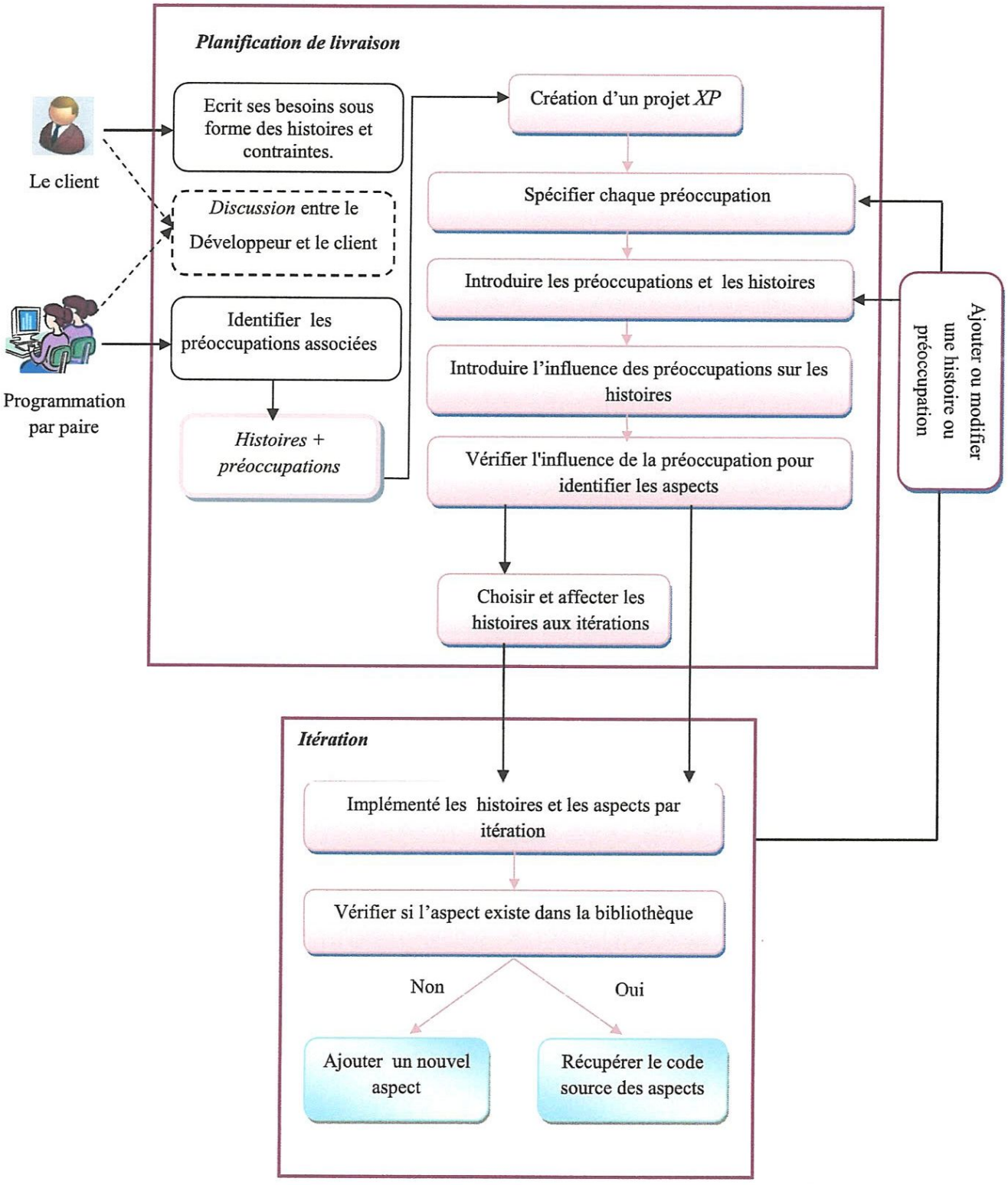

Figure III.2 *Extreme Programming Orienté Aspect*

Au départ le client écrit ses besoins sous forme des histoires et contraintes puis les clients et l'équipe de développement se réunissent pour discuter les fonctionnalités principales à atteindre par le projet. Les développeurs ne peuvent décider si tel ou tel besoin a une valeur importante pour les clients. Seuls ces derniers ont la connaissance nécessaire sur le produit pour en décider.

Les fonctionnalités sont écrites par les clients avec leur langage, et non celui des développeurs, ces clients sont dans la bonne position pour décrire le comportement souhaité de leur produit. Leurs besoins sont écrits et discutés ultérieurement avec les développeurs pour dégager les détails ainsi que pour permettre aux développeurs de mieux comprendre les objectifs.

Chaque histoire est une description d'une fonctionnalité qui devrait être implémentée dans le produit logiciel. Chaque histoire a un nom court qui décrit la fonctionnalité et autres détails.

Après écriture de ses histoires et en se basant sur les estimations fournies par les développeurs, les priorités fournies par le client, la vélocité et le risque, les développeurs et le client peuvent discuter et négocié le choix des histoires qui doivent être complétées dans la livraison suivante. Par la suite, ils écrivent le plan de livraison. Comme ce type de développement est itératif et une itération a une période de temps fixe (d'une à trois semaines), le projet est décomposer, les histoires de haute priorité sont affectées à la première itération pour être sure qu'elles seront complétées.

Après ce plan de livraison, les développeurs ont un nombre d'itérations à compléter avant la livraison du produit. Les développeurs travaillent sur chaque itération séparément, ils peuvent ajouter, raffiner des besoins, etc.

Vu qu'une livraison doit se terminer après un déroulement de quelques itérations, nous présentons notre intégration en deux temps: une itération initiale (la première itération dans le projet) et les itérations restantes qui correspondent à l'ensemble des itérations qui suivent l'itération initial et qui seront implémentées toutes de la même façon. Dans la section suivante nous présenterons les étapes de notre intégration.

1/ Identification des préoccupations

La plus part des analystes distinguent entre deux types des besoins: les besoins non fonctionnels et les besoins fonctionnels. Typiquement les besoins fonctionnels se réfèrent à la valeur de production du système c'est-à-dire ce que le système doit faire, alors que les besoins non fonctionnels représentent la qualité et les contraintes que le système résultant doit satisfaire.

Parmi les besoins non fonctionnel, on trouve

- La réutilisation
- La maintenance
- La persistance
- La sécurité
- L'audit
- Le temps de réponse

Dans notre travail, nous considérons que les besoins non fonctionnels (préoccupations) peuvent être représentés comme des contraintes sur le comportement d'un système.

Les préoccupations sont identifiées de deux façons :

1- Les préoccupations qui sont déjà mentionnées par le client sous forme de contraintes.
2- Les préoccupations qui sont identifiées par une analyse des histoires identifiées précédemment. Ces préoccupations sont invisibles pour le client et sont reliées à des histoires et sont difficiles à exprimer par le client.

A ce stade on aura un ensemble d'histoires et préoccupations.

L'étape qui suit est la création du projet, le développeur va créer son nouveau projet, chaque projet est définit par les attributs suivants:

- ✓ Nom projet
- ✓ Description du projet
- ✓ Nombre des itérations
- ✓ Nombre des histoires

2/ Spécification des préoccupations

Les préoccupations du système pour l'itération courante sont déjà identifiées dans l'étape précédente en se basant sur l'approche décrite dans [44], nous spécifions ces préoccupations par le modèle présenté dans le tableau III.1.

Rubrique	Contenu
Nom	Nom de la préoccupation
Description	Description de la préoccupation
Influence	Liste des histoires d'utilisateur affectées par cette préoccupation
Priorité	La priorité affectée à cette préoccupation
Relation	Le type de relation d'une préoccupation avec d'autres préoccupations

Tableau III.1: *Modèle de la spécification d'une contrainte*

Dans ce qui suit nous expliquons les constituants de ce modèle.

- ***Nom*** : Représente le nom qui désigne la préoccupation.
- ***Description :*** Une description de la préoccupation qui soit brève et concise.
- ***Influence :*** Ce champ contient la liste des histoires utilisateur affectées par cette préoccupation. Il ceci permet de détecter les préoccupations qui sont considérées comme des aspects. Dans ce cas si une préoccupation affecte plusieurs histoires utilisateur alors cette préoccupation sera un aspect.
- ***Priorité :*** C'est la priorité affectée à cette préoccupation, cette priorité est déjà identifiée précédemment par le client et avec négociation de l'équipe. Cette priorité permet d'aider dans l'identification des conflits entre les préoccupations qui seront des aspects. La priorité est indiquée par l'une des valeurs : indispensable, importante ou utile, selon le degré d'importance de cette préoccupation dans le système à développé.
- ***Relation***: la relation ici exprime le type de relation d'une préoccupation avec d'autres préoccupations.

On a définit trois types de relations:

- ✓ Décomposition: La décomposition signifie qu'une préoccupation est décomposable à d'autres préoccupations (sous préoccupations).
- ✓ Précédence: Montre quelle préoccupation peut être implémentée en premier.

- ✓ Contribution: La contribution présente l'influence entre les préoccupations, cette contribution peut être négative (-) vis-à-vis de chacun des autres préoccupations ce qui signifie qu'il y a une influence mutuelle entre les préoccupations comme elle peut être positive (+) ce qui signifie que ces préoccupations sont collaboratives.

3/ Introduire les préoccupations et les histoires

Dans cette étape, le développeur doit introduire l'ensemble des histoires utilisateur ainsi que les préoccupations.

4/ Introduire l'influence des préoccupations sur les histoires

Selon la spécification de chaque préoccupation, le développeur introduire l'influence entre la préoccupation et les histoires utilisateur.

5/ Vérifier l'influence et identifier des aspects

Les préoccupations transversales se trouvent dans des modules multiples. Ces préoccupations sont séparées et modularisés sous forme de modules indépendants appelés aspects. Cette séparation supporte bien l'évolution si les changements n'affectent que l'aspect ou que les histoires de base.

Cette étape consiste à identifier les aspects (les préoccupations transversales) parmi l'ensemble des préoccupations identifiées. Si une préoccupation peut affecter plusieurs histoires utilisateur alors cette préoccupation est considérée comme un aspect. Ceci est fait en tenant compte l'influence. Dans le cas où l'influence concerne plus d'une histoire utilisateur, alors cette préoccupation transverse plusieurs histoires utilisateur et par conséquence elle est séparée comme un aspect.

6/ Affecter les histoires et les aspects aux itérations

Introduire les préoccupations et les histoires selon l'ordre des itérations: à ce niveau les préoccupations et les histoires introduites précédemment seront affectées aux itérations appropriées.

7/ Implémenté les histoires et les aspects par itération

Au niveau de chaque itération un ensemble des histoires et aspect seront implémenté.

8/ Vérifier si l'aspect existe dans la bibliothèque

Dans cette étape le développeur doit vérifier si l'aspect existe dans la bibliothèque d'aspect qu'on a créé. Dans ce cas on est devant deux situations:

- Si l'aspect existe alors on peut récupérer leur code source.
- Sinon le développeur ajoute un nouveau code source d'un aspect dans la bibliothèque.

A la fin ou au cours de chaque itération le client peut modifier ou bien ajouter une nouvelle histoire utilisateur ou préoccupation. Dans ce cas le cycle est répété des l'étape de spécification si la mise à jour concerne la préoccupation si non le cycle est répété Introduire les préoccupations et les histoires

III.2.2 Modèle de traçabilité

Pour suivre un projet informatique il faut garder la trace de ses différents éléments tout au long de cycle de vie. Maintenir les informations permet d'aider le développeur.

Dans le but d'assurer le suivi d'un projet XP, nous avons utilisé un mécanisme qui permet garder la trace et accélérer la recherche de l'information, pour accomplir cet objectif nous avons proposé le modèle de traçabilité présenté par la figure III.3.

La traçabilité en ligne des changements s'agit de l'enregistrement immédiat et automatique des traces.

Afin de visualiser les traces dans un projet on a choisi une représentation sous la forme de graphe, l'interprétation des graphes est plus facile et apporte plus d'informations. Même pour un système complexe, on s'assure que c'est toujours clair.

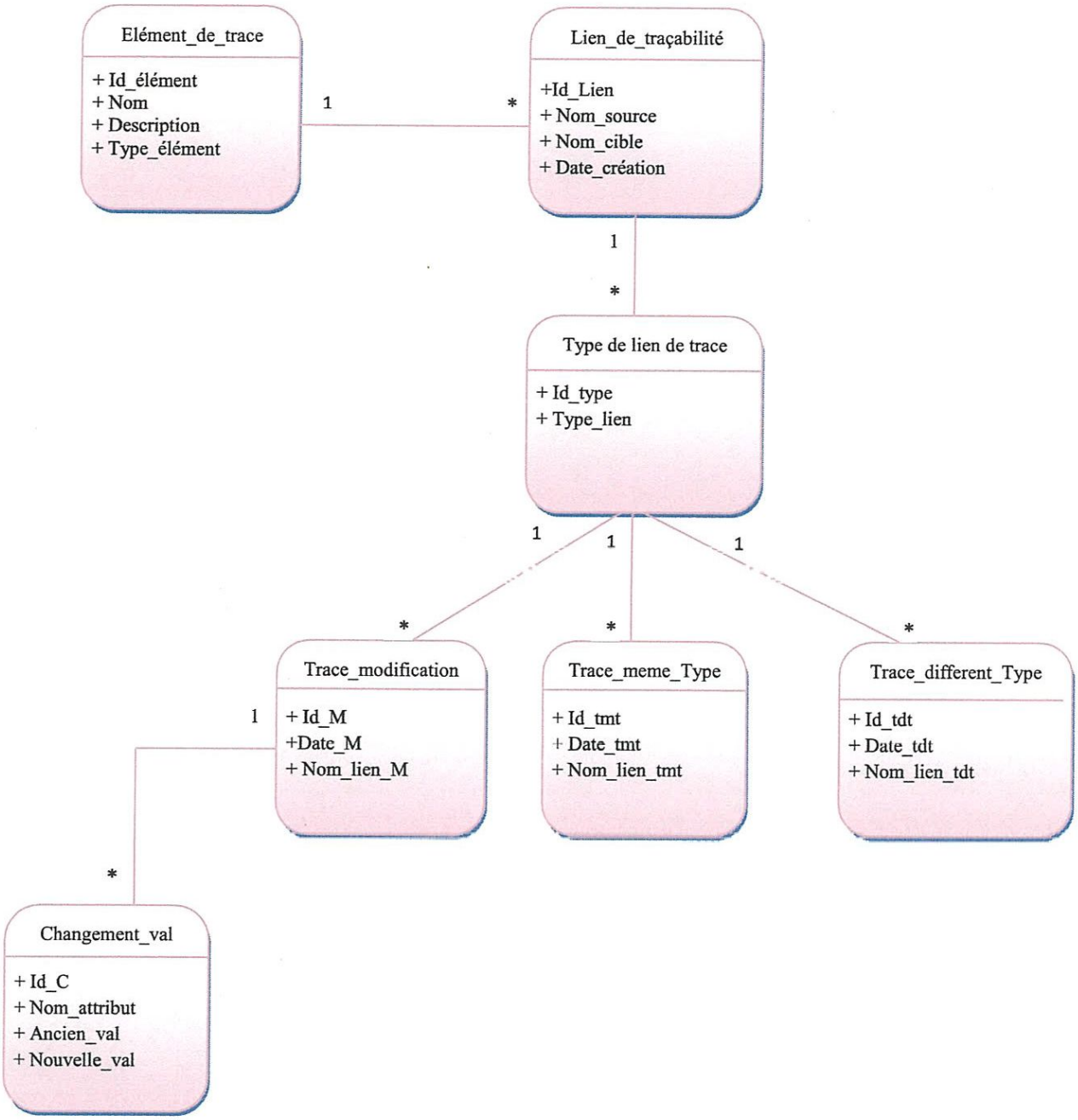

Figure III.3 *Un modèle de traçabilité*

Dans ce qui suit nous expliquons les constituants de ce modèle

1: Elément de trace

Cet élément est un artéfact. Chaque élément est définit par les propriétés suivantes:

- Id_élément: est l'identificateur qui présente l'artéfact.
- Nom: représente le nom qui désigne l'artéfact.
- Description: Une description de l'artefact qui soit brève et concise.
- Type_élément: ce type peut être un projet, une itération, une histoire, une préoccupation ou un aspect.

2: Lien de traçabilité

C'est le lien de traçabilité entre deux artéfacts, ce lien est définit par les attributs suivants :

- Id_Lien: l'identificateur de lien de traçabilité.
- Nom_source: le nom de l'artéfact source.
- Nom_cible: le nom de l'artéfact cible.
- Date_création: la date de création de lien.

3: Type de lien de trace

C'est le type de lien de traçabilité entre les différents artéfacts, ce type est définit par:

- Id_type: L'identificateur de type de lien.
- Type_lien: ce type peut être modification, trace pour les artéfacts de même type ou les artéfacts de types différents.

Il existe dans notre modèle trois types de lien de traçabilité: *trace de modification*, *trace de même Type* et *trace de type différent.*

Dans ce qui suit on va expliquer chaque type:

Trace_modification

Lien de trace qui présente le lien de l'artéfact à lui-même, c'est-à-dire les différentes mises à jour concernant l'artéfact lui même.

Ce type est définit par les propriétés suivantes:

- Id_M: l'identificateur de la trace de modification.
- Date_M: date de modification de l'artéfact.
- Nom_lien_M: le nom de lien qui peut être un lien d'évolution.

Trace_meme_Type

Lien de trace entre deux artéfacts de même type.

- Id_tmt: l'identificateur de l'artéfact.
- Date_tmt: Date de création de la trace.
- Nom_lien_tmt: Nom de lien de la trace qui peut être:

  Décomposition: un artéfact peut être décomposé on sous artéfacts.

  Contribution: la contribution peut être (+) ou (-) entre les préoccupations.

  Précédence: un artéfact est implémenté avant un autre artéfact.

Trace_different_Type

Lien de trace entre deux types différents d'artéfacts.

- Id_tdt: l'identificateur de trace pour les artéfacts de type différents.
- Date_tdt: Date de création de la trace.
- Nom_lien_tdt: Nom de lien de la trace qui peut être :

  Décomposition: Un artéfact peut être décomposé on sous artéfacts.

  Influence: Ce lien présente l'affectation d'un artéfact par un autre par. exemple entre préoccupation et histoire (une préoccupation peut affecter une histoire).

  Inclusion: Un artéfact est inclut dans un autre artéfact par exemple un aspect est inclut dans une préoccupation.

Changement_val

" Changement_val " est défini lorsqu'une valeur d'un attribut est modifiée.

- Id_C : l'identificateur de changement.
- Nom_attribut: Nom de la propriété.
- Ancien_val: l'ancienne valeur de l'attribut.
- Nouvelle_val: la nouvelle valeur de l'attribut.

III.3 Conclusion

Dans ce chapitre nous avons présenté notre démarche d'intégration de la notion de séparation des préoccupations dans une approche Extreme programming et cela au niveau des besoins aussi nous avons introduire le mécanisme de traçabilité afin de suivre les différents artéfacts d'un projet XP. Dans cette démarche, nous avons montré les différentes étapes de la démarche. L'implémentation de cette dernière est exposée dans le chapitre suivant.

CHAPITRE IV :
IMPLÉMENTATION

IV.1 Introduction

La réalisation d'un projet en informatique implique deux phases principales, la conception et l'implémentation. Cette dernière est la phase finale d'élaboration d'un système qui permet au matériel, aux logiciels et aux procédures d'entrer en fonction.

Pour bien concevoir et réaliser notre application, celle-ci repose tout d'abord sur la présentation de l'environnement et les conditions de travail, nous commençons tout d'abord par présenter les outils utilisés à savoir SGBD, IDE, … par la suite nous décrivons la démarche suivie dans la phase d'implémentation et on termine par quelques extraits d'exécution de notre travaille.

IV.2 Environnement de la machine

- Processeur : Intel® Core™ i3.
- Capacité de mémoire RAM : 4GB.
- Vitesse d'horloge : 2.27 GHz.
- Type de système : Système d'exploitation 32bits.
- Système d'exploitation: windows8.

IV.3 Présentation des outils de développement

Dans cette section nous présentons les différentes technologies utilisées pour développer notre application, notre choix est focalisé sur quatre outils logiciels :

- Microsoft SQL Server pour la création et manipulation des bases de données.
- Java langage de programmation informatique orienté objet.
- Eclipse SDK -java-environnement de développement fondé sur le langage Java.
- AspectJ est intégré sur eclipse SDK-java pour le développement orienté aspect.

IV.3.1 Microsoft SQL Server

Microsoft SQL Server est un Système de gestion de base de données (SGBD) relationnel et transactionnel développé et commercialisé par Microsoft [45].

Il permet de stocker des données sur une base et de gérer ces données en les modifiant et en les mettant à jour. Il permet aussi de définir des relations entre les tables en assurant

l'intégrité des données qui sont stockées. Ces relations peuvent être utilisées pour modifier ou supprimer en chaîne des enregistrements liés.

Microsoft SQL Server utilise le langage T-SQL (Transact-SQL) pour ses requêtes, c'est une implémentation de SQL qui prend en charge les procédures stockées et les déclencheurs (trigger). Pour les transferts de données, il utilise le format TDS (Tabular Data Stream).
La première version est sortie en 1989 sur les plateformes Unix et OS/2 et, depuis, Microsoft a porté ce système de base de données sous Microsoft Windows. Il est uniquement supporté sur ce système.

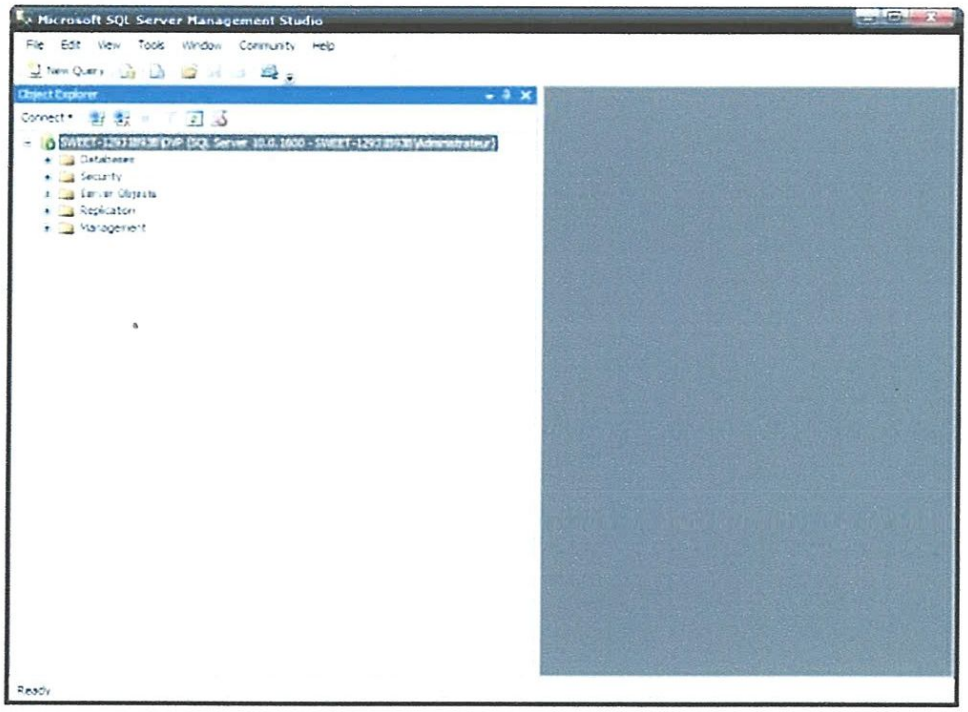

Figure IV.1 : *Interface de Microsoft SQL Server*

IV.3.2 Java

Java cst un langage de programmation récent (les premières versions datent de 1995) développé par Sun MicroSystems. Il est fortement inspiré des langages C et C++.

Comme C++, Java fait partie de la « grande famille » des langages orientés objets. Il répond donc aux trois principes fondamentaux de l'approche orientée objet (POO): l'encapsulation, le polymorphisme et l'héritage [46].

IV.3.2.1 Historique

Le langage Java trouve ses origines dans les années 1990. A cette époque, quelques ingénieurs (innovateurs) de SUN Microsystems commencent à parler d'un projet d'environnement indépendant du hardware pouvant facilement permettre la programmation d'appareil aussi variés que les téléviseurs, les magnétoscopes etc ... James Grosling (SUN Microsystems) développe un premier langage, Oak, permettant de programmer dans cet environnement. En 1992, tout est prêt pour envahir le marché avec cette nouvelle technologie mais la tentative se solde par un échec [46].

Bill Joy (co-fondateur de SUN Microsystems) sauve malgré tout le projet. Devant la montée en puissance de l'Internet, il lui semble intéressant d'insister sur l'élaboration d'un langage indépendant des plates-formes et des environnements (les principaux problèmes rencontrés sur l'Internet étant liés à l'hétérogénéité des machines et des logiciels utilisés).

Dès lors tout s'accélère. Oak est renommé (en 1995) Java et soumis à la communauté Internet grandissante [46]. Une machine virtuelle, un compilateur ainsi que de nombreuses spécifications sont données gratuitement et Java entame une conquête fulgurante. Aujourd'hui, après de nombreuses améliorations (parfois modifications) Java n'est plus uniquement une solution liée à l'Internet. De plus en plus de sociétés (ou de particuliers) utilisent ce langage pour l'ensemble de leurs développements (applications classiques, interfaces homme-machine, applications réseaux …).

IV.3.2.2 Les principales raisons du succès de Java

Java a rapidement intéressé les développeurs pour quatre raisons principales [46] :

- C'est un langage orienté objet dérivé du C, mais plus simple à utiliser et plus « pur » que le C++. On entend par « pur » le fait qu'en Java, on ne peut faire que de la programmation orienté objet contrairement au C++ qui reste un langage hybride, c'est-à-dire autorisant plusieurs styles de programmation. C++ est hybride pour assurer une compatibilité avec le C.
- Il est doté, en standard, de bibliothèques de classes très riches comprenant la gestion des interfaces graphiques (fenêtres, boites de dialogue, contrôles, menus, graphisme), la programmation multi-threads (multitâches), la gestion des exceptions, les accès aux

fichiers et au réseau … L'utilisation de ces bibliothèques facilitent grandement la tâche du programmeur lors de la construction d'applications complexes.

- Il est doté, en standard, d'un mécanisme de gestions des erreurs (les exceptions) très utile et très performant. Ce mécanisme, inexistant en C, existe en C++ sous la forme d'une extension au langage beaucoup moins simple à utiliser qu'en Java.

IV.3.3 Eclipse SDK

Eclipse est un IDE, *Integrated Development Environment* (EDI environnement de développement intégré en français), c'est-à-dire un logiciel qui simplifie la programmation en proposant un certain nombre de raccourcis et d'aide à la programmation. Il est développé par IBM, est gratuit et disponible pour la plupart des systèmes d'exploitation.

Au fur et à mesure que vous programmez, eclipse compile automatiquement le code que vous écrivez, en soulignant en rouge ou jaune les problème qu'il décèle. Il souligne en rouge les parties du programme qui ne compilent pas, et en jaune les parties qui compilent mais peuvent éventuellement poser problème (on dit qu'eclipse lève un avertissement, ou *warning* en anglais). Pendant l'écriture du code, cela peut sembler un peu déroutant au début, puisque tant que la ligne de code n'est pas terminée (en gros jusqu'au point-virgule), eclipse indique une erreur dans le code.

Il est déconseillé de continuer d'écrire le programme quand il contient des erreurs, car eclipse est dans ce cas moins performant pour vous aider à écrire le programme [47].

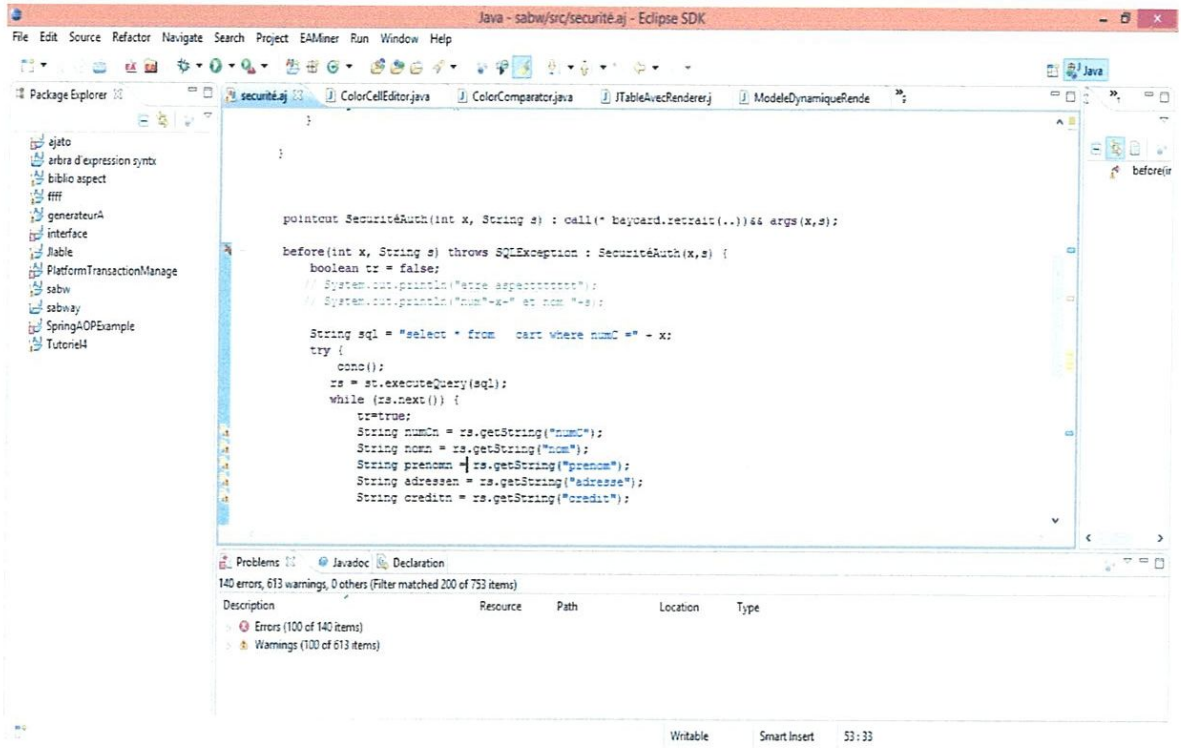

Figure IV.2 : *Interface d'Eclipse SDK*

IV. 3.4 AspectJ

AspectJ est Le langage de POA le plus utilisé, il ajoute des mots clef au langage Java. Le langage Java est actuellement le plus utilisé pour la programmation par aspects, et les deux projets les plus avancés sont AspectJ et JAC. Il s'agit cn fait dc tisscurs pour l'aspect avec un environnement de programmation [48].

AspectJ ajoute les concepts suivants au langage Java:

- ***Joinpoints*** : Ce sont des points particuliers dans l'exécution d'un program me. Un joinpoint peut par exemple défini run appel à une méthode d'une classe.
- ***Pointcuts*** : Ce sont les constructeurs qui désignent les joinpoints, et qui permettent de collecter un certain nombre d'informations à ces points précis.
- ***Advice***: Portions de code qui sont exécutées lorsque certaine conditions est rencontrée. Par exemple, un advice peut logger un message avant d'exécuter un joinpoint.

Les pointcuts et les advices forment ensemble les règles de tramage. Les aspects, un concept proche de celui de classes, regroupent les pointcuts et les advices pour former une unité qui se recoupe. Les pointcuts spécifient les point d'exécution ainsi que le contexte de

tramage, tandis que les advices spécifient les actions, les opérations, où les décisions qui seront prises en ces points.

IV.3.4.1 L'intégration AspectJ dans eclipse SDK

Dans cette section on va représenter l'intégration de l'AspectJ sous eclipse SDK, pour ce la il faut premièrement télécharger le fichier JAR de l'AspectJ d'après le site officiel d'eclipse puis il faut suivre les étapes suivantes:

- Dans la barre de menu, on choisir **help**.
- Pour la deuxième étape on va choisir **Install New Software**, une fenêtre s'affichera (Figure IV.3).

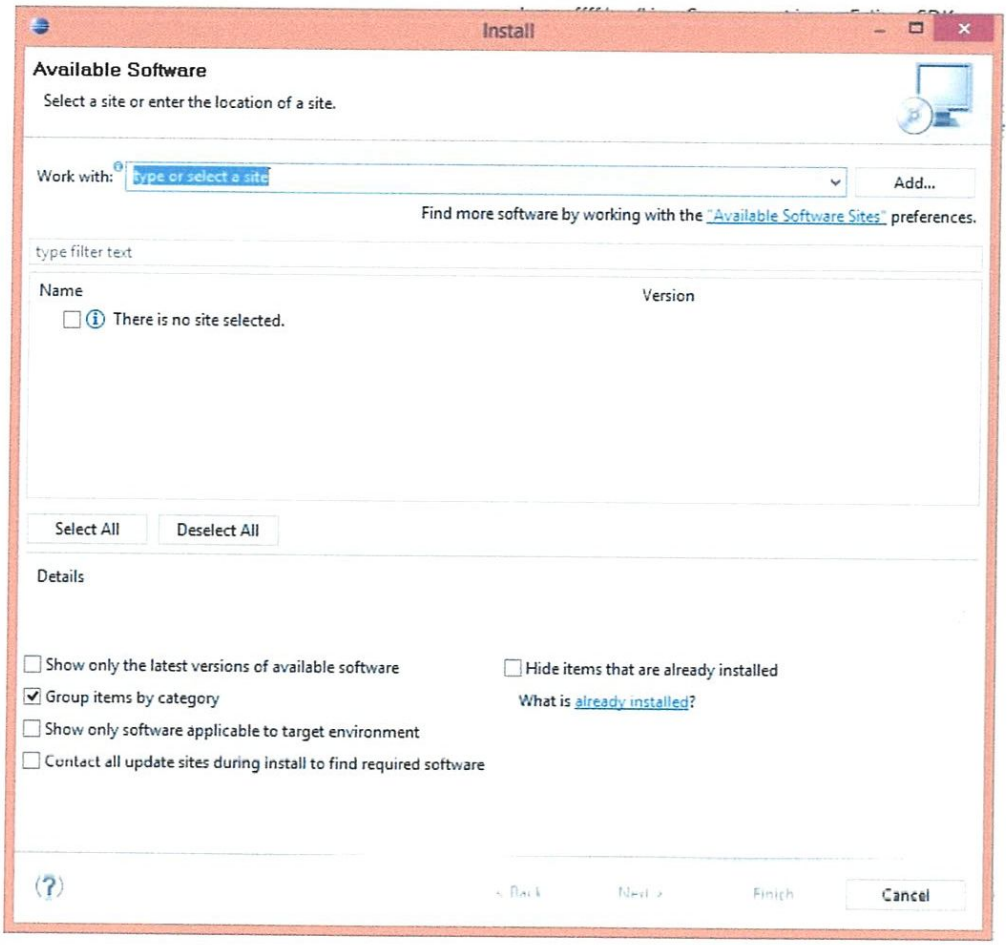

***Figure IV.3**: Fenêtre pour installer un nouveau logiciel*

- En suite on va choisir le bouton **Add**, puis cliquer sur **Local** et choisir le fichier AspectJ.jar téléchargé précédemment. l'installation est terminée on cliquant sur **finish**.

IV.4 Présentation du système

Après avoir présenté l'environnement de développement utilisé pour l'implémentation de notre projet, cette section est consacrée à la représentation des différentes fonctionnalités de notre système.

Au lancement de notre application, la fenêtre principale est présentée (Figure IV.4).

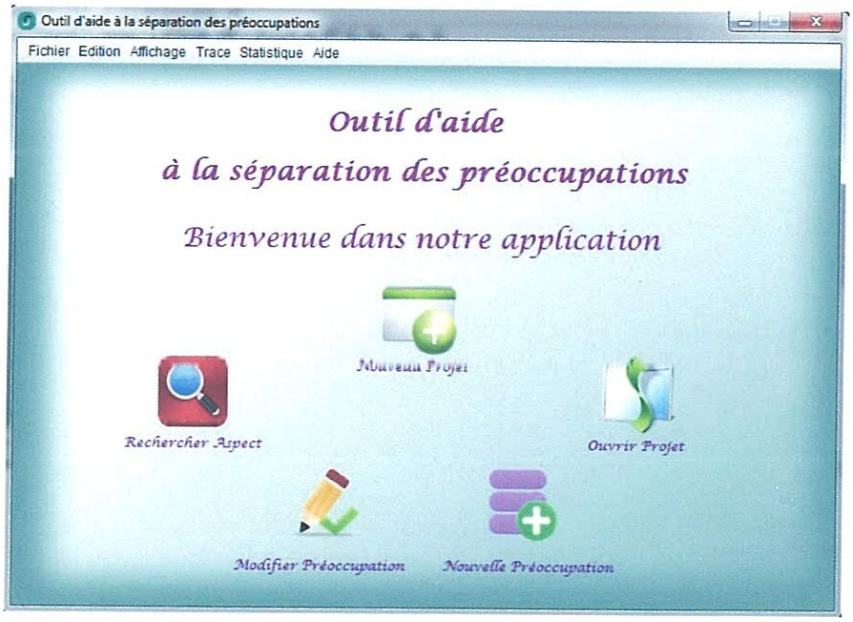

Figure IV.4 : Fenêtre principale

Pour commencer, le développeur peut cliquer sur nouveau projet, une fenêtre s'affichera. Cette dernière lui permettre de créer son projet.

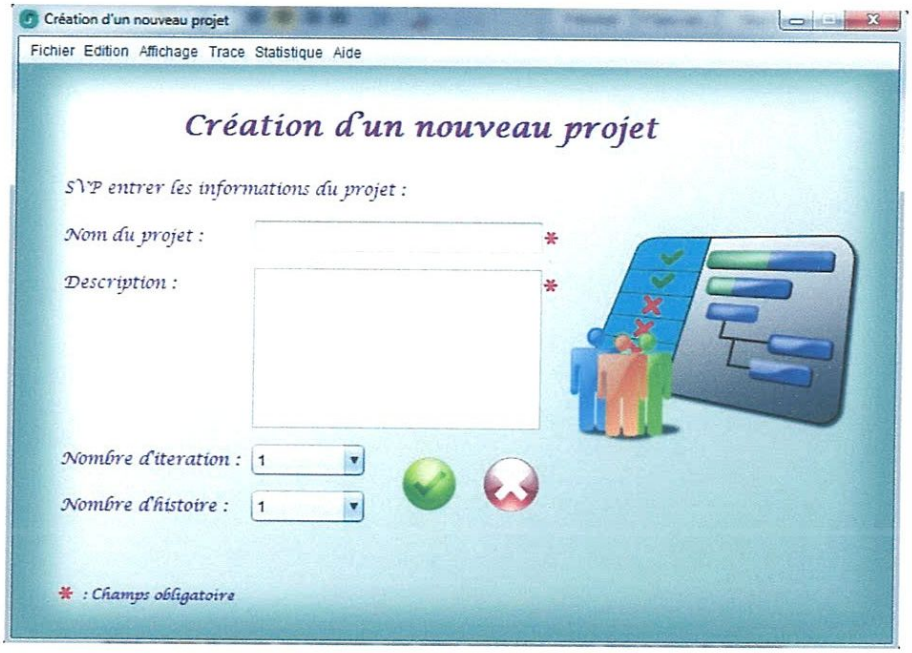

Figure IV.5 : *Fenêtre pour ajouter un nouveau projet*

Après l'ajout de projet, les itérations sont créées automatiquement dans la base de données.

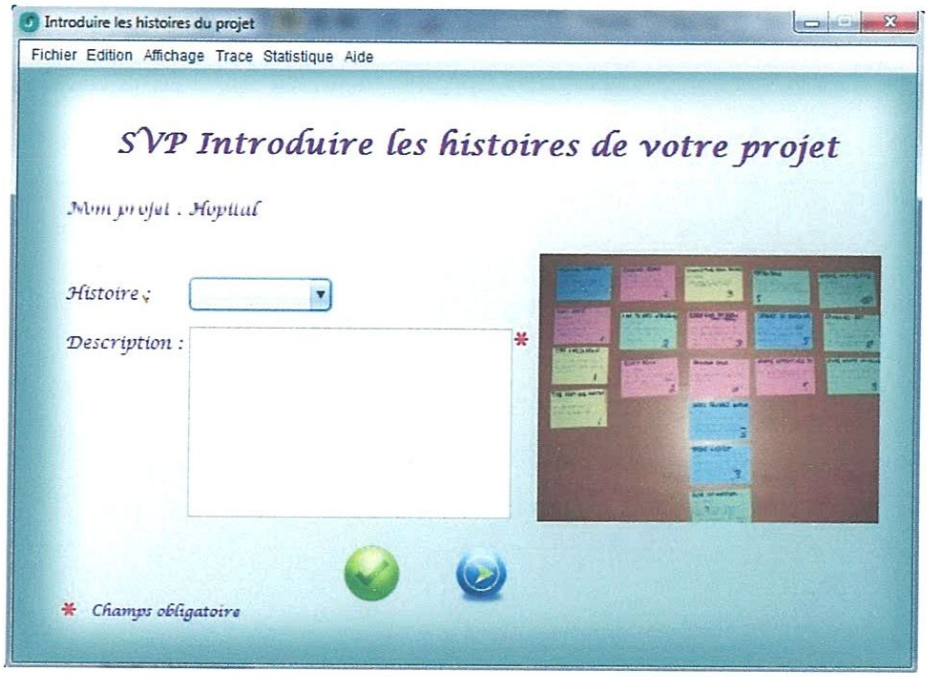

Figure IV.6 : *Fenêtre pour introduire les histoires d'un projet*

Dans la fenêtre présentée par la figure IV.6, le développeur peut introduire les histoires de projet à développer.

La fenêtre ci-dessous permette d'introduire les préoccupations identifiées.

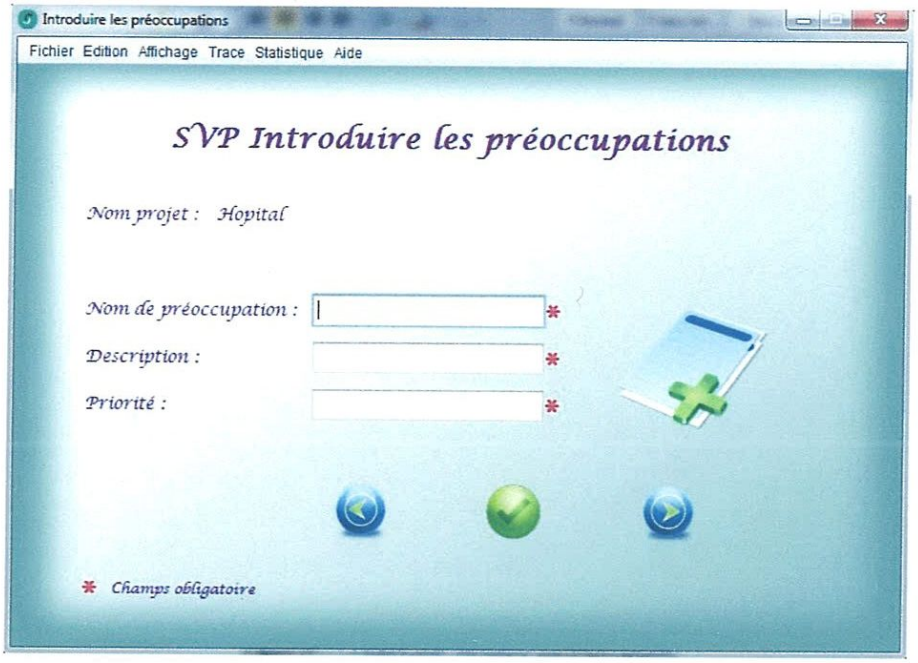

Figure IV.7 : *Fenêtre pour introduire une préoccupation*

Ensuite une fenêtre s'affichera afin de remplir l'influence et cela par le choix des histoires affectées pour chaque préoccupation.

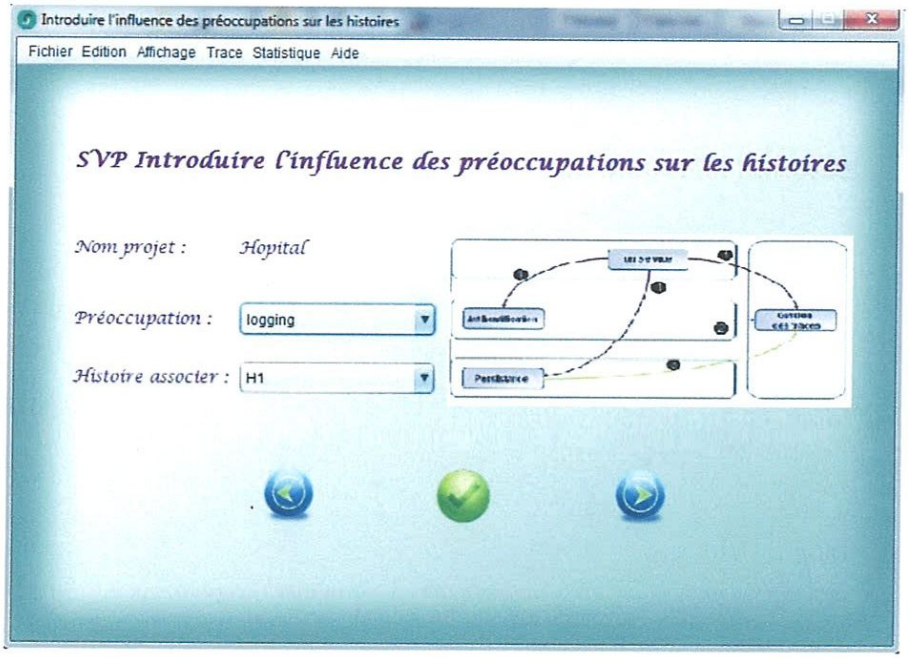

Figure IV.8: *Fenetre pour introduire l'influence*

La figure IV.9 présente la fenêtre qui permet au développeur de choisir le type de relation entre deux préoccupations ainsi que le développeur peut introduire des relations manuelles.

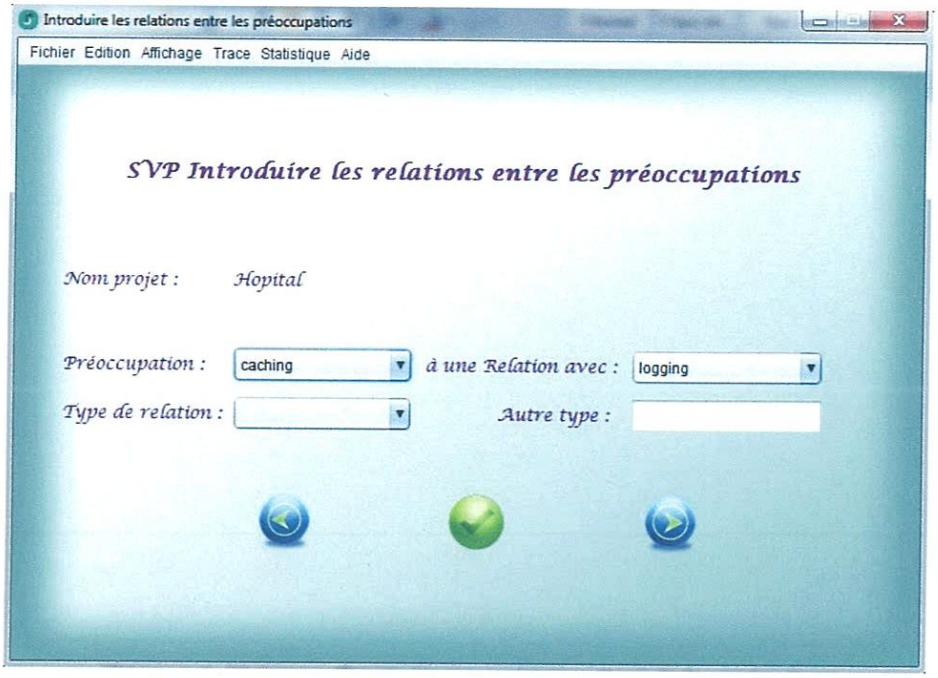

Figure IV.9 : *Fenêtre pour introduire les relations entre les préoccupations*

Maintenant, le développeur affecté les histoires aux itérations dans la fenêtre représentée par la figure VI.10.

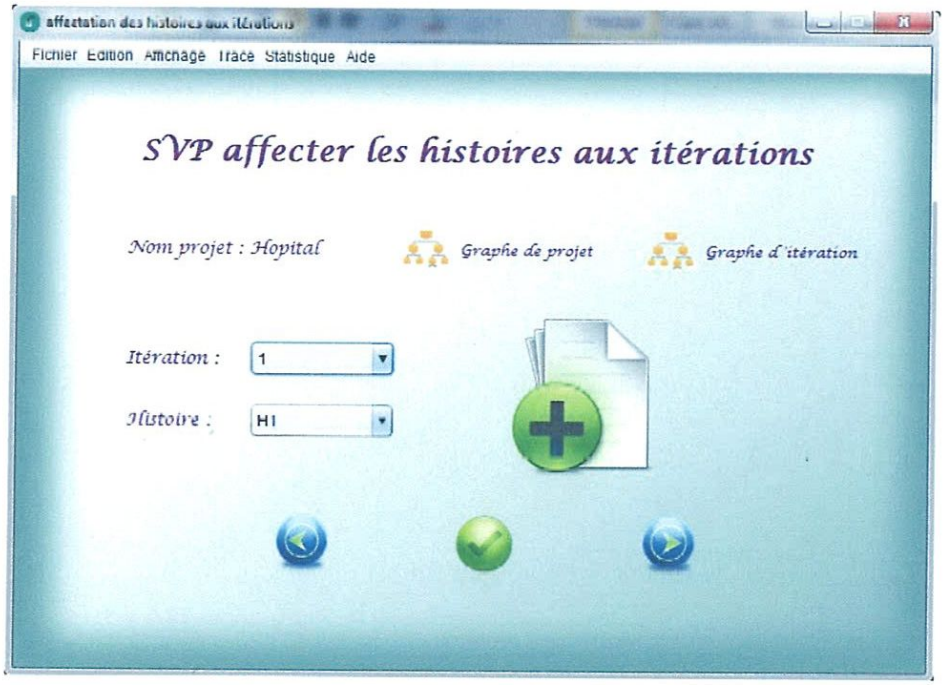

Figure VI.10 : *Fenêtre d'affectation des histoires*

Le développeur peut récupérer le code source d'un aspect s'il existe dans la base de données, Celle-ci est représentée dans la figure VI.11.

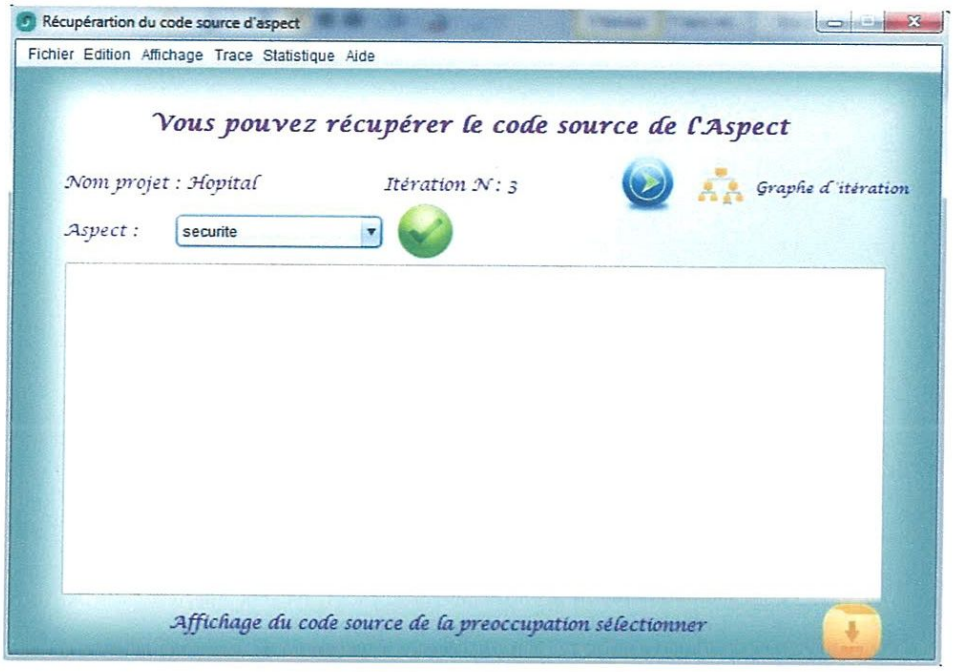

Figure IV.11 : *Fenêtre de téléchargement de code source de préoccupation*

Si le code source d'aspect n'existe pas, le développeur peut ajouter un aspect dans la fenêtre représentée par la figure IV.12.

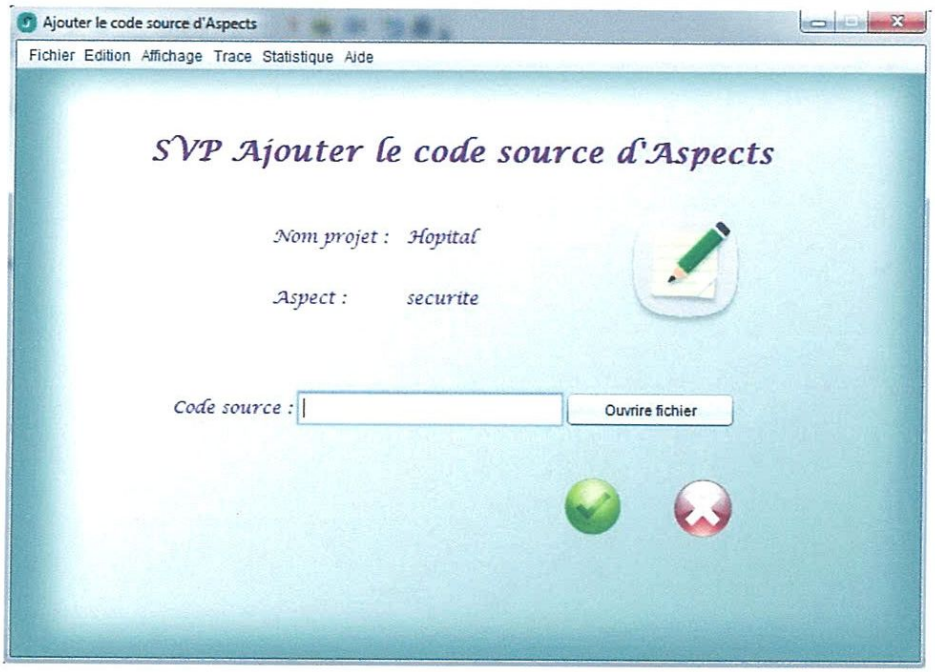

Figure IV.12 : *Fenêtre pour ajouter un aspect*

Le développeur peut voir la trace de leur projet en cliquant sur graphe projet, cela est présenté dans la figure VI.13.

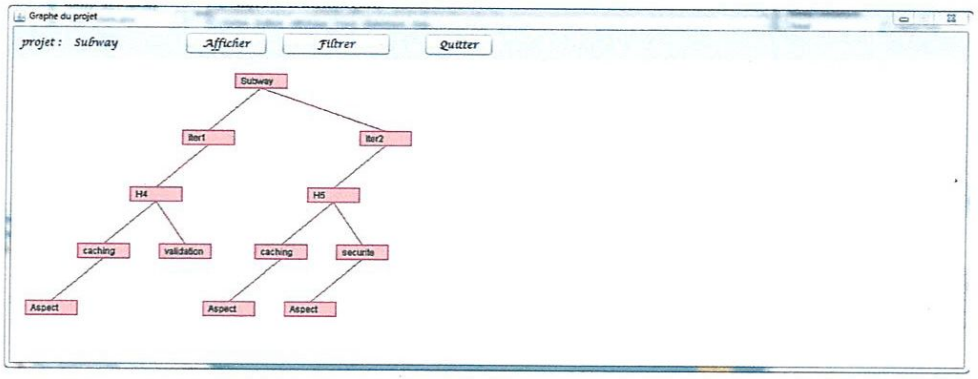

Figure IV.13: *Fenêtre d'affichage de graphe de traçabilité d'un projet*

IV.5 Conclusion

Dans le présent chapitre nous avons aboutis à un système qui prend en charge d'une façon générale la majorité des fonctions définies dans la partie conception et analyse de notre système. Ainsi nous avons développé un système qui a comme objectif d'aider le développeur dans la séparation et la traçabilité. Enfin nous avons présenté le mode d'utilisation de cette application.

CONCLUSION ET PERSPECTIVE

Toute application doit être testée afin de garantir à l'utilisateur final une qualité de service qui soit satisfaisante. La programmation orientée aspect est une nouvelle méthodologie qui permet de séparer les préoccupations subsidiaires qui entrecoupent les fonctionnalités principales d'un système. Elle permet la production d'un code métier « pur » découplé du code non-fonctionnel. C'est une solution aux problèmes de mélange et de dispersion du code des propriétés non fonctionnelles rencontrés avec la programmation par objets, qu'elle consiste à séparer et découpler leurs définitions comme le veut le principe de la séparation des préoccupations.

Ce travail vise à mettre en place un modèle d'ingénierie des besoins orienté aspect intégré à l'approche Extreme progamming. Cette dernière étant l'une des approches les plus efficaces et les plus utilisées dans le contexte du développement itératif en générale et agile en particulier.

Dans un deuxième temps, on a définie un modèle de traçabilité qui permet de garder la trace afin d'aider le développeur d'avoir les informations nécessaires.

Nous résumons les principaux apports de notre travail en ce qui suit:

- La proposition d'un modèle orienté aspect pour l'ingénierie des besoins qui couvre l'identification des aspects et leur réutilisation dans un projet XP. Notre proposition se focalise sur les histoires utilisateur et les contraintes comme un point de départ de cette intégration.
- La contribution par la mise en place d'une nouvelle vision pour les projets XP. Les approches XP classiques appliquent les histoires utilisateur sans prendre en compte l'enchevêtrement et l'éparpillement de certaines d'entre elles. Le modèle proposé repose sur un ensemble d'étapes qui permettent d'éviter l'enchevêtrement et l'éparpillement et par conséquence réduit le temps et l'effort nécessaire pour la conception et l'implémentation ultérieures.
- La proposition d'un modèle de traçabilité qui permet de donner une force supplémentaire à l'XP dans le sens d'aider le développeur de suivre son projet.

Ce travail ouvre la voie vers diverses perspectives de recherche qui se situent sur deux plans : le premier est relatif à l'approfondissement de la recherche réalisée et l'autre est relatif à l'élargissement du domaine de la recherche.

Pour ce qui est de l'approfondissement du travail réalisé, il serait intéressant de développer une approche XP orientée aspect dans sa totalité allant des besoins à l'implémentation.

En ce qui concerne l'élargissement du domaine de recherche nous souhaitons intégrer les concepts de l'orienté aspect dans d'autres approches de développements agiles puis opérer des comparaisons afin de dégager, éventuellement, une nouvelle approche agile orientée aspect.

[1] B. Boehm software Engineering Economics Prentice-Hall, 1981.

[2] Audibert, UML 2.0, Institut Universitaire de Technologie de Villetaneuse – Département Informatique, France, web : http ://www-lipn.univ-paris13.fr/audibert/pages/enseignement/cours.htm.

[3] N. Belkhiter, Processus et méthodologies de conception. Notes de cours "Ingénierie des Interfaces personne-machine (IFT-65119)". Département d'informatique, Université Laval, Sainte-Foy, 2002.

[4] Professeur Jean-Pierre FOURNIER, http://www.infeig.unige.ch/support/se/lect/gl/models/node12.html

[5] Jean Tabaka.2008.gestion de projet vers les méthodes agiles. P42.

[6] Shore, James, et Shane Warden. 2007. The art ofagile development: O'Reilly.

[7] Memoire de HOUDA BAGANE. Mars 2011. ANALYSE DES PRINCIPES DU GÉNIE LOGICIEL AU NIVEAU DU DÉVELOPPEMENT AGILE. P 31.

[8] J. Highsmith, Adaptive Software Development, Dorset House, New York, NY, 2000.

[9] K. Beck, Extreme programming explained: Embrace change, Addison Wesley, September 1999. [10]- J. Stapleton., Dynamic Systems Development Method: The method in practice, Addison-Wesley, 1997.

[10] J. Stapleton., Dynamic Systems Development Method: The method in practice, Addison-Wesley, 1997.

[11] A. Cockburn, Crystal methodologies: The Cooperative Game, Addison-Wesley, 2006.

[12] K. Schwaber, M. Beedle, Scrum: Agile Software Development, Prentice-Hall, 2002.

[13] P. Coad, Lefebvre, E. and J. De Luca, Java Modeling in Color, Prentice-Hall, Englewood Cliffs, NJ, 1999.

[14] K. Beck, Ambracing change with extreme programming. IEEE computer 32(10):70-77, 1999.

[15] J. Haungs, Pair programming on the c3 project, Computer 34 (2): 118-119, 2001.

[16] J. Hunt. Agile software construction, 2006.

[17] K. Beck. Extreme Programming: a gentle introduction. Site http://www.extremeprogramming.org/

[18] R. E. Jeffries. An Agile Software Development Resource, Site http://www.xprogramming.com, 2004.
[20]-J. Brewer, Extreme Programming FAQ. Site http://www.iera.com/techinfo/xpfaq .html, 2001.

[19] J. Brewer, Extreme Programming FAQ. Site http://www.iera.com/techinfo/xpfaq .html, 2001.

[20] Adaption Soft.Test Driven Development: Way Fewer Bugs. Site http://www.adaptionsoft.com/tdd.html.

[21] Stefan Winkler and Jens Pilgrim,A survey of traceability in requirements engineering and model-driven development, Software Systems Modeling 9(2009), no. 4, 529–565.

[22] Gotel, O.C.Z., Finkelstein, A.C.W.: An analysis of the requirements traceability problem. In: 1st IEEE International Requirements Engineering Conference (RE'94) Proceedings, pp. 94–101.IEEE Computer Society, New York (1994).

[23] Pinheiro, F.A.C.: Requirements traceability. In: Sampaio do Prado Leite, J.C., Doorn, J.H. (eds.) Perspectives on Software Requirements, pp. 93–113. Springer, Berlin (2003).

[24] Lee, C., Guadagno, L., Jia, X.: An agile approach to capturing requirements and traceability. In: Proceedings of the 2nd International Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE '03). ACM, New York (2003).

[25] Wieringa, R.: An introduction to requirements traceability. Tech. rep., Institute for Mathematics and Computer Science, Vrije Universiteit, Amsterdam, The Netherlands (1995).

[26] Duan, C., Cleland-Huang, J.: Visualization and analysis in automated trace retrieval. In: 1st International Workshop on Requirements Engineering Visualization (REV'06). IEEE Computer Society, New York (2006).

[27] Card, D.N.: Designing software for producibility. J. Syst. Softw.17(3), 219–225 (1992).

[28] Dahlstedt, Å.G., Persson, A.: Requirements interdependencies:
state of the art and future challenges. In: Engineering and Managing Software Requirements, pp. 95–116. Springer, Berlin (2005).ISBN 978-3-540-25043-2.

[29] Espinoza, A., Alarcon, P.P., Garbajosa, J.: Analyzing and systematizing current traceability schemas. In: Proceedings of the 30th Annual IEEE/NASA Software Engineering Workshop, pp. 21–32. IEEE Computer Society, New York (2006).

[30] Limón, A.E., Garbajosa, J.: The need for a unifying traceability scheme. In: ECMDA Traceability Workshop (ECMDA-TW) 2005 Proceedings, pp. 47–56. Sintef, Trondheim (2005). ISBN 978-82-14-03813-2.

[31] Ramesh, B., Jarke, M.: Toward reference models for requirements traceability. IEEE Trans. Softw. Eng.27(1), 58–93 (2001).

[32] Spanoudakis, G., Zisman, A. Software traceability: a roadmap. In:
Chang, S.K. (ed.) Handbook of Software Engineering and Knowledge Engineering, vol. 3—Recent Advances, pp. 395–428. World Scientific, Singapore (2005). ISBN 978-9-8125-6273-9.

[33] von Knethen, A., Paech, B.: A survey on tracing approaches in practice and research. Research Report 095.01/E, Fraunhofer IESE, Kaiserslautern, Germany (2002).

[34] Ramesh, B., Stubbs, C., Powers, T., Edwards, M.: Requirements traceability: theory and practice. Ann. Softw. Eng.3, 397– 415 (1997).

[35] Brooks F.P., "No Silver Bullet: Essence and Accidents of Software Engineering" IEEE Computer, 20(4), April 1987, p. 10 – 19.

[36] Gray L., "A Comparison of IEEE/EIA 12207, ISO/IEC 12207, J-STD-016,and MIL-STD-498 for Acquirers and Developers" Abelia Corporation, Technical Report, 1999, disponible à http://www.abelia.com/docs/122_016.pdf (accédé le 13/09/2009).

[37] Larman, Craig, et Victor R. Basili. 2003. «Iterative and Incremental Development: A Brief History». IEEE Computer, vol. 36, p. 47-56.

[38] Fowler, Martin. 2005. «The New Methodology». En ligne. <http://martinfowler.eom>.

[39] Memoire de Abdelkrim AMIRAT. 2007. Une Approche Hybride pour la Séparation des Préoccupations avec Résolution de Conflits durant l'Ingénierie des Besoins.

[40] K. Beck. Extreme Programming Explained: Embrace change. Boston: Addison-Wesley, 2000.

[41] I. Jacobson, Pan-Wei Ng, Aspect-Oriented Software Development with Use Cases, Addison Wesley. December 2004

[42] E. Baniassad, P.c.Clements, J.Araujo et A.Moreira, A.Rashid, B.Tekierdogan, Discovering early aspects, January/February IEEE Software 2006.

[43] A. Rashid, A. Moreira, J. Araujo, Modularization and composition of aspectual Requirements, 2nd International Conference on Aspect-Oriented Software Development, ACM. Pages 11-20, Boston, USA, 2003.

[44] A. Moreria, J. Arojo, and I. Brito, Crosscutting quality attributes for requirements engineering,SEKE2002: Fourteenth International Conference on Software Engineering and Knowledge Engineering, Ischia, Italy, 15-19 July 2002.

[45] A. Vimory, Test de Microsoft SQL-Server 2008, page 3, 6 mai 2009.

[46] C. Delannoy, Programmer en java, page 5, 2008.

[47] Site web: http://www.enseignement.polytechnique.fr/informatique/profs/Julien.Cervelle/eclipse/, Juin 2015.

[48] Ferut Térence, Leroy Sébastien," La programmation Orientée Aspects", Juin 2004.