UNIVERSITÉ DU QUÉBEC À MONTRÉAL

ANALYSE DES PRINCIPES DU GÉNIE LOGICIEL AU NIVEAU DU DÉVELOPPEMENT AGILE

MÉMOIRE

PRÉSENTÉ

COMME EXIGENCE PARTIELLE

DE LA MAÎTRISE EN INFORMATIQUE

PAR

HOUDA BAGANE

MARS 2011

REMERCIEMENTS

Je tiens à remercier tout d'abord mon directeur de recherche, Monsieur Guy Tremblay, professeur à l'Université du Québec à Montréal, pour sa disponibilité, sa patience, sa lecture méticuleuse de chacun des chapitres de ce mémoire ainsi que pour ses précieux conseils. Je suis particulièrement reconnaissante pour le temps qu'il m'a si généreusement accordé et la promptitude de ses rétroactions sur mon travail. Ce fut un plaisir de travailler avec lui.

J'aimerais aussi remercier mon codirecteur, Monsieur Normand Séguin, professeur à l'Université du Québec à Montréal, pour m'avoir communiqué son enthousiasme contagieux envers le sujet de recherche. Je lui suis très reconnaissante d'avoir été attentif aux avancés de ma recherche et de m'avoir soutenu lors de la rédaction.

Je désire remercier mes directeurs pour leur support financier (subvention CRSNG du prof. Tremblay et subvention de démarrage de la Faculté des sciences du prof. Séguin), ce qui m'a aussi permis de bénéficier de la bourse FARE de l'UQAM. Je remercie aussi la Mission Universitaire de Tunisie en Amérique du Nord pour la bourse d'exemption des frais majorés qu'elle m'a accordée durant mes études.

Je remercie également Monsieur Hafedh Mili, professeur à l'Université du Québec à Montréal, qui m'a permis de faire partie du laboratoire LATECE (LAboratoire de recherche sur les TEchnologies du Commerce Electronique). Qu'il soit ici assuré de ma profonde gratitude et de mon très grand respect pour son aide professionnel et personnel. Mes remerciements vont également aux professeurs chargés de corriger ce mémoire et à tous mes enseignants de maîtrise, en particulier M. Guy Tremblay, M. Ghislain Lévesque, M. Roger Nkambou, M. Roger Villemaire, M. Abdellatif Obaïd, M. Bévo E. Valery et M. Daniel Dubois. Une pensée très particulière est adressée à Ghizlane El Boussaidi, professeure à l'École de technologie supérieure. Je la remercie pour sa présence dans les moments difficiles. Un remerciement spécial à tous les membres du LATECE, en particulier Romdhane, Wassim, Jean-Sébastien, Rana, Gino et Mélanie.

Ces remerciements ne peuvent se compléter sans remercier : mon père pour sa confiance et ses encouragements qui sont les piliers fondateurs de ce que je suis et de ce que je fais, ma mère pour son soutien discret et constant, mon frère Hazem et mes sœurs Hiba et Haifa. Enfin, un très grand merci à Zied et Wafa pour leur présence dans ma vie.

TABLE DES MATIÈRES

LISTE DES FIGURES

LISTE DES TABLEAUX

RÉSUMÉ

Dans le cadre du développement de logiciel, deux courants de pensée se font concurrence depuis quelques années : le développement de logiciels s'appuyant sur une modélisation plus détaillée et le développement de logiciels basé sur des méthodes agiles qui mettent l'accent sur la production de code opérationnel plutôt que sur la documentation. Chaque courant s'appuie sur ses propres principes. Diverses recherches ont été menées sur les fondements du génie logiciel plus « classique », de façon à en définir les principes. Parmi ces travaux figure la thèse de Normand Séguin, notre codirecteur. En revanche, aucune recherche n'a tenté d'étudier les fondements du développement agile de logiciel.

Au début des années 2000, plusieurs grands noms du développement logiciel se sont réunis, parmi eux Cunningham, Beck, Schwaber, Sutherland et Fowler. Ces derniers ont tenté d'extraire, de leurs approches respectives, quatre valeurs et douze principes communs à tous dans le but de produire un « Manifeste agile ». Le développement agile de logiciel peut se faire à l'aide d'un ensemble de méthodes agiles. Bien que chaque méthode adhère aux valeurs et principes du manifeste, chacune met de l'avant des valeurs, des principes et des pratiques complémentaires.

Ce mémoire présente les résultats d'une étude de 32 énoncés présentés comme étant des « principes agiles » répertoriés dans la littérature. La liste des principes étudiés est formée à partir des douze principes du manifeste agile, cinq principes de l'*eXtreme programming* (*XP*), six de *Scrum* et neuf de *Dynamic System Development Method* (*DSDM*). Notre étude est fondée sur une méthodologie analytique reposant sur une liste de critères pour vérifier si un « principe agile » correspond bien à un principe de génie logiciel. Notre analyse a permis d'identifier 19 principes qui répondent aux critères. Tous les principes retenus sont des propositions prescriptives guidant l'action dans le processus de développement de logiciel.

Mots clés : génie logiciel, développement agile, manifeste agile, méthodes agiles, principe, SWEBOK, eXtreme Programming, Scrum, DSDM.

INTRODUCTION

Au cours de la dernière décennie, le développement de logiciel agile a changé la façon dont le développement de logiciel est perçu aujourd'hui. Les méthodes et les pratiques des approches agiles s'imposent de plus en plus dans le paysage du développement logiciel. C'est probablement parce qu'elles démontrent, dans certaines situations, une forte capacité à livrer un logiciel qui répond aux attentes des clients. En suivant un processus itératif, incrémental et adaptatif, en mettant au premier plan les facteurs humains, le développement agile permet de livrer régulièrement un logiciel utile, fonctionnel et de qualité.

La notion de développement agile de logiciel est née à travers un manifeste signé par 17 auteurs et créateurs de méthodes agiles. Le manifeste se base sur quatre valeurs fondamentales qui se déclinent en douze principes généraux communs à toutes les méthodes agiles. Bien que toutes les méthodes agiles aient des caractéristiques communes, chacune introduit aussi ses propres valeurs, principes et pratiques.

Le présent travail s'inspire de la thèse de doctorat de Normand Séguin (Séguin, 2006) qui avait analysé 308 principes du génie logiciel recensés depuis 1970 à l'aide d'une méthodologie analytique pour ne conserver que 34 principes qui satisfont à des critères d'identification précis. En nous basant sur cette méthodologie, nous analyserons un total de 32 supposés principes énoncés par les tenants du mouvement agile depuis le début des années 2000. Notre but principal est donc de vérifier si les principes agiles publiés étaient vraiment des principes selon les critères d'identification proposés dans la thèse de Normand Séguin (Séguin, 2006).

Pour atteindre ce but, plusieurs tâches doivent être réalisées :

- ✓ Définir les éléments de notre cadre conceptuel;
- ✓ Clarifier ce qu'est le développement agile et dissiper certaines ambiguïtés entourant sa définition, ses méthodes et son vocabulaire;
- ✓ Analyser les principes du manifeste agile selon des critères d'identification précis;
- ✓ Analyser les principes des méthodes *eXtreme Programming*, *Scrum*, *DSDM* selon les mêmes critères d'identification;
- ✓ Présenter une synthèse des résultats obtenus.

À notre connaissance, aucun autre travail de recherche ne s'est intéressé à ce genre d'étude. Les travaux portant sur le développement agile de logiciel se limitent tous à expliquer les principes agiles. C'est ce qui distingue notre recherche.

Afin d'atteindre nos objectifs, la structure du mémoire suivante a été adoptée. Dans le premier chapitre, nous allons présenter et définir les concepts clés du développement agile, ces concepts nous paraissant fondamentaux dans la compréhension de ce travail. Nous clarifierons, dans le deuxième chapitre, comment le développement agile ainsi que ses méthodes ont évolué et pourquoi le besoin d'employer les méthodes agiles s'est imposé. Dans le troisième chapitre, nous présentons notre méthodologie d'analyse ainsi que les différents critères d'identifications utilisés. Le quatrième chapitre explorera le manifeste agile et examinera ses quatre valeurs et ses douze principes en profondeur. L'ensemble des méthodes agiles respecte les principes du manifeste, mais chaque méthode a ses propres principes. Par conséquent, nous étudierons, dans le cinquième chapitre, les principes de trois méthodes agiles. La première méthode, *eXtreme Programming*, est la méthode la plus connue et utilisée parmi les méthodes agiles. Elle est centrée sur la partie construction de logiciel. Une de ses particularités réside dans l'approche de planification qui se matérialise sous la forme d'une pratique intitulée *Planning game*. On notera aussi d'autres pratiques liées au code comme le *Pair programming*, le *Refactoring* que nous abordons en détail dans le chapitre cinq. La deuxième méthode étudiée, *Scrum*, est considérée comme un *framework*

de gestion conçu pour les projets de développement de logiciels. Elle souligne sa différence dans des pratiques de réunions quotidiennes qui ont pour objectifs d'améliorer la motivation de l'équipe, de synchroniser les tâches et d'améliorer le partage des connaissances. Enfin, la troisième méthode étudiée, *Dynamic Systems Development Method*, se caractérise par le fait qu'elle couvre l'ensemble du cycle de développement, contrairement à *Scrum* qui est axé sur la gestion de projet. Chaque méthode étudiée offre une approche légèrement différente, mais toutes ces méthodes ont des processus et des principes spécifiques qui soutiennent une ou plusieurs valeurs du manifeste agile. Dans le dernier chapitre, nous conclurons et discuterons les résultats de notre étude.

CHAPITRE I

CADRE CONCEPTUEL

« Une origine est toujours la fille d'une origine plus ancienne. »

Erik Orsenna

1.1 Introduction

En 2001, de nouvelles idées de développement de logiciel ont été présentées sous la forme d'un manifeste appelé « manifeste agile ». Ce dernier présente quatre valeurs et douze principes. Le but de notre étude est d'analyser le manifeste agile et d'en extraire les principes qui étaient vraiment des principes selon des critères d'identification qui seront présentés en détail dans le troisième chapitre. Cette analyse est faite en se basant sur un travail de recherche analytique sur les principes fondamentaux du génie logiciel présenté dans la thèse de Normand Séguin (Séguin, 2006). Séguin a développé un cadre pour comparer les principes proposés par divers auteurs et par la suite il les a évalués en utilisant ce cadre.

Le développement agile est un nouveau courant de pensée. Il n'a pas été complètement et objectivement examiné dans la littérature. La plupart des articles et des livres concernant le développement de logiciel agile ont été écrits par les auteurs du manifeste agile qui favorisent également leurs propres méthodes agiles. En conséquence, leurs écrits n'effectuent pas d'analyse objective des forces et des faiblesses du développement agile de logiciel. La plupart des méthodes agiles ont été à l'origine conçues pour de petites équipes. Dans cette étude nous visons à analyser le manifeste agile dans la théorie, plus spécifiquement au niveau des principes sous-jacents.

Le manifeste agile donne une base à toutes les méthodes dites agiles, telles que l'*eXtreme programming* (*XP*) et *Scrum*. Nous avons analysé les principes du manifeste agile et avons découvert par la suite que la littérature présente aussi des principes associés à chaque méthode agile. Ainsi, nous avons décidé de les analyser.

1.2 Définition des concepts clés

Notre démarche d'identifications de principes agiles nécessite une définition minutieuse de plusieurs termes. Il existe une confusion au sujet des termes « concept », « principes », « valeur », « pratique » et « méthode ». La littérature recense un nombre important de définitions. Nous avons repris certaines de ces définitions et nous les présentons dans ce premier chapitre.

1.2.1 Qu'est-ce qu'un concept?

Il est impossible de définir exactement ce qu'est un concept, néanmoins Séguin (2006) a dégagé quelques caractéristiques qui semblent importantes pour avoir une meilleure compréhension de ce terme. Dans sa définition de la notion de « concept », Séguin note qu'un concept est « une représentation mentale et générale d'une idée, d'un objet ou d'une notion. Le concept, pouvant se symboliser par un terme, est l'unité de base de composition d'une proposition. » (Séguin, 2006).

Première caractéristique, un concept ne doit pas être confondu avec une idée, car un concept s'élève au-dessus des idées. Le concept permet d'unifier les représentations des objets et des idées. Deuxième caractéristique, un concept peut être symbolisé par un terme. Cela semble peut-être évident, mais ce n'est pas obligatoire. La relation qui unit le terme au concept est une relation d'instance. Le terme est l'expression d'un concept dans un contexte précis. Troisième caractéristique, un concept est l'unité de base d'une proposition. En effet, un concept est le porteur de la structure d'une proposition (Séguin, 2006).

1.2.2 Qu'est-ce qu'un principe?

Au départ d'une recherche ou d'un raisonnement, il est indispensable de définir les éléments qui constituent le cadre conceptuel. Dans notre recherche, le terme « principe » est un composant majeur.

Nous utiliserons la définition suivante du terme principe : « Proposition fondamentale de la discipline formulée sous forme prescriptive (règle), à la source des actions, pouvant être vérifiée dans ses conséquences et par l'expérience. » (Séguin, 2006). Comme le fait remarquer Normand Séguin, un principe serait d'abord une proposition composée de concepts. Un concept à lui seul ne pourrait être considéré comme un principe. Une proposition première signifie qu'un principe ne peut pas être déduit d'un autre principe. Un principe est indispensable à la mise en œuvre d'une pratique.

1.2.3 Qu'est-ce qu'une valeur?

Power (2006) définit les valeurs comme étant des idéaux qui englobent un groupe de personnes. Ils peuvent être positifs ou négatifs. Ces valeurs sont implicites dans la personnalité ou la culture d'une société. Les valeurs sont souvent sensibles : elles représentent les forces motrices derrière les personnes (Power, 2006).

1.2.4 Qu'est-ce qu'une pratique?

Aubry définit une pratique comme suit :

> « Une pratique est une approche concrète et éprouvée qui permet de résoudre un ou plusieurs problèmes courants ou d'améliorer la façon de travailler dans un développement. » (Aubry, 2010)

Aubry rajoute que « les valeurs et les principes sont du niveau de la culture et ne changent pas d'un projet à l'autre, tandis que les pratiques sont leur application dans une situation particulière » (Aubry, 2010). En effet, sans les principes, nous ne pouvons pas savoir comment mettre en œuvre les pratiques étant donné que les principes guident les pratiques.

Shore et Warden (2007) soulignent que les méthodes agiles reposent sur un ensemble de pratiques qui couvre les activités de la réalisation d'un logiciel. Il s'agit simplement d'une combinaison de pratiques agiles avec d'autres pratiques plus classiques.

> « *Agile methods consist of individual elements called practices. Most of these practices have been around for years. Agile methods combine them in unique ways, accentuating those parts that support the agile philosophy, discarding the rest, and mixing in a few new ideas.* » (Shore et Warden, 2007)

1.2.5 Qu'est-ce qu'une méthode?

> « *A method, or process, is a way of working. Whenever you do something, you're following a process. [...] Agile methods are processes that support the agile philosophy. Examples include Extreme Programming and Scrum.* » (Shore et Warden, 2007)

Shore et Warden (2007) soulignent qu'une méthode doit d'une part définir la forme de la communication entre les acteurs dans un projet, d'autre part fournir des supports pour la faciliter.

1.3 Domaines de connaissance du génie logiciel

Dans notre analyse des principes agiles, nous allons nous baser sur le *Guide to the Software Engineering Body of Knowledge (guide SWEBOK)* (Abran *et al.*, 2004) publié par la *IEEE Computer Society Press* et publié également comme *ISO/IEC Technical Report 19759* en 2005. Le guide *SWEBOK* présente l'ensemble des concepts de base de la discipline du génie logiciel.

Le développement agile de logiciel permet d'entreprendre des projets de génie logiciel. Par conséquent, dans la formulation des principes agiles que nous allons analyser devront apparaître des concepts liés directement au génie logiciel. Pour être retenu, un principe agile devra contenir au moins un concept de la discipline du génie logiciel.

Le guide *SWEBOK* fournit les frontières de la discipline de génie logiciel et un accès topique au corpus de connaissances soutenant la discipline du génie logiciel. Le corpus des connaissances est subdivisé en dix domaines de connaissances (DC) :

1. Exigences du logiciel.
2. Conception du logiciel.
3. Construction du logiciel.
4. Essai du logiciel.
5. Maintenance du logiciel.
6. Gestion de la configuration du logiciel.
7. Management du génie logiciel.
8. Processus du génie logiciel.
9. Outils et méthodes du génie logiciel.
10. Qualité du logiciel.

Les domaines de connaissances sont découpés en thèmes décrivant la décomposition en des sous-domaines, des sujets et sous-sujets comme l'illustrent les figures 1.1 et 1.2.

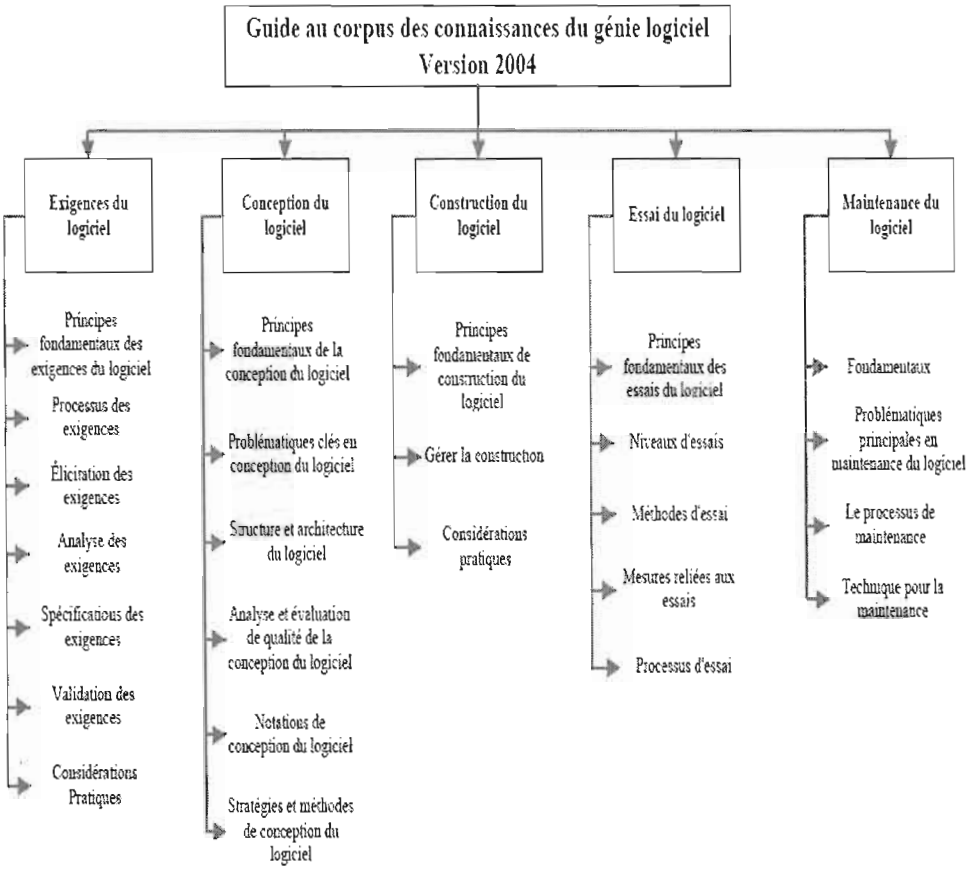

Figure 1.1 Découpage des thèmes pour les cinq premiers domaines de connaissances (Abran *et al.*, 2004).

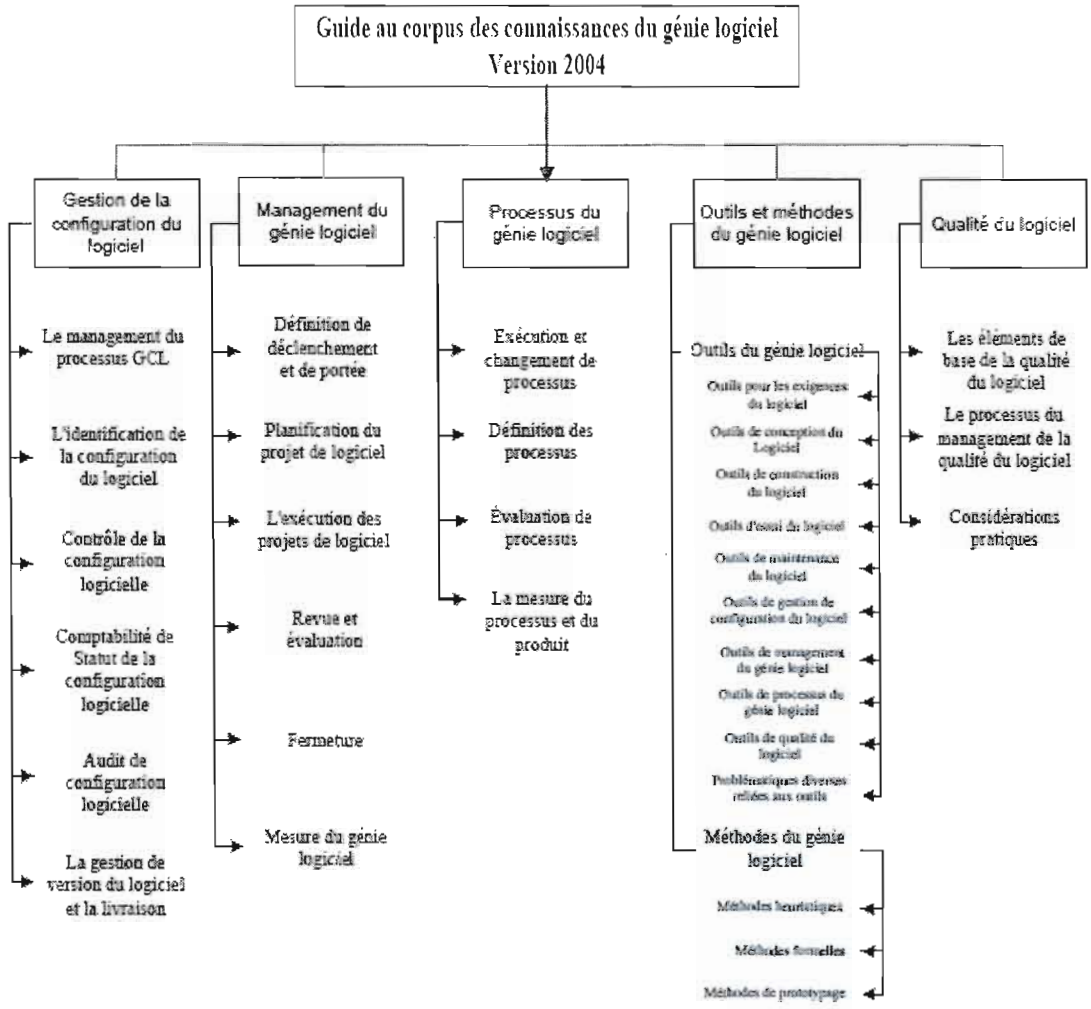

Figure 1.2 Découpages des thèmes pour les cinq derniers domaines de connaissances (Abran *et al.*, 2004).

1.4 Activités du génie logiciel

Pour l'analyse des principes agiles, il est nécessaire d'identifier les activités du génie logiciel car un des critères d'identification des principes que nous allons détailler dans le troisième chapitre exige que le principe ne représente pas une activité du génie logiciel. Pour définir ces activités, nous allons nous référer à la norme *ISO/IEC 12207*. Cette norme présente les processus mis en œuvre dans le cycle de vie d'un logiciel. Elle ne définit aucun cycle de développement particulier (tous les types de projet sont pris en compte). Elle identifie 17 processus du cycle de vie du logiciel : 5 processus primaires, 8 processus de soutien et 4 processus organisationnels. Chaque processus est composé d'un ensemble d'activités (Tableau 1.1).

Processus		Activités	
Processus primaires	**Acquisition**	• Démarrage • Appel d'offres • Préparation du contrat	• Suivi des fournisseurs • Acceptation
	Approvisionnement	• Démarrage • Préparation d'appel d'offres • Contrat • Planification	• Exécution et suivi • Revue et évaluation • Livraison
	Développement	• Démarrage • Analyse des exigences système • Architecture du système • Analyse des exigences logicielles • Conception architecturale du logiciel • Conception détaillée	• Programmation et tests • Intégration logicielle • Test de qualification de logiciel • Intégration système • Test de qualification du système • Installation du logiciel • Acceptation du logiciel
	Exploitation	• Démarrage • Test opérationnel	• Opération du système • Soutien aux utilisateurs
	Maintenance	• Démarrage • Analyse des problèmes et des modifications • Mise en place des modifications	• Revue et acceptation • Migration • Retrait du logiciel
Processus de soutien	**Documentation**	• Démarrage • Conception et développement	• Rédaction • Mise à jour
	Gestion des configurations	• Démarrage • Identification des éléments • Contrôle de la configuration • Suivi de la configuration	• Évaluation de la configuration • Gestion des versions et des livraisons
	Assurance qualité	• Démarrage • Conformité du produit • Conformité du processus	• Conformité du système de qualité

Processus		Activités	
	Vérification	• Démarrage • Contrats • Des processus	• Exigences, du design • Code, intégration et documentation
	Validation	• Démarrage • Cas de tests • Conformité des tests avec les exigences	• Effectuer les tests • Valider la conformité du logiciel • Tester le logiciel dans son environnement
	Revue conjointe	• Démarrage • Du code par rapport au design • De la documentation	• Des tests avec les exigences • De l'exécution des activités • Des coûts et des échéanciers
	Audit	• Démarrage • Revue de suivi de projet	• Revue technique
	Résolution des problèmes	• Démarrage • Description du problème • Analyse du problème	• Solution au problème • Mise en place
Processus organisationnels	**Gestion de projet**	• Démarrage • Planification • Déroulement et suivi	• Revue et évaluation • Fin de projet
	Infrastructure	• Démarrage • Mise en place de l'infrastructure	• Mise à jour de l'infrastructure
	Amélioration	• Mise en place • Évaluation	• Amélioration
	Formation	• Démarrage • Conception du matériel de formation	• Mise en place du plan de formation

Tableau 1.1 Processus du cycle de vie logiciel *ISO/IEC 12207* (Séguin, 2006).

1.5 Conclusion

Dans ce chapitre, nous avons défini ce que nous entendons par les termes « concept », « principe », « valeur », « pratique » et « méthode ». Ensuite nous avons présenté brièvement deux principaux éléments de notre cadre conceptuel à savoir les domaines de connaissance et les activités du génie logiciel. Ces éléments seront utilisés dans la description de notre méthodologie de recherche présentée dans le troisième chapitre. Mais auparavant, dans le deuxième chapitre, nous présentons un survol rapide du développement agile.

CHAPITRE II

LE DÉVELOPPEMENT AGILE

« Les espèces qui survivent ne sont pas les espèces les plus fortes, ni les plus intelligentes, mais celles qui s'adaptent le mieux aux changements. »

Charles Darwin

2.1 Introduction

Ce chapitre a pour objectif de présenter le courant de pensée agile, ses origines et ses différentes méthodes. Dans notre travail, nous nous sommes penchés sur trois méthodes agiles les plus répandues ces dernières années, soit *eXtreme Programming*, *Scrum* et *Dynamic Software Development Method*. En outre, nous voulons indiquer que les racines des méthodes agiles proviennent du développement traditionnel de logiciel auxquels se sont ajoutés de nouvelles méthodes ou techniques. Nous commençons par expliquer le terme « agile ». Par la suite, en nous basant sur notre revue de littérature, nous soulignons l'origine du courant de pensée agile.

2.2 Origine du terme Agile

Le terme « agile » a été introduit dans la langue au XIVe siècle. L'agilité n'est pas un concept unique au développement de logiciel. Il apparaît pour la première fois quand un groupe de chercheurs du *Iacocca Institute* à *Lehigh University* définit un nouveau paradigme en management appelé « *Agile Manufacturing* » (Nagel et Devia, 1999).

Au milieu des années 90, un mouvement qui s'appelle *Agile Modeling* est apparu et qui avait pour but de rendre le processus de développement de logiciels plus efficace. Ce mouvement propose une série de recommandations, valeurs, et principes. Ensuite, la

formation de l'*Agile Alliance* en 2001 et la publication du manifeste agile ont officiellement introduit le terme agile dans le champ du développement de logiciel.

Le terme « AGILE » (Peter, 2007) utilisé comme un acronyme décrit les caractéristiques suivantes :

- ✓ ***A**daptive* (l'apprentissage prime la prédiction).
- ✓ ***G**roup effort* (collaboration au sein de l'équipe).
- ✓ ***I**terative* (développement rapide, intégration continue, tests et livraison fréquente).
- ✓ ***L**ean* (processus simples et souples avec le minimum de bureaucratie).
- ✓ ***E**mpowered* (bien équipé pour réussir le projet).

Plusieurs autres définitions ont été données au terme agile, mais nous allons en souligner seulement quelques-unes, car le nombre de définitions est important et de nouvelles interprétations voient le jour régulièrement. L'agilité est :

- ✓ l'habilité à bouger rapidement de façon légère et gracieuse, caractéristique de ce qui s'adapte facilement (Merriam-Webster, 1990).
- ✓ l'habilité à créer de façon à se démarquer dans un monde des affaires en constante évolution (Agile-Québec, 2005).
- ✓ la capacité de créer et de s'adapter au changement (Agile-Québec, 2005).

Selon ces définitions, nous pouvons dire que l'agilité n'est pas, en soit, une méthodologie; c'est plutôt un courant de pensée qui permet l'existence d'une famille de méthodologies appelées « méthodologies agiles ».

2.3 Origine du courant de pensée Agile

> « *For many people the appeal of these agile methods is their reaction to the bureaucracy of the monumental method.* » (Fowler, 2005).

Fowler (2005) affirme que les méthodes agiles sont nées en réaction aux méthodes dites lourdes ou traditionnelles. Il ajoute que ces dernières se basent sur une planification à la

fois précise et détaillée du développement. Il souligne que ces méthodes ont une rigueur lourde en document et qu'elles sont souvent trop rigides pour pouvoir intégrer les changements des spécifications. À l'inverse, les méthodes agiles sont caractérisées par une planification simple leur permettant de s'adapter aux changements (Fowler, 2005).

Le développement itératif et incrémental n'est pas une pratique moderne, son application remonte aux milieu des années 50 (Larman et Basili, 2003). À la fin des années 80, les travaux de Martin (à l'université d'*Oxford*) dévoilaient les fondements du développement itératif, incrémental et adaptatif, base des approches agiles actuelles. Mais les débuts empiriques des méthodes agiles remontent aux années 80 lorsque Boehm (1988) a introduit un nouveau modèle de développement itératif et incrémental, précurseur des méthodes *eXtreme Programming* et *Scrum*, soit le modèle en spirale (Boehm, 1988).

Dans les années 90, plusieurs ont fait le constat que l'industrie du logiciel fonctionne moins bien et qu'on produit de nombreux logiciels inutiles et beaucoup de documentation qui n'est presque jamais lue. Nous faisons référence ici à toute documentation exhaustive (rapport de conception détaillée, documentation technique, etc.), cela exclut toute documentation succincte et précise. En réaction à cela, il y avait eu un mouvement qui supportait ce qu'on appelait à l'époque des méthodologies légères. En 1991, Martin (1991) s'appuyant sur cette vision, présente une méthode de développement rapide d'application. La structure de cette méthode déterminait un principe adaptatif fondé sur la validation permanente du logiciel par les utilisateurs. Cette méthode de développement, appelée *Rapid Application Development (RAD)*, fut publiée sous la forme d'un guide complet de mise en œuvre servant comme référence. L'ouvrage en question (Martin, 1991) décrit ce que devrait être la méthode idéale de développement rapide. Cette méthode consiste en un cycle de développement court basé sur trois phases (cadrage, conception et construction), avec une période idéale de 90 jours, ou 120 jours au maximum.

En 1995, la méthode *Dynamic Software Development Method (DSDM)* est publiée par Jennifer Stapleton (1995). DSMD est, à l'origine, un *framework* méthodologique basée sur la méthode RAD. Pendant la seconde moitié des années quatre-vingt-dix, une vague d'une

dizaine de méthodes, dont l'*eXtreme Programming* et *Scrum,* sont les principales représentantes, introduit certaines pratiques de planification et de gestion de projet.

En 1995, Shwarber et Sutherland (1995) publièrent les bases de *Scrum* qui constitue une méthode générique de conduite de projets. Cette dernière a été formalisée et présentée par la suite dans la conférence annuelle pour les chercheurs en génie logiciel OOPSLA (Schwaber et Sutherland, 1995).

La méthode *eXtreme Programming* a été mise en œuvre pour la première fois en 1996 sur le projet *Chrysler Comprehensive Compensation System,* qui consistait à mettre en place un nouveau système de gestion de la paie des dix mille salariés d'un fabricant automobile. Après avoir travaillé sur ce projet, Beck et Jeffries officialisèrent la méthode *eXtreme Programming* en 1999. La même année, la méthode *RAD* parvient à maturité, *RAD2* détaillant étape par étape la conduite de projet adaptative. La méthode *RAD2* a été publiée par le *Gartner Group* (1999) dans un rapport intitulé « Réingénierie du Développement d'Applications » (Vickoff, 1999).

Les méthodes se disant vraiment « agiles » arrivent avec la création de *l'Agile Alliance* et la rédaction du manifeste agile, lequel sert de base pour diverses méthodes agiles : comme nous le verrons, les méthodes *XP* et *Scrum* ainsi que de nombreuses autres méthodes agiles sont semblables en termes de valeurs, principes et pratiques.

Au début de 2001, les gens les plus importants ou les plus influents dans le domaine agile se sont réunis pour créer un organisme, l'*Agile Alliance*[1]. Ensuite, 17 praticiens agiles ont écrit et signé un manifeste qui véhicule les valeurs et les principes des approches agiles. Ce manifeste introduit quatre valeurs fondamentales déclinées en douze principes permettant de définir une nouvelle façon de développer des logiciels.

---

[1] Le manifeste est aujourd'hui disponible en 13 langues différentes: arabe, anglais, espagnol, français, italien, japonais, hollandais, norvégien, persan, brésilien, portugais, serbe et suédois.

Plusieurs pensent qu'il y a une méthode agile, ce qui n'est pas le cas. Pour cela, il faut insister sur le fait que « la méthode agile » n'existe pas. Le manifeste agile donne des valeurs et des principes et il y a différentes méthodologies ou approches qui sont conformes à ces principes et ces valeurs.

Certaines valeurs, principes et pratiques agiles étaient déjà mises en œuvre par un nombre important d'équipes de développement sans qu'elles s'en réclament. En effet, les méthodes agiles combinent de manière unique les valeurs, les principes et les pratiques qui prennent en charge la philosophie agile et en rajoutant de nouvelles idées (Shore et Warden, 2007). L'élément fondamental de l'approche agile est de ramener l'individu au centre du processus, donc le processus devient un outil plutôt qu'une fin. L'individu est la variable la plus importante dans le succès d'un projet logiciel. C'est l'un des paradigmes les plus importants que l'approche agile essaie de promouvoir.

Pour ce qui est du passage des méthodes traditionnelles aux méthodes agiles, Ambler (2008) a publié une synthèse des résultats d'une étude faite aux États-Unis. Dans cette étude, 69 % des équipes qui ont répondu au questionnaire disent utiliser une méthode agile. L'étude met aussi l'accent sur le fait que ces équipes agiles produisent des niveaux plus élevés de qualité, sont plus productives, bénéficient d'une plus grande satisfaction des clients et sont ainsi parvenues à réduire les coûts de développement par rapport aux équipes traditionnelles (Ambler, 2008).

Le tableau 2.1 présente de manière chronologique l'apparition de différentes méthodes agiles ainsi que du manifeste agile.

Année	Méthode agile	Auteur(s)
1988	*Spiral Model*	Barry Boehm
1991	*Rapid Application Development*	James Martin
1995	*Dynamic Systems Development Method*	Jennifer Stapleton
1995	*Scrum*	Ken Shwarber et Jeff Sutherland
1999	*Extreme Programming Explained*	Kent Beck et Ron Jeffries
1999	*RAD2*	Gartner Group
2001	*Agile Manifesto*	Agile Alliance

Tableau 2.1 Évolution des méthodes agiles.

Nous pouvons signaler que le modèle en spirale, créé par Barry Boehm (Boehm, 1988) pour le développement logiciel, a influencé toutes les autres méthodes agiles par la suite.

2.4 Le développement agile

Shore et Warden (2007) définissent le développement agile comme suit :

> « *Agile development is a philosophy. It's a way of thinking about software development. The canonical description of this way of thinking is the Agile Manifesto, a collection of 4 values and 12 principles.* » (Shore et Warden, 2007)

Ainsi, pour faire du développement agile il faut mettre les quatre valeurs et les douze principes agiles en pratique. Un processus de développement décrit la méthode qui permet de construire, déployer et éventuellement maintenir le logiciel.

Les adeptes du développement agile le désignent plus souvent comme suit : « Agile = Itératif + Incrémental + Adaptatif ». Toutes les méthodes agiles étudiées dans ce travail utilisent un cycle de développement itératif, incrémental et adaptatif. Avant d'étudier chaque méthode, ce que nous ferons au quatrième chapitre, il convient de se familiariser avec certaines notions fondamentales.

2.4.1 Développement itératif

Le développement itératif n'est pas une idée récente, il existe depuis longtemps sous différentes appellations : évolutionnaire, à paliers, spirale, ainsi que de nombreuses autres désignations. D'après Fowler, l'idée dans le développement itératif est de produire fréquemment une version du système qui fonctionne, mais n'ayant qu'une partie des fonctionnalités demandées qui doivent être intégrées et testées aussi soigneusement qu'une livraison finale (Fowler, 2005).

Une méthode itérative est caractérisée par sa capacité à planifier une itération de production en termes de fonctionnalités et d'interdépendances. Le processus de développement est appliqué plusieurs fois. Le terme itération fait référence à la nature cyclique d'un processus dans lequel les activités sont répétées d'une manière structurée.

D'après Larman (2003), le développement itératif s'organise en une série de mini-projets courts, de durée fixe, nommée itérations. Le résultat de chacune des itérations est un système testé, intégré et exécutable (Larman et Basili, 2003).

2.4.2 Développement incrémental

Dans le développement itératif, chaque itération augmente la quantité d'information, la quantité de logiciel fonctionnel, etc. Dans une méthode dite incrémentale, le logiciel évolue par incrément et chaque itération correspond à un incrément. Le terme incrément fait donc référence au résultat de chaque itération. Le système croît avec le temps de façon incrémentale, itération par itération, d'où le terme de développement incrémental.

2.4.3 Développement adaptatif

Une méthode adaptative est caractérisée par sa capacité à accepter les changements. Selon Fowler (2005), il existe deux aspects à l'adaptabilité : le premier est dans le contexte d'un projet qui adapte un logiciel fréquemment pour tenir compte des changements des besoins des utilisateurs; le deuxième est celui du processus qui change au fur et à mesure du

temps. L'équipe va découvrir ce qui fonctionne pour elle et modifier le processus en conséquence (Fowler, 2005).

2.5 Conclusion

Ce chapitre a présenté l'origine du terme agile suivi d'une introduction sur l'origine de l'agilité et du courant de pensée agile et conclu en présentant le développement agile. Le chapitre a également identifié que le développement agile constitue une combinaison d'idées établies, de nouvelles idées, et d'idées existantes améliorées. Les diverses méthodes agiles diffèrent dans leurs pratiques, mais elles partagent un certain nombre de caractéristiques communes y compris le développement itératif, incrémental et adaptatif.

CHAPITRE III

OBJECTIFS ET MÉTHODOLOGIES

« Cherchons comme cherchent ceux qui doivent trouver et trouvons comme trouvent ceux qui doivent chercher encore. Car il est écrit : celui qui est arrivé au terme ne fait que commencer. »

Saint Augustin

3.1 Introduction

Dans le cadre de développement de logiciel, on entend beaucoup parler d'agilité et de développement agile. Cette nouvelle approche a été grandement popularisée par l'introduction du manifeste agile publié en 2001. Ce dernier explique les fondements des méthodes dites agiles telles que *Scrum* et e*Xtreme Programming*. Le manifeste repose sur quatre valeurs et douze principes. En pratique, ces valeurs et de ces principes sont soutenus par l'adoption d'une ou de plusieurs méthodologies ainsi que par des pratiques spécifiques à chacune d'entre elles. Les méthodes agiles sont présentées comme une alternative aux méthodes dites traditionnelles, considérées comme bureaucratiques et lourdes. Elles sont aussi considérées comme une réponse au grand nombre de projets informatiques qui ont échoué durant la dernière décennie.

C'est dans ce contexte que le présent travail rassemble l'information nécessaire sur le développement agile de logiciel dans le but de faire un inventaire théorique de tous les principes agiles pour les analyser par la suite.

3.2 Sujet de recherche

Le travail présenté dans ce mémoire peut être vu comme une suite du travail de recherche analytique sur les principes fondamentaux du génie logiciel présenté dans la thèse de Normand Séguin : « Inventaire, analyse et consolidation des principes fondamentaux du génie logiciel » (Séguin, 2006). Cette étude a répertorié plus de 300 principes du génie logiciel proposés depuis 35 ans. Dans cette thèse, Normand Séguin a traité les principes recensés pour ne conserver qu'une liste réduite de principes. À cette fin, il a conçu une démarche d'analyse que nous allons brièvement présenter. Tout d'abord, il a défini ce qu'était un principe. La définition du terme « principe » est essentielle pour l'analyse des principes. Ensuite, il a déterminé les critères d'identification des principes. Certains critères dépendent de la définition du terme principe. Par la suite, il a conçu une méthodologie composée de quatre phases présentées au tableau 3.1. Puis il a vérifié la liste des principes retenus afin d'évaluer le degré de couverture de la discipline du génie logiciel avec un modèle d'ingénierie et avec le corpus de normes du génie logiciel de l'*IEEE*.

Phases	Description
1) Définition du cadre conceptuel d'analyse.	• Concepts du génie • Définition du génie logiciel et identification des concepts • Identification des activités du génie logiciel • Définitions : concept, principe, lois • Critères d'identification des principes
2) Évaluation selon les critères individuels	• Application des critères individuels
3) Évaluation selon les critères d'ensemble, catégorisation et liens avec la norme *ISO/IEC 12207*	• Catégorisation des principes • Application des critères d'ensemble • Liens avec la norme *ISO/IEC 12207*
4) Évaluation du degré de couverture des principes retenus	• Avec les éléments d'un modèle de l'ingénierie • Avec les normes du génie logiciel de l'*IEEE*

Tableau 3.1 Phases de la méthodologie de recherche pour l'identification des principes du génie logiciel (Séguin, 2006).

Bien que la philosophie d'analyse des principes agiles soit directement issue du travail de Séguin (2006), la démarche n'est pas identique. En particulier, notre démarche d'analyse est construite à partir de trois phases principales : la définition du cadre conceptuel, la phase d'évaluation selon les critères individuels et l'interprétation des résultats obtenus. Cette méthodologie nous permettra d'identifier les principes qui correspondent bien à des principes selon les critères d'identification détaillés dans la section 3.4 de ce chapitre.

Le travail que nous avons effectué vise également à démystifier les valeurs et les principes proposés par la communauté agile dans le but d'identifier les principes du mouvement agile. Nous effectuerons en premier lieu une revue des principes répertoriés dans les références agiles depuis le fondement de ce mouvement. Cette revue met spécifiquement l'accent sur les principes du manifeste agile et les principes de trois méthodes agile : *eXtreme Programming*, *Scrum* et *DSDM*. Pour ce faire et comme il y a de nombreuses citations et de références bibliographiques, il a fallu favoriser les plus pertinentes et les plus récurrentes dans la communauté agile, qu'on désigne habituellement comme références dans les livres, les articles et les sites consacrés au mouvement agile. Nous avons choisi d'utiliser le manifeste agile comme base pour expliquer et évaluer les idées liées au développement de logiciel agile.

3.3 Objectif de la recherche

Le but de notre travail est d'analyser la liste des principes proposés par la communauté agile en appliquant des critères d'identification individuels. L'objectif de cette analyse est de vérifier si les principes agiles proposés sont vraiment des principes selon les critères. Les principes analysés sont ceux du manifeste agile et ceux des trois méthodes agiles: douze principes du manifeste agile, cinq principes de la méthode *eXtreme Programming*, six principes de *Scrum* et neuf principes de *DSDM*. Ainsi, nous disposons d'un total de 32 principes agiles à analyser selon les critères d'identification individuels. Pour réaliser cette analyse, nous allons suivre la méthodologie de recherche présentée à la section suivante.

3.4 Méthodologie

Notre méthode d'analyse (figure 3.1) utilise une liste de critères pour vérifier si les principes recensés de la revue de littérature correspondent à des principes selon des critères d'identification. Comme l'indique Séguin : « Ces critères sont supportés par les activités du génie logiciel (*ISO/IEC 12207*), les concepts du génie logiciel (*SWEBOK*), les concepts généraux du génie et les concepts de l'informatique. » (Séguin, 2006)

Il existe deux types de critères :

- ✓ des critères individuels qui s'appliquent individuellement à chacun des principes.
- ✓ des critères d'ensemble qui s'appliquent à un ensemble de principes.

Dans ce mémoire, nous ne nous attarderons qu'aux critères individuels. Les cinq critères individuels d'identifications que nous allons appliquer aux principes agiles sont les suivants :

- ✓ **Critère n° 1 :** Un principe est une proposition formulée de façon prescriptive.
- ✓ **Critère n° 2 :** Un principe ne doit pas être associé directement ou découler d'une technologie, d'une méthode ou d'une technique ou être une activité du génie logiciel.
- ✓ **Critère n° 3 :** Le principe ne doit pas énoncer un compromis entre deux actions ou concepts.
- ✓ **Critère n° 4 :** Un principe du génie logiciel devrait inclure des concepts reliés à la discipline ou au génie.
- ✓ **Critère n° 5 :** La formulation d'un principe doit permettre de le tester en pratique ou de le vérifier dans ses conséquences.

Le premier critère tente de vérifier si les principes étudiés sont formulés sous forme d'une proposition prescriptive. Le principe doit spécifier l'action à faire, sans pour autant spécifier comment la faire. La formulation des principes est étudiée comme présentée par les auteurs.

Le deuxième critère est soutenu par la norme *ISO/IEC 12207*. Si la formulation du principe renferme une activité déjà identifiée par la norme ou fait référence à une technologie, une technique ou une méthode, le principe n'est pas retenu. Ainsi, un principe ne devrait pas être une activité, une technique ou une méthode du génie logiciel.

Le troisième critère vérifie si le principe étudié ne comporte pas de dosage ou de compromis entre deux concepts ou actions.

Le quatrième critère vérifie si un principe agile contient un ou des concepts explicites du génie logiciel. Le guide *SWEBOK* présenté au premier chapitre fournit l'ensemble des concepts de base de la discipline du génie logiciel et permet de soutenir l'application de ce critère. Plus précisément, dans la formulation des principes agiles devront apparaître des concepts liés directement au génie logiciel. Ainsi, un principe agile ne sera retenu que s'il contient au moins un concept de la discipline du génie logiciel.

Le cinquième critère s'assure que la formulation du principe permet de le tester et de le vérifier dans ses conséquences. Le principe doit spécifier le comment faire. L'application de ce critère n'est pas une tâche facile. Nous devons vérifier dans la formulation du principe ou dans l'explication donnée par les auteurs comment le principe peut être testable et vérifiable dans ses conséquences.

Les deux critères d'ensemble sont les suivants :

- ✓ **Critère n° 1 :** Les principes devraient être indépendants (non déduits).
- ✓ **Critère n° 2 :** Un principe ne doit pas contredire un autre principe connu.

Dans le présent travail, nous ne nous intéresserons qu'aux critères d'identifications individuels. L'analyse des principes agiles selon les critères d'ensemble sera à faire dans un travail futur. Dans l'application de ces critères, un principe n'est retenu que s'il satisfait aux cinq critères individuels. Par contre dans le cas où un principe ne satisferait pas au premier critère et satisferait aux quatre autres critères, une reformulation mineure pourra être appliquée afin de le rendre sous forme prescriptive. Cette reformulation devrait être mineure. Dans le cas où le principe doit être entièrement réécrit, la reformulation est considérée comme majeure. Dans ce cas, le principe sera écarté.

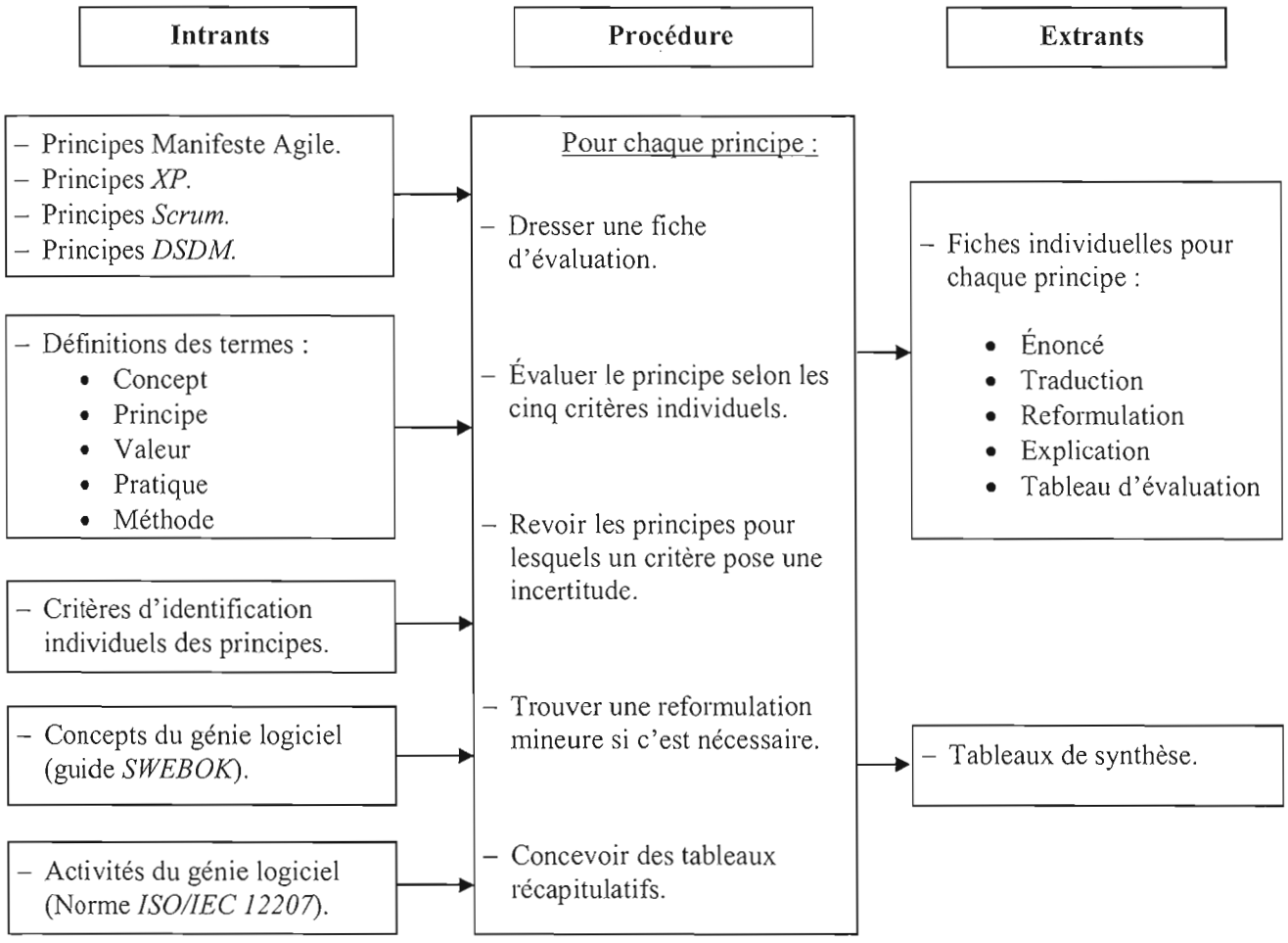

Figure 3.1 Méthodes d'analyse des principes agiles selon Séguin (Séguin, 2006).

L'évaluation individuelle des principes comprendra les informations suivantes : l'énoncé du principe tel que mentionné, une explication du principe, un tableau regroupant les cinq critères individuels d'identification des principes. Pour chacun des critères, on identifie si le principe satisfait ou non le critère avec la justification.

3.5 Structure du document

La structure du mémoire suit les phases principales de la méthode d'analyse utilisée. La figure 3.2 montre la structure du notre mémoire.

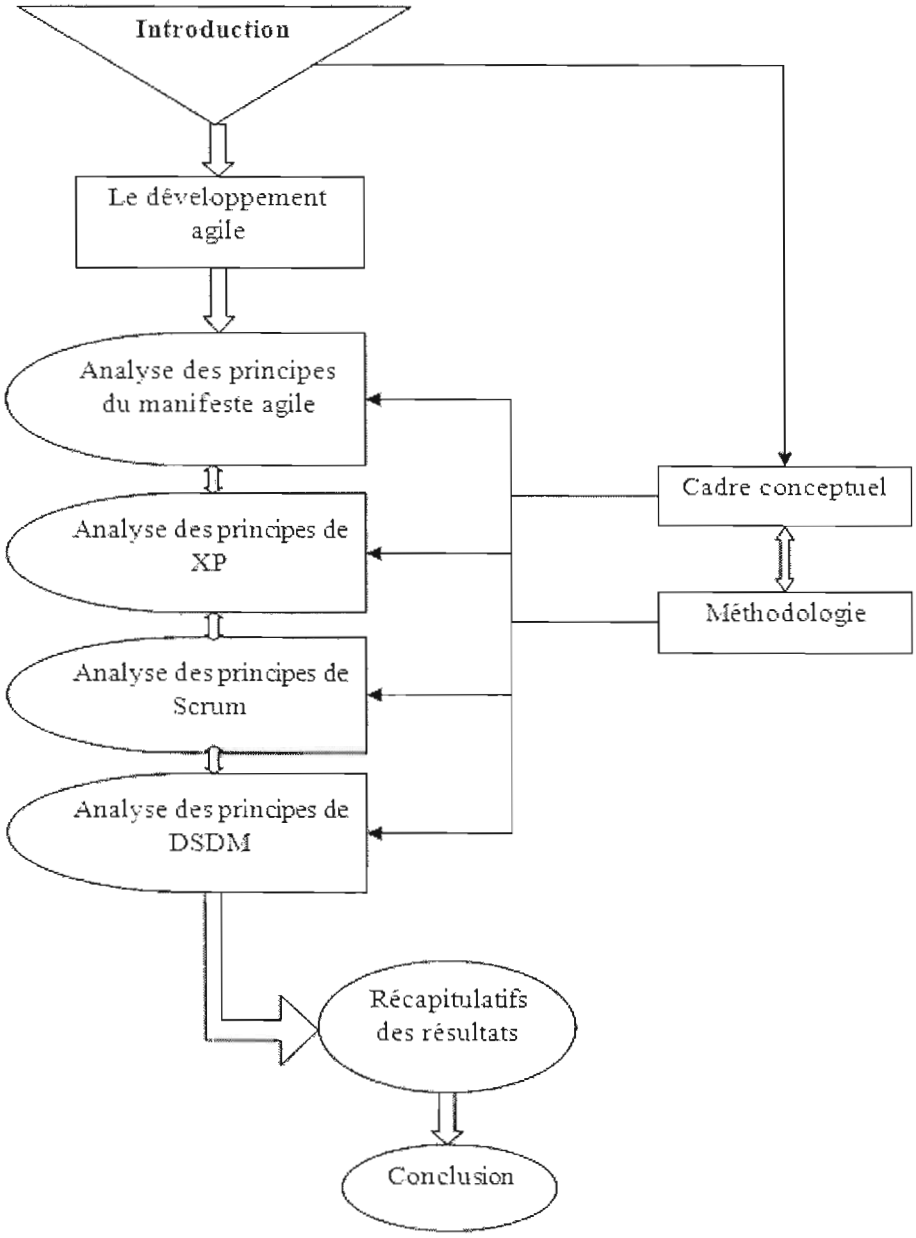

Figure 3.2 Structure du mémoire.

CHAPITRE IV

ANALYSE DES PRINCIPES DU MANIFESTE AGILE

« Attendre d'en savoir assez pour agir, c'est se condamner à l'inaction. »

Jean Rostand

4.1 Introduction

Du 11 au 13 février 2001, un certain nombre d'experts de l'industrie et d'acteurs importants du monde des méthodes agiles tels que Kent Beck, Ken Schwaber, Alistair Cockburn, Jim Highsmith et des praticiens des méthodes agiles comme *XP*, *SCRUM*, *ASD*, *FFD*, *Crystal Method*, *DSDM* se sont regroupés pour créer l'*Agile Alliance*. Leur but était de décrire les valeurs et les principes qui permettraient à une équipe de développement de logiciel de progresser rapidement tout en ayant une capacité à s'ajuster au changement (Beck *et al.*, 2001).

Pour ce faire, ils ont mis au point et signé un « *Agile Manifesto* » comportant quatre valeurs et douze principes. Le mot « manifeste » exprime une déclaration de convictions et l'adjectif « agile » capture la capacité d'adaptation et de réaction au changement (Fowler, 2005). Les douze principes représentent les déclinaisons opérationnelles des quatre valeurs fondamentales agiles. Ils mettent en valeur la collaboration, l'interaction, les livraisons fréquentes, la validation permanente, etc.

Notre objectif dans ce chapitre est de comprendre et d'analyser les principes du manifeste agile et cela en les évaluant par rapport aux critères d'identification individuels que nous avons définis au troisième chapitre. Cette évaluation nous permettra de savoir si les douze principes du manifeste agile sont vraiment des principes selon les critères. Pour

pouvoir procéder à cette évaluation, il faut comprendre autant les principes que les valeurs du mouvement agile.

Ainsi, nous commencerons tout d'abord par expliquer ce que le mouvement agile appelle « valeur » en passant en revue chaque valeur du manifeste. Ensuite, nous donnerons une explication détaillée et ferons une évaluation de chaque principe qui forme le manifeste agile. Nous présenterons la synthèse de notre analyse à la section 4.4.

Le texte des quatre valeurs et des douze principes du manifeste agile sera présenté à la fois en anglais et en français :

- ✓ En français en se référant à la traduction qui a été faite par le Groupe d'utilisateurs Agiles de Montréal[2].
- ✓ En anglais en se référant au manifeste agile (Beck *et al.*, 2001).

4.2 Valeurs agiles

Selon Fowler (2005), les méthodes agiles sont moins orientées « paperasse », évoquant généralement une quantité moindre de documentation pour une tâche donnée. Elles sont plutôt orientées « code », en suivant l'idée selon laquelle l'élément clé de la documentation est le code. Elles sont adaptatives plutôt que prédictives, elles s'accommodent favorablement du changement. Elles sont orientées vers l'humain, plutôt que vers le processus (Fowler, 2005).

---

[2] Ce groupe a pour but de favoriser et de soutenir l'échange d'idées, de connaissances et d'expériences concernant le développement agile de logiciels ainsi que ses méthodes et ses pratiques. Le Groupe d'utilisateurs Agiles de Montréal est membre de l'*Agile Alliance* et supporté par le CRIM.

L'*Agile Alliance* a formulé ses idées en se basant sur quatre valeurs (Agile-Alliance, 2002):

1. *Individuals and interactions over processes and tools.*
2. *Working software over comprehensive documentation.*
3. *Customer collaboration over contract negotiation.*
4. *Responding to change over following a plan.*

Notre étude des valeurs agiles sera présentée comme suit :

- ✓ L'énoncé de la valeur telle que mentionnée dans le manifeste agile.
- ✓ La traduction du principe selon le Groupe d'utilisateurs Agiles de Montréal.
- ✓ Une explication de la valeur en se basant sur celle donnée par les auteurs du manifeste.

Figure 4.1 Le manifeste pour le développement Agile de logiciels (Beck *et al.*, 2001).

Les auteurs du manifeste agile déclarent avoir trouvé de nouvelles façons de développer les logiciels. Ainsi, ils accordent une certaine valeur aux items de droite de la figure 4.1, mais ils donnent plus de valeur à ceux de gauche.

4.2.1 Première valeur du manifeste

Énoncé : *Individuals and interactions over processes and tools.*

Traduction : Les individus et les interactions davantage que les processus et les outils.

Explication

Les individus et les interactions doivent primer sur les processus et les outils : les membres de l'*Agile Alliance* affirment que la meilleure garantie de succès réside dans les personnes (motivation, sens de l'initiative, travail en groupe, communication interpersonnelle) (Agile-Alliance, 2002).

Dans le cadre agile, l'équipe est bien plus importante que les moyens matériels ou les procédures. En effet, la pensée agile valorise les personnes et les interactions plutôt que les processus et les outils. Cela est constaté par l'accent mis sur les êtres humains en tant qu'individus et sur l'expertise des équipes de développement, qui doivent communiquer entre elles de façon étroite et dans un esprit constant de confiance.

> « *These new methods attempt a useful compromise between no process and too much process, providing just enough process to gain a reasonable payoff.* » (Fowler, 2005)

Cette déclaration de Fowler explique, à ceux qui ont interprété cette valeur en disant que les promoteurs de l'approche agile ne veulent ni processus ni outils, que les processus et les outils sont présents pour supporter les individus dans leurs interactions. Ce qui a déjà été énoncé est considéré comme une réaction aux approches dites traditionnelles comme l'approche en cascade où il est nécessaire de produire beaucoup de documentation. L'idée de l'approche agile est de disposer d'abord d'individus motivés à collaborer, ensuite il faut leur donner un bon guide des objectifs précis et les processus ainsi que les outils pour les soutenir.

4.2.2 Deuxième valeur du manifeste

Énoncé : *Working software over comprehensive documentation.*

Traduction : Un logiciel fonctionnel davantage que la documentation exhaustive.

Explication

Le développement de logiciels fonctionnels doit primer sur la documentation exhaustive. Partant du principe que le meilleur transfert de connaissances s'effectue par le travail en équipe, les développeurs agiles doivent participer à toutes les étapes du développement et travailler ensemble. Les documents contenant les exigences, les spécifications ou la conception peuvent être utiles pour guider le travail des développeurs, mais l'*Agile Alliance* déclare que l'écriture et la maintenance de la documentation consomment trop de ressources (Agile-Alliance, 2002).

Les membres de l'*Agile Alliance* sont favorables à la documentation concise, ne décrivant que les grandes lignes de l'architecture du produit et régulièrement mise à jour, et à la documentation permanente du code lui-même.

> « *We want to restore a balance. We embrace modeling, but not in order to file some diagram in a dusty corporate repository. We embrace documentation, but not hundreds of pages of never-maintained and rarely-used tomes.* » (Agile-Alliance, 2002)

Cette déclaration lève l'ambiguïté sur l'interprétation qui dit que lorsque l'on fait du développement agile, on ne produit pas de documentation, ce qui n'est pas le cas, car il faut valoriser le logiciel fonctionnel à la place de la documentation détaillée. Dans ce cas, une question surgit : « Quel type de documentation est requis et apporte de la valeur? » À titre d'exemple, un document de conception bien détaillé mais qui n'est plus à jour n'a plus de valeur. Certes, la documentation représente une charge de travail importante, et peut même être néfaste si elle n'est pas à jour, mais cela ne se traduit pas par l'absence de document de conception, puisque la modélisation peut être faite dans un contexte agile (Ambler, 2002).

L'idée dans l'agilité est de bien choisir la documentation qu'on va produire. Ce choix est soutenu par la méthode « *Agile Modelling* »[3]. La modélisation agile (AM) est une collection de valeurs, de principes, et de pratiques pour la modélisation du logiciel qui peut être appliquée sur un projet de développement de logiciel et combinée à d'autres méthodes telles que *XP*.

> « Nous sommes d'accord pour documenter, mais pas des tomes de plusieurs centaines de pages jamais maintenus à jour et rarement utilisés. Nous sommes d'accord pour planifier, mais nous reconnaissons les limites de la planification dans un environnement turbulent. » (Highsmith, 2001)

Pour les promoteurs des méthodes agiles, il est vital que l'application fonctionne et le reste, comme la documentation technique, est secondaire, même si une documentation succincte est utile comme moyen de communication. Cette idée est soutenue par les notions de développement itératif et d'intégration continue, en insistant sur la simplicité et la robustesse du code produit par le biais des tests.

Dans la formulation de cette valeur, on retrouve l'expression « *working software* » ce qui est traduit par un logiciel fonctionnel et opérationnel assuré par des tests client faits à chaque fin d'itération.

L'exemple de la figure 4.2 montre un *backlog* produit dans un projet *Scrum*. Cet extrait du *backlog* produit désigne un type de document fait par l'équipe de développement. Dans le cas de la première fonctionnalité, le document produit sera un diagramme de séquence UML.

---

[3] http://www.agilemodeling.com

PRODUCT BACKLOG (example)					
ID	**Name**	**Imp**	**Est**	**How to demo**	**Notes**
1	Deposit	30	5	Log in, open deposit page, deposit €10, go to my balance page and check that it has increased by €10.	Need a UML sequence diagram. No need to worry about encryption for now.
2	See your own transaction history	10	8	Log in, click on "transactions". Do a deposit. Go back to transactions, check that the new deposit shows up.	Use paging to avoid large DB queries. Design similar to view users page.

Figure 4.2 Extrait d'un *backlog* produit dans un projet *Scrum* (Kniberg, 2007).

Il existe d'autre type de document produit par une équipe agile comme le montre la figure 4.3. Chaque méthode agile possède une documentation adaptée à son cycle de vie.

Method	Documents
Scrum	Backlog, Running Code
XP	Stories, Running Code, Tests
Crystal Orange	Release sequence; Schedule (user viewings and deliveries); Annotated use cases or feature descriptions; Requirements document (purpose, use cases, non-functional requirements, interface definitions); Design sketches and notes as needed; UI Design / Screen drafts; Common object model; Running code; Migration code; Test cases; User Manual; Status reports; Inter-team specs;
DSDM	Feasibility Report; Outline Plan; Business Area Definition; Non-Functional Requirements List; Systems Architecture Definition; Development Plan; Functional Model; Functional Prototype; Design Prototype; Tested system; Delivered system; Implementation Plan; Development Risk Analysis Report; Review Records; Test records; User documentation; Project Review Document;

Figure 4.3 Exemples de documentations utilisées dans les méthodes agiles (Agile-Alliance, 2002).

4.2.3 Troisième valeur du manifeste

Énoncé : *Customer collaboration over contract negotiation.*

Traduction : La collaboration avec le client davantage que la négociation de contrat.

Explication

La collaboration avec le client doit primer sur la négociation contractuelle : les membres de l'*Agile Alliance* affirment que le succès d'un projet requiert un *feed-back* régulier et fréquent de la part du client. La meilleure manière de procéder pour le client est de travailler en étroite collaboration avec l'équipe de développement ce qui assure un meilleur contrôle du projet (Agile-Alliance, 2002). Cela n'est pas simple à mettre en œuvre et nécessite une réelle relation de confiance entre le client et l'équipe de développement.

Tout au long d'un projet agile, le client collabore avec l'équipe et fournit un *feed-back* continu sur l'adaptation du logiciel à ses attentes. Par conséquent, le client devient un partenaire à part entière, qui participe au développement dans le sens où il détermine l'objectif à atteindre pour obtenir une réelle amélioration.

Selon Cockburn (2001), une bonne collaboration avec le client peut sauver un contrat en situation problématique ou peut au contraire rendre un contrat inutile. Dans les deux cas, la collaboration est l'élément gagnant (Cockburn, 2001). Un contrat est défini comme un ensemble d'éléments liant un demandeur et un fournisseur et définissant d'une manière précise, complète et cohérente, leurs obligations respectives (Grand-dictionnaire-terminologique, 1997). Autrement dit, un contrat de développement logiciel engage un développeur à réaliser un logiciel, et ce, conformément aux conditions prévues par le cahier des charges.

Cockburn (2001) fournit une collection de types de contrat[4] agile qui sont essentiels lors de la présentation des contrats agiles aux clients. Bien entendu, il faut que le client soit convaincu des avantages du développement agile.

4.2.4 Quatrième valeur du manifeste

Énoncé : *Responding to change over following a plan.*

Traduction : La réponse au changement davantage que le suivi d'un plan.

Explication

Dès le début du projet, il est évident que personne, pas même le client, ne peut anticiper avec précision l'ensemble des besoins. Le développement agile vise à atteindre un compromis entre les spécifications initiales présentées aux développeurs et le résultat final qui bien souvent s'en éloigne un peu, voire beaucoup, en incluant des modifications tout au long du cycle de développement. Cela impose l'utilisation d'outils de suivi et une attitude constante de coopération avec le client et, à n'en pas douter, que la capacité à accepter le changement fait bien souvent la réussite ou l'échec d'un projet agile.

Il faut être conscient que le changement peut arriver à n'importe quelle phase du projet et il faut voir les changements comme une opportunité plutôt qu'un obstacle. Certes, l'ouverture au changement doit primer le suivi d'un plan rigide, car les premières versions du logiciel vont souvent provoquer des demandes de modifications par le client (Cockburn, 2001). Lorsque l'utilisateur teste le logiciel, c'est à ce moment que les défauts apparaissent surtout, au niveau des besoins mal compris par l'équipe de développement.

---

[4] Il existe un groupe de travail dont l'objectif est de produire des contrats agiles réutilisables pour les entreprises effectuant le développement agile.
www.openplans.org/projects/agile-contracts/summary

4.3 Les principes derrière le manifeste agile

Des quatre valeurs citées ci-dessus découlent douze principes généraux communs aux méthodes agiles.

Pour les explications des principes, nous nous sommes principalement basés sur le livre de Cockburn (Cockburn, 2001).

La liste ci-dessous présente la liste des principes telle qu'énoncée par le manifeste agile :

1. *Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.*
2. *Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.*
3. *Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.*
4. *Business people and developers must work together daily throughout the project.*
5. *Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.*
6. *The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.*
7. *Working software is the primary measure of progress.*
8. *Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.*
9. *Continuous attention to technical excellence and good design enhances agility.*
10. *Simplicity—the art of maximizing the amount of work not done—is essential.*
11. *The best architectures, requirements, and designs emerge from self-organizing teams.*
12. *At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.*

L'analyse individuelle des principes comprendra les informations suivantes :

- ✓ L'énoncé du principe tel que mentionné dans le manifeste agile.
- ✓ La traduction du principe selon le Groupe d'utilisateurs Agiles de Montréal.
- ✓ Un ajustement que nous proposons de cette traduction pour certains de ces principes.
- ✓ Une explication du principe en se basant sur celle donnée par les auteurs du manifeste.
- ✓ Un tableau regroupant les cinq critères individuels d'identification des principes. Pour chacun des critères, on identifie si le principe satisfait ou non le critère avec la justification.

4.3.1 Principe 1

Énoncé : *Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.*

Traduction : Notre plus haute priorité est de satisfaire le client en lui livrant rapidement, et ce, de façon continue un logiciel de qualité.

Ajustement de la traduction : Notre plus haute priorité est de satisfaire le client en lui livrant rapidement, et de façon continue, un logiciel de qualité.

Explication

Le produit croît avec le temps de façon incrémentale, itération par itération. L'affinement successif du produit par le biais d'itérations multiples et le *feed-back* du client constituent la base la plus importante qui permet d'avoir un produit de qualité, opérationnel et utile.

La communauté agile donne une grande importance à la satisfaction du client et cela en lui livrant rapidement et de façon continue un « logiciel de qualité ». L'interprétation de l'expression « logiciel de qualité » diffère selon les intervenants du projet. Pour le

développeur, un logiciel de qualité est un logiciel bien conçu respectant tous les concepts orientés objet tels que l'encapsulation, l'héritage et l'abstraction si le langage utilisé est un langage-objet. Pour l'utilisateur final, un logiciel de qualité est un logiciel avec une interface utilisateur personnalisable, riche de fonctionnalités et facile à utiliser. Pour le service de soutien, un logiciel de qualité est un logiciel facile à installer, à configurer et à maintenir.

La livraison d'un logiciel fonctionnel doit se faire régulièrement et assez souvent pour avoir un *feed-back* rapide de la part du client, ce qui guide les prochaines étapes du projet agile (Cockburn, 2001).

Les méthodes agiles recommandent de livrer dans les premières semaines une version initiale de l'application puis de livrer des versions auxquelles des fonctionnalités s'ajoutent progressivement. À chaque version du logiciel, un *feed-back* de la part du client est nécessaire pour permettre soit de continuer le développement comme prévu, soit d'effectuer des changements. De cette manière, les changements dans les spécifications interviennent tôt dans le processus de développement et sont moins problématiques.

Ce principe incite les développeurs à se concentrer sur la livraison fréquente des fonctionnalités du système et met l'accent sur la participation active du client dans le processus de développement agile. Autrement dit, livrer des versions incrémentales et fréquentes dès que le produit offre une valeur ajoutée.

Une méthode traditionnelle, non itérative et non incrémentale, de développement de logiciel effectue la livraison quand le système au complet est terminé, tandis qu'une méthode de développement agile attend le *feed-back* du client avant de tout construire pour pouvoir réviser ce qui ne fonctionne pas. Par exemple, la méthode de gestion de projet *Scrum* recommande un développement par itérations de durée maximale de 30 jours suivie d'une démonstration. Cette dernière permet à l'équipe de développement de décider si elle doit effectuer des modifications au niveau des fonctionnalités, intégrer les fonctionnalités et livrer le produit.

Évaluation

N° du critère	Critère satisfait	Justification
1	X	Formulation prescriptive.
2	X	Ne découle pas d'une technologie, d'une méthode, d'une technique ou d'unc activité du génie logiciel.
3	X	Pas de compromis entre deux actions ou concepts.
4	X	Dans sa formulation le principe comprend « *valuable software* » qui est un concept du génie logiciel (selon le guide *SWEBOK*).
5	X	Ce principe est soutenu par l'approche de développent incrémentale et itérative. Ainsi, sa formulation est vérifiable en pratique par le biais de la production et la livraison fréquente et continue des versions du logiciel. Puisqu'atteindre la satisfaction des mandataires est un des principaux objectifs du projet, il est important que le progrès vers ce but soit évalué formellement et périodiquement (Abran et all., 2004). Par conséquent, dans un cadre agile, la livraison d'un logiciel fonctionnel le plus tôt possible permet aux clients de le tester, de l'évaluer et d'exprimer s'il est

Tableau 4.1 Grille d'analyse du principe 1 selon les critères individuels.

Résultat

Ce principe satisfait les cinq critères, par conséquent il est retenu comme principe selon les critères d'identification individuel.

4.3.2 Principe 2

Énoncé : *Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.*

Traduction : Accepter les changements de besoins, même lors du développement. Les processus agiles exploitent les changements pour augmenter les avantages compétitifs du client.

Ajustement de la traduction : Accueillir les changements aux exigences, même tardifs. Les processus agiles mettent à profit les changements pour augmenter les avantages compétitifs du client.

Explication

Le client est souvent dans l'incapacité de définir ses besoins de manière exhaustive dès le début d'un projet. Dans ces conditions, les exigences logicielles définies sont moins précises et ont tendance à changer au fur et à mesure de l'avancement du projet. Pour remédier à ce problème, une équipe agile essaie de conserver la structure de son logiciel flexible pour permettre l'intégration des changements sans avoir un grand impact sur le projet. Pour maintenir cette flexibilité, l'équipe peut par exemple appliquer les principes et les pratiques de la conception orientée objet. Par exemple, ces principes et pratiques sont décrits par Martin dans son livre « *Agile Software Development : Principles, Patterns, and Practices.* » (Martin, 2003).

Ce principe suggère d'intégrer les changements aux exigences même s'ils arrivent tard dans le processus de développement. Chaque méthode agile a sa façon de procéder pour exploiter les changements de sorte que l'impact sur le logiciel reste minimal. La gestion de changement dans un projet agile n'existe pas, car chaque demande de changement est considérée comme un nouveau besoin à ajouter dans la liste de fonctions à développer appelée *backlog* produit dans le jargon de la méthode de gestion de projet *Scrum*. Autrement dit, un projet agile *Scrum* est précédé par une phase d'établissement des besoins initiaux qui sont listés dans le *backlog*, puis à chaque fin d'itérations, une mise à jour des besoins du *backlog* est effectuée étant donné que les tâches les plus importantes sont faites en premier. Ainsi, il y aura une classification régulière des besoins exprimés dans le *backlog* pour en choisir les fonctionnalités qui seront implémentées en priorité et il y a toujours la possibilité de changer ce catalogue à tout moment : ajouter ou enlever des éléments, changer les priorités et modifier les descriptions.

Ce principe est applicable au niveau de toutes les phases de projet, étant donné que la communauté agile affirme que les changements à l'initiative du client sont acceptés en tout

temps et qu'ils sont bien considérés, vu que cela est synonyme d'amélioration (Agile-Alliance, 2002).

Évaluation

N° du critère	Critère satisfait	Justification
1	X	Formulation prescriptive.
2	X	Ne découle pas d'une technologie, d'une méthode, d'une technique ou d'une activité du génie logiciel.
3	X	Pas de compromis entre deux actions ou concepts.
4	X	Le changement a un impact dans le génie logiciel, ainsi il est considéré comme concept du génie logiciel. Dans sa formulation, le principe comprend « *Change* » « *Requirements* » qui sont des concepts du génie logiciel (selon le guide *SWEBOK*).
5	X	Le principe est vérifiable en pratique du fait que l'équipe de développement accepte et intègre les changements proposés par le client.

Tableau 4.2 Grille d'analyse du principe 2 selon les critères individuels.

Résultat

Cette proposition satisfait les cinq critères d'identification individuels des principes, par conséquent elle est retenue comme principe agile.

4.3.3 Principe 3

Énoncé : *Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.*

Traduction : Livrer fréquemment un logiciel fonctionnel, en visant les délais les plus courts, de quelques semaines à quelques mois.

Ajustement de la traduction : Livrer fréquemment un logiciel fonctionnel, de quelques semaines à quelques mois, en privilégiant les courts délais.

Explication

Ce principe met l'accent sur la livraison fréquente d'un logiciel fonctionnel. Livrer un logiciel fonctionnel permet aux utilisateurs de fournir leur rétroaction sur le produit plus tôt dans le projet. Ceci laisse aux développeurs la possibilité d'ajuster leur interprétation et de la maintenir alignée avec la vision des utilisateurs.

L'Agile Alliance incite à livrer le plus souvent possible des versions opérationnelles de l'application, avec une fréquence de quelques semaines ou de quelques mois, en visant de courts délais (Agile-Alliance, 2002).

L'idée dans l'approche agile est de toujours livrer le plus rapidement possible au client une solution qui satisfait ses besoins. Le choix des fonctionnalités qui seront livrées au client à la fin de chaque itération est une question clé. Bien entendu, un choix doit être effectué en tenant compte des priorités du client (Agile-Alliance, 2002).

Évaluation

N° du critère	Critère satisfait	Justification
1	X	Formulation prescriptive.
2	X	Ne découle pas d'une technologie, d'une méthode, d'une technique ou d'une activité du génie logiciel.
3	X	Pas de compromis entre deux actions ou concepts.
4	X	Dans sa formulation, le principe comporte « *Software* » qui est un concept du génie logiciel.
5	X	Formulation vérifiable en pratique par la livraison fréquente d'une application qui fonctionne. La livraison fréquente fait référence à la durée des cycles de développement.

Tableau 4.3 Grille d'analyse du principe 3 selon les critères individuels.

Résultat

Cette proposition satisfait les cinq critères individuels, par conséquent elle est retenue comme principe agile selon les critères d'identification.

4.3.4 Principe 4

Énoncé : *Business people and developers must work together daily throughout the project.*

Traduction : Gestionnaires et développeurs doivent travailler ensemble, de façon quotidienne, pour toute la durée du projet.

Ajustement de la traduction : Le client et les développeurs doivent travailler de concert quotidiennement durant tout le projet.

Explication

Pour qu'un projet puisse être considéré comme agile, il faut qu'il y ait une interaction permanente entre le client et les développeurs. L'absence de communication ou la communication par le biais d'intermédiaires entre les développeurs et les clients peut entraîner une mauvaise compréhension des besoins. À titre d'exemple, la méthode *XP* recommande que le client soit présent sur le site. Donc, il faut consacrer du temps à la communication entre les clients et les développeurs et cela en organisant des réunions régulières.

Certes, la détermination des besoins du client représente une des étapes les plus difficiles du développement logiciel. Ceci est rarement possible au début d'un projet : les utilisateurs ne savent en général pas ce qu'il leur faut exactement, et la communication entre les clients et les développeurs est rarement parfaite. Ce n'est qu'après avoir vu fonctionner la première mouture du programme que l'on pourra se faire une idée de ce qui doit et ne doit pas être développé. Pour cela, il faut que le client et les développeurs travaillent en étroite collaboration durant tout le projet. Il faut aussi être prêt à intégrer les changements.

L'application de ce principe au sein d'une équipe de développement permet d'améliorer la communication et le partage d'idées.

Évaluation

N° du critère	Critère satisfait	Justification
1	X	Formulation prescriptive.
2	X	Ne découle pas d'une technologie, d'une méthode, d'une technique ou d'une activité du génie logiciel.
3	X	Pas de compromis entre deux actions ou concepts.
4	X	Le client et les développeurs sont des concepts du génie logiciel. De même, la communication entre le client et les développeurs représente un concept du génie logiciel selon le guide *SWEBOK*. Ce concept est présent tout au long du processus de développement de logiciel : exigences logicielles, conception, tests, etc.
5	X	Formulation vérifiable en pratique par les rencontres fréquentes entre les clients et les développeurs en début de projet, puis les réunions régulières pour la suite du projet.

Tableau 4.4 Grille d'analyse du principe 4 selon les critères individuels.

Résultat

Cette proposition satisfait les cinq critères d'identification individuels des principes, par conséquent elle est retenue comme principe.

4.3.5 Principe 5

Énoncé : *Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.*

Traduction : Bâtir des projets autour d'individus motivés. Donner leur l'environnement et le support qu'ils nécessitent, et ayez confiance qu'ils feront le travail.

Ajustement de la traduction : Bâtir des projets autour d'individus motivés. Donnez-leur l'environnement et le support requis, et ayez confiance qu'ils feront le travail.

Explication

Dans un projet agile, les personnes sont considérées comme le facteur clé du succès, tous les autres facteurs, processus, environnement, gestion sont portés au changement s'ils se révèlent comme obstacle au bon fonctionnement de l'équipe. Selon Fowler, l'un des objectifs des méthodes traditionnelles est de développer un processus dans lequel les personnes sont remplaçables. Ainsi, les personnes sont traitées comme des ressources qui sont disponibles sous différents types : analyste, codeurs, testeurs, gestionnaire. Les individus ne sont pas si importants, seuls les rôles sont importants. Un des traits caractéristiques des méthodes agiles est qu'elles rejettent cette notion de remplaçabilité. Elles considèrent que les personnes sont le facteur le plus important du développement logiciel et, donc, la notion d'individus en tant que ressources est rejetée dans la pensée du courant agile (Fowler, 2005).

Un projet agile n'exige pas l'écriture de documents, mais l'équipe de développement peut les créer si c'est un besoin important (Figure 4.2). Les documents ne représentent pas le moyen par défaut pour transmettre de l'information au sein d'une équipe agile étant donné que le moyen par défaut est la communication et l'échange d'information.

La responsabilité des actions est confiée à des ressources humaines bien formées, relativement autonomes et rationnellement coopératives. L'environnement de travail doit encourager la créativité et le partage des idées, d'où le rôle du gestionnaire. Pour ce faire, il faut fournir et mettre en place des outils, un ensemble de méthodes et des techniques permettant d'organiser, de retenir, et de partager les connaissances au sein de l'équipe. Un autre point est que les développeurs doivent être capables de prendre toutes les décisions techniques étant donné qu'ils sont les seuls qui peuvent estimer combien de temps leur est nécessaire pour exécuter une tâche.

Évaluation

N° du critère	Critère satisfait	Justification
1	X	Formulation prescriptive.
2	X	Ne découle pas d'une technologie, d'une méthode, d'une technique ou d'une activité du génie logiciel.
3	X	Pas de compromis entre deux actions ou concepts.
4	X	La sélection du personnel motivé au sein d'un projet est un concept relié à la gestion de projet en génie logiciel selon le guide *SWEBOK*. Aussi, le choix de l'environnement, des outils et des supports de travail fait partie des concepts reliés aux outils et méthodes du génie logiciel.
5	X	La formulation du principe est vérifiable en partie en ce qui concerne l'environnement et le support de travail, mais pas la confiance.

Tableau 4.5 Grille d'analyse du principe 5 selon les critères individuels.

Résultat

Cette proposition satisfait les cinq critères individuels, par conséquent elle est retenue comme principe agile selon les critères d'identification.

4.3.6 Principe 6

Énoncé : *The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.*

Traduction : La méthode la plus efficace de transmettre l'information à l'équipe de développement et à l'intérieur de celle-ci est par conversation de personne-à-personne.

Explication

Le mode de communication par défaut au sein d'une équipe agile est la conversation face à face (Agile-Alliance, 2002).

Des documents peuvent bien sûr être rédigés, mais ils n'ont en aucun cas pour but de consigner par écrit la totalité des informations relatives au projet. Même les spécifications peuvent ne pas être écrites de manière formelle. Par exemple pour assurer le transfert de connaissances, l'*eXtreme Programming* (XP) suggère de pratiquer la programmation en dyade, en regroupant un développeur senior avec un développeur junior, plus particulièrement pour les parties les plus complexes du projet.

D'autres moyens de communication sont aussi valables, par exemple, certaines informations pertinentes doivent être notées par écrit pour le cas où on en aurait besoin plus tard. Cette documentation devrait contenir seulement les informations importantes.

Évaluation

N° du critère	Critère satisfait	Justification
1		Formulation non prescriptive.
2	X	Nous devons différencier deux types de conversation face à face : la première est pratiquée au sein de l'équipe de développement et la deuxième est pratiquée entre les clients et l'équipe de développement. Dans le premier cas, on parle ici d'une activité du développement agile qui est la pratique de programmation en dyade et ce principe découle d'une activité provenant du processus agile. Dans le deuxième cas, nous considérons que ce principe ne découle pas d'une technologie, d'une méthode, d'une technique ou d'une activité du génie logiciel.
3	X	Pas de compromis entre deux actions ou concepts.
4		Dans sa formulation, le principe n'inclut pas de concepts reliés au génie logiciel.
5	X	Formulation vérifiable en pratique par la pratique de programmation en dyade et par les rencontres régulières entre le client et l'équipe du

Tableau 4.6 Grille d'analyse du principe 6 selon les critères individuels.

Résultat

Cette proposition ne satisfait pas à deux critères des cinq critères que nous utilisons pour l'évaluation individuelle des principes et, par suite, elle est éliminée de notre liste de principes. Concernant le critère no.1, elle n'est pas formulée de façon prescriptive. Au niveau du critère no.2, nous cherchons à identifier si la proposition n'est pas associée à une méthode, à une technique ou à une activité du génie logiciel. Cette proposition découle d'une pratique exercée au sein d'une équipe agile, mais nous rappelons que dans le cadre de notre recherche nous identifions seulement les pratiques du génie logiciel définies par la norme *ISO/IEC 12207*. Ainsi, la proposition satisfait au critère no.2. Concernant le critère no.4, cette proposition est éliminée, car elle ne contient aucun concept explicite du génie logiciel.

4.3.7 Principe 7

Énoncé : *Working software is the primary measure of progress.*

Traduction : Un logiciel fonctionnel est la mesure principale de l'avancement.

Ajustement de la traduction : Un logiciel fonctionnel est la mesure primaire de l'avancement du projet.

Explication

Ce principe est relié aux premier et troisième principes. En effet, dans le cadre d'un développement itératif, le fait de disposer de logiciels fonctionnels à la fin de chaque itération est le premier indicateur d'avancement du projet. Contrairement à d'autres méthodes, l'avancement du projet ne se mesure pas à la quantité de documentation rédigée ni en termes de phase dans laquelle il se trouve, mais bien en terme de pourcentage de fonctionnalités effectivement mises en place.

Dans le cadre du développement agile de logiciel, les types de livrables dépendent du type du projet, mais l'important est de suivre les fondements de l'agilité et de produire ce qui est utile et juste nécessaire en se référant au dixième principe.

Évaluation

N° du critère	Critère satisfait	Justification
1		Formulation non prescriptive.
2	X	Ne découle pas d'une technologie, d'une méthode, d'une technique ou d'une activité du génie logiciel.
3	X	Pas de compromis entre deux actions ou concepts.
4	X	Dans sa formulation, le principe comporte « *Software* » qui est un concept du génie logiciel.
5	X	Formulation vérifiable en pratique. Chaque fonctionnalité doit être pleinement développée dans sa totalité, dans la mesure où elle doit être livrée, avant de passer à la suite. Ainsi, on peut avoir une visibilité de ce qui est achevé à un moment précis du projet.

Tableau 4.7 Grille d'analyse du principe 7 selon les critères individuels.

Résultat

A priori, cette proposition ne peut être retenue comme principe, car elle n'est pas formulée comme une proposition prescriptive, donc elle ne satisfait pas au critère no.1. Par contre, elle satisfait aux quatre autres critères d'évaluation individuels. Pour retenir cette proposition comme principe, nous avons tenté de la reformuler sous forme prescriptive. Nous proposons la reformulation mineure suivante : « *Measure the progress primarily through working software.* ». Ainsi, nous pouvons retenir ce principe agile comme principe selon les critères d'identification individuels.

4.3.8 Principe 8

Énoncé : *Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.*

Traduction : Les processus agiles favorisent le développement maintenable. Les responsables, développeurs et les usagers devraient pouvoir conserver un rythme constant indéfiniment.

Ajustement de la traduction : Les processus agiles favorisent le développement durable. Les responsables, les développeurs et les usagers devraient pouvoir maintenir un rythme constant jusqu'à la fin du projet.

Explication

Les gestionnaires, les développeurs et les utilisateurs devraient être en mesure de maintenir un rythme constant jusqu'à la fin du projet. L'équipe se doit donc d'adapter son rythme pour préserver la qualité de son travail durant toute la durée du projet.

Les développeurs doivent être capables de livrer le produit quand il est considéré comme acceptable, plutôt qu'attendre que tous les éléments destinés à être livrés soient complétés (Agile-Alliance, 2002).

Une des pratiques de l'*eXtreme Programming* exige que les semaines de travail ne comptent jamais plus de 40 heures. Chaque développeur doit gérer ses semaines pour ne pas avoir des heures supplémentaires durant deux semaines d'affilée.

Évaluation

N° du critère	Critère satisfait	Justification
1	X	Formulation prescriptive.
2	X	Ne découle pas d'une technologie, d'une méthode, d'une technique ou d'une activité du génie logiciel.
3	X	Pas de compromis entre deux actions ou concepts.
4	X	Dans sa formulation, le principe comporte l'expression « *sustainable development* » qui est un concept du génie logiciel.
5	X	Formulation vérifiable par l'application de la pratique du développement à rythme constant et par le respect d'heures de travail convenable.

Tableau 4.8 Grille d'analyse du principe 8 selon les critères individuels.

Résultat

Cette proposition est retenue comme principe agile, puisqu'elle répond aux cinq critères d'évaluation individuels.

4.3.9 Principe 9

Énoncé : *Continuous attention to technical excellence and good design enhances agility.*

Traduction : Une attention continue à l'excellence technique et un bon design augmentent l'Agilité.

Ajustement de la traduction : Une attention constante à l'excellence technique et un bon design augmentent l'agilité.

Explication

Il faut maintenir le code propre, le nettoyer et le réécrire, processus qui est désigné par le terme *refactoring* ou remaniement. Le nettoyage du code se valide lors de séances de travail collectif regroupant toute l'équipe de développement.

Fowler explique le *refactoring*[5] comme suit :

> « *Refactoring is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior.* » (Fowler, *Refactoring Home Page*)

Ainsi, le *refactoring* est défini comme une activité permanente et quotidienne qui consiste à retoucher ou réorganiser des portions existantes de code, à comportement fonctionnel identique, en vue d'améliorer la structure d'ensemble de l'application. Le *refactoring* permet d'éviter les duplications dans le code et permet de le rendre plus clair afin de faciliter le travail subséquent des développeurs.

Évaluation

N° du critère	Critère satisfait	Justification
1		Formulation non prescriptive.
2	X	Ne découle pas d'une technologie, d'une méthode, d'une technique ou d'une activité du génie logiciel.
3	X	Pas de compromis entre deux actions ou concepts.
4	X	Dans sa formulation, le principe comporte « *good design* » qui est un concept du génie logiciel.
5		Formulation non vérifiable en pratique étant donné que l'amélioration de l'agilité n'est pas testable.

Tableau 4.9 Grille d'analyse du principe 9 selon les critères individuels.

[5] Le site `refactoring.com` contient une liste de sources d'informations sur le processus de *refactorisation*.

Résultat

Cette proposition n'est pas retenue comme principe puisqu'elle ne satisfait pas à deux critères d'évaluations individuels. Elle n'est pas formulée de façon prescriptive, donc elle ne satisfait pas le critère no.1. Elle ne peut être vérifiée dans ses conséquences du fait que la formulation n'indique pas des méthodes pour atteindre un haut niveau d'agilité. Ainsi, elle ne satisfait pas le critère no.5.

4.3.10 Principe 10

Énoncé : *Simplicity—the art of maximizing the amount of work not done—is essential.*

Traduction : La simplicité — l'art de minimiser la quantité de travail fait inutilement — est essentielle.

Explication

Ce principe soutient l'idée de construire le système le plus simple répondant aux besoins actuels. Rien ne sert d'essayer d'anticiper les besoins futurs. Au contraire, il faut construire le système le plus simple répondant aux besoins actuels pour que celui-ci soit facilement adaptable dans le futur.

D'après Cockburn, la notion de simplicité est subjective et chacun peut l'interpréter à sa manière (Cockburn, 2001). Selon Larman, le principe de simplicité est appliqué non seulement pendant la phase de conception de logiciels, mais à toutes les étapes du cycle de développement d'un projet agile (Larman et Basili, 2003).

Ainsi, la conception est seulement réalisée pour les fonctionnalités existantes et non pour les fonctionnalités futures. Ceci « maximise le travail non effectué » et permet également au logiciel d'être livré plus rapidement.

Évaluation

N° du critère	Critère satisfait	Justification
1		Formulation non prescriptive.
2	X	Ne découle pas d'une technologie, d'une méthode, d'une technique ou d'une activité du génie logiciel.
3	X	Pas de compromis entre deux actions ou concepts.
4		Dans sa formulation, le principe n'inclut pas de concepts reliés au génie logiciel. Mais du point de vue de la sémantique, le principe renferme les concepts suivants (selon le guide *SWEBOK*) : Conception du logiciel et
5		Formulation non vérifiable en pratique. Mais à partir de l'explication du principe nous remarquons que la vérification de ce principe peut être faite en vérifiant si l'équipe de développement a seulement mis en œuvre les fonctionnalités qui ont été convenues et fixées avec le client et rien

Tableau 4.10 Grille d'analyse du principe 10 selon les critères individuels.

Résultat

Cette proposition n'est pas formulée de façon prescriptive, donc elle ne satisfait pas le critère no.1. La formulation de cette proposition ne renferme pas de concepts reliés au génie logiciel et, par suite, elle ne satisfait pas le critère no.4. Elle ne peut être vérifiée dans ses conséquences du fait que la formulation n'indique pas clairement comment on peut maximiser le travail non effectué. Ainsi, elle ne satisfait pas le critère no.5. Par conséquent, cette proposition n'est pas retenue comme principe agile selon les critères d'identification.

4.3.11 Principe 11

Énoncé : *The best architectures, requirements, and designs emerge from self-organizing teams.*

Traduction : Les meilleures architectures, exigences et designs surgissent d'équipes auto-organisées.

Ajustement de la traduction : Les meilleures architectures, exigences et conception émergent d'équipes auto-organisées.

Explication

Dans le cadre agile, la définition des responsabilités est confiée à toute l'équipe et les tâches sont partagées selon le principe du volontariat. Certaines équipes de projet ne sont pas assez mûres et n'ont pas le niveau nécessaire de compétence pour développer les meilleures conceptions et architectures. Pour cela, il est important que les bonnes personnes soient choisies pour ce type de projets. Les personnes doivent avoir une bonne maîtrise des techniques de conception.

Rappelons que dans une organisation traditionnelle, chaque développeur travaille sur une partie séparée de l'application. Cette séparation n'est plus appropriée dans une organisation agile. L'organisation d'une équipe agile impose de nouvelles contraintes sur le fonctionnement de l'équipe. L'assignation des tâches se fait de manière « auto-organisée » : chaque développeur sélectionne les tâches sur lesquelles il veut travailler en cours d'une itération. Cela est réalisable par le biais de certaines pratiques, par exemple, deux pratiques fondamentales de la méthode *eXtreme Programming* : la responsabilité collective du code, et le travail en binômes.

En quoi une équipe « auto-organisée » favorise-t-elle l'émergence des exigences et de la conception ? Dans un projet agile, la liste de fonctionnalités du produit ne se décide précisément qu'au fur et à mesure des itérations, par conséquent l'implémentation ne peut se baser sur une conception définie en début de projet. L'équipe doit être capable de faire

émerger la conception et les exigences tout au long du développement. Une autre pratique permettant de faire émerger une conception adaptée aux besoins du client est le *refactoring* qui permet d'éliminer au fur et à mesure tout ce anuit à la simplicité et à la clarté de l'application. Ainsi, le *refactoring* assure l'émergence progressive d'une conception propre et qui respecte les exigences du client.

Évaluation

N° du critère	Critère satisfait	Justification
1		Formulation non prescriptive.
2	X	Ne découle pas d'une technologie, d'une méthode, d'une technique ou d'une activité du génie logiciel.
3	X	Pas de compromis entre deux actions ou concepts.
4	X	Dans sa formulation, le principe renferme : « *architectures* », « *requirements* » et « *designs* ». Ces termes se trouvent au niveau de l'architecture, les exigences et de la conception de logiciel (selon le guide
5		Le principe tel que formulé est non vérifiable en pratique. Mais la vérification en pratique peut être confirmée par le fait que l'architecture n'est pas fixée au début du projet, au contraire elle évolue avec le processus de développement. Ainsi, l'équipe de développement s'auto-organise et améliore l'architecture au fur et à mesure de l'avancement.

Tableau 4.11 Grille d'analyse du principe 11 selon les critères individuels.

Résultat

Cette proposition n'est pas formulée de façon prescriptive, ainsi elle ne satisfait pas au critère no.1. Au niveau du critère no.5, la formulation de la proposition n'est pas vérifiable en pratique puisqu'elle ne fournit pas de règle d'actions. En d'autres termes, la formulation ne fournit pas comment exercer ce principe en pratique. Par conséquent, cette proposition n'est pas retenue comme principe agile, car elle ne répond pas à deux critères d'évaluation individuels.

4.3.12 Principe 12

Énoncé : *At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.*

Traduction : À intervalles réguliers, l'équipe réfléchit sur une façon de devenir plus efficace, puis adapte et ajuste son comportement en conséquence.

Ajustement de la traduction : À intervalles réguliers, l'équipe doit réfléchir à la façon de devenir plus efficace, puis adapter et ajuster son comportement en conséquence.

Explication

Ce principe est assigné directement à l'équipe de développement. Dans le contexte agile, l'équipe essaie de s'ajuster continuellement tout en demeurant dans un cadre organisationnel et, par la suite, ce principe peut être aussi lié à la notion d'apprentissage organisationnel (réorientation des compétences, systèmes d'information, motivation et promotion de l'initiative personnelle).

Une équipe agile est consciente que son environnement est en perpétuelle évolution. C'est pourquoi elle ajuste continuellement son organisation, ses règles et son fonctionnement de manière à rester agile. D'abord, l'équipe réfléchit sur les façons de devenir plus efficace, et ce en essayant constamment de comprendre ses forces et ses faiblesses, et de comprendre comment le processus de développement peut être amélioré. Ensuite, elle ajuste son comportement en conséquence. Mais, il faut aussi que le chef de projet ait une connaissance des « ressources » et des « compétences » dont il dispose pour pouvoir orienter son équipe vers les bons choix.

Citons, comme exemple de pratique « la rétrospective de *Sprint*[6] » qui est une des réunions majeures dans les pratiques de la méthode *Scrum*. Cette pratique est faite à la fin de chaque *Sprint* et a pour but l'amélioration du processus de développement pour le prochain *Sprint*. Elle permet d'inspecter un *Sprint* pour en déduire les points d'améliorations.

Évaluation

N° du critère	Critère satisfait	Justification
1		Formulation non prescriptive.
2	X	Ne découle pas d'une technologie, d'une méthode, d'une technique ou d'une activité du génie logiciel.
3	X	Pas de compromis entre deux actions ou concepts.
4		Dans sa formulation, le principe n'inclut pas de concepts reliés au génie logiciel. Mais l'adaptation du processus du génie logiciel représente un concept selon le guide *SWEBOK*. Relativement à la sémantique du principe, nous constatons qu'il renferme le concept d'adaptation.
5		Le principe tel que formulé est non vérifiable en pratique, mais tel que l'expliquent les auteurs du manifeste agile la vérification en pratique peut être faite à travers les réunions inter-équipe de développement. Les réunions doivent être mises en place afin de discuter de comment

Tableau 4.12 Grille d'analyse du principe 12 selon les critères individuels.

Résultat

Cette proposition n'est pas retenue comme principe, car elle ne répond pas à trois critères d'évaluation individuels. D'abord, la proposition n'est pas formulée de façon prescriptive, ainsi elle ne satisfait pas au critère no.1. Ensuite, la formulation de la

[6] Plus connu sous le nom de *Sprint Retrospective Meeting* réunissant l'équipe du développement pour faire un bilan du *Sprint* et identifier les problèmes rencontrés et les progrès à faire.

proposition ne comporte aucun concept explicite du génie logiciel et, par suite, elle ne satisfait pas au critère no.4. Enfin, le principe tel que formulé ne satisfait pas le critère no.5 vu qu'il est non vérifiable en pratique. On ne dispose pas de manière ni pour vérifier si l'équipe a ajusté son comportement, ni pour identifier clairement si l'équipe a fait les changements nécessaires.

4.4 Synthèse

Dans ce qui suit, nous allons analyser les résultats que nous avons obtenus afin de déterminer les principes que nous avons retenus.

Sans nous éloigner de l'objectif de notre analyse, nous proposons tout d'abord deux sections : la première section met l'accent sur le lien qui existe entre les valeurs et les principes agiles alors que la deuxième section souligne la correspondance entre les principes du manifeste agile et les concepts du génie logiciel.

4.4.1 Dépendance entre les valeurs et les principes agiles

À partir de notre étude des valeurs et des principes, nous avons pu identifier pour chaque principe l'ensemble des valeurs dont il découle. Le tableau 4.13 établit le lien qui existe entre les principes et les valeurs agiles. Un X signifie que le principe renvoie à la valeur mentionnée.

Principe / **Valeur**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
1. *Individuals and interactions over processes and tools.*				X	X	X		X			X	X
2. *Working software over comprehensive documentation.*	X		X				X		X	X	X	
3. *Customer collaboration over contract negotiation.*	X	X		X				X				
4. *Responding to change over following a plan.*	X	X	X									X

Tableau 4.13 Les valeurs agiles versus les principes agiles.

Comme on peut le constater, les douze principes du manifeste sont bien liés à des valeurs et toutes les valeurs sont couvertes par les principes.

4.4.2 Correspondance entre les principes agiles et les concepts du génie logiciel

Durant notre évaluation des principes du manifeste agile décrits dans les sections qui précèdent de ce chapitre, nous avons pu identifier plusieurs correspondances entre les douze principes du manifeste et les dix domaines de connaissance du génie logiciel tels que définis dans le *Guide to the Software Engineering Body of Knowledge (SWEBOK)*. Dans le tableau 4.14, nous mettons en correspondance ces principes et ces concepts. Un X signifie qu'un principe est lié au domaine de connaissance *SWEBOK* mentionné.

Principe / **Domaine de connaissance**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
1. Exigences du logiciel.	X	X	X	X		X					X	
2. Conception du logiciel.	X		X			X			X	X	X	
3. Construction du logiciel.	X		X			X			X	X		
4. Essai du logiciel.			X						X			
5. Maintenance du logiciel.												
6. Gestion de la configuration du												
7. Management du génie logiciel.	X		X	X	X		X	X				X
8. Processus du génie logiciel.	X		X	X			X					X
9. Outils et méthodes du génie					X							
10. Qualité du logiciel.	X		X			X			X			X

Tableau 4.14 Les concepts du génie logiciel versus les principes du manifeste agile.

On remarque que les dix domaines de la connaissance ne sont pas couverts en totalité par les principes du manifeste agile, mais chaque domaine de la connaissance est soutenu par un ou plusieurs principes agiles.

Lors du travail présenté dans ce chapitre, nous avons analysé douze principes pour n'en conserver que sept. Ces principes satisfont aux cinq critères individuels d'identification. Un seul principe a été révisé légèrement afin de le reformuler sous forme prescriptive : il s'agit du septième principe « *Working software is the primary measure of progress.* ». La reformulation que nous considérons comme mineure est la suivante : « *Measure the progress primarily through working software.* ».

Les principes du manifeste agile retenus sont donc les suivants :

- ✓ **Principe 1 :** *Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.*
- ✓ **Principe 2 :** *Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.*
- ✓ **Principe 3 :** *Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.*
- ✓ **Principe 4 :** *Business people and developers must work together daily throughout the project.*
- ✓ **Principe 5 :** *Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.*
- ✓ **Principe 6 :** *Measure the progress primarily through working software.*
- ✓ **Principe 7 :** *Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.*

À la lecture du tableau 4.15, qui synthétise les principes du manifeste agile analysés selon les critères individuels, plusieurs remarques s'imposent :

- ✓ le critère no.1, indiquant qu'un principe est une proposition prescriptive, a éliminé six propositions. Ces propositions étaient formulées de façon descriptive, mais seule le

septième principe a été reformulé sous forme prescriptive. En réalité, les principes 6, 7 et 12 étaient tous trois des candidats pour une reformulation, mais notre contrainte était la suivante : la reformulation est effectuée seulement sur les énoncés qui satisfont les quatre autres critères. Donc, seul le principe 7 a été reformulé et, par la suite, retenu comme principe.

- ✓ Le critère no.2, à savoir qu'un principe ne découle pas d'une technologie, d'une méthode, d'une technique ou d'une activité du génie logiciel, n'a écarté aucune proposition. En effet, aucun des douze principes du manifeste agile n'est associé de façon directe à une activité du génie logiciel répertoriée dans la norme *ISO/IEC 12207*.

- ✓ Le critère no.3 n'a écarté aucun principe. Les douze principes du manifeste agile satisfont ce critère.

- ✓ Le critère no.4 a éliminé quatre principes. Ces principes ne renferment aucun concept explicite du génie logiciel.

- ✓ Le critère no.5 a écarté quatre principes. Les formulations de ces principes ne permettent pas de les vérifier en pratique.

Dans le tableau 4.15, deux types de signe sont utilisés entre les principes et les critères. Un signe « + » indique que le critère est satisfait par le principe considéré. Un signe «-» indique que le principe traité ne répond pas au critère.

Critère / Principe	1	2	3	4	5	Résultat
1. *Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.*	+	+	+	+	+	Retenu
2. *Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.*	+	+	+	+	+	Retenu

Principe \ Critère	1	2	3	4	5	Résultat
3. *Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.*	+	+	+	+	+	Retenu
4. *Business people and developers must work together daily throughout the project.*	+	+	+	+	+	Retenu
5. *Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.*	+	+	+	+	+	Retenu
6. *The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.*	-	+	+	-	+	Rejeté
7. *Working software is the primary measure of progress.*	-	+	+	+	+	Retenu*
8. *Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.*	+	+	+	+	+	Retenu
9. *Continuous attention to technical excellence and good design enhances agility.*	-	+	+	+	-	Rejeté
10. *Simplicity—the art of maximizing the amount of work not done—is essential.*	-	+	+	-	-	Rejeté
11. *The best architectures, requirements, and designs emerge from self-organizing teams.*	-	+	+	-	-	Rejeté
12. *At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.*	-	+	+	-	-	Rejeté

Tableau 4.15 Résultat de l'évaluation des principes selon les critères individuels.

*avec reformulation

4.5 Conclusion

Notre démarche a fait appel aux cinq critères individuels d'identification, au manifeste agile et aux explications de leurs auteurs. Ce chapitre nous a menés à introduire les concepts du développement agile. Ensuite, nous avons pu comprendre les valeurs et les principes du manifeste agile qui sont considérés comme un noyau commun à l'ensemble des méthodes agiles. Enfin, nous avons pu identifier les énoncés qui étaient en fait des principes.

Dans le prochain chapitre, nous allons analyser plus spécifiquement les principes de quelques méthodes agiles qui se réclament des idées du manifeste agile.

CHAPITRE V

ANALYSE DES PRINCIPES DE QUELQUES MÉTHODES AGILES

« Dis-moi et j'oublierai, montre moi et je me souviendrai, implique moi et je comprendrai. »

Confucius

5.1 Introduction

Il n'existe pas une seule méthodologie ou méthode agile, mais plusieurs, notamment *eXtreme Programming*, *Scrum*, *DSDM*, *FDD* et *Crystal*. Nous avons choisi de nous focaliser sur trois méthodes parmi les plus connues et les plus utilisées : *eXtreme Programming*, *Scrum* et *DSDM*.

Les idées exprimées par le manifeste agile sont matérialisées dans les méthodes agiles. Pour cela, nous retrouvons une concordance entre ce qui est mentionné dans le manifeste agile et ce que les méthodes agiles présentent. Cependant, chaque méthode agile a ses propres principes : *eXtreme Programming* repose sur cinq principes, *Scrum* sur six principes et *DSDM* sur neuf principes. Dans ce chapitre, nous voulons vérifier si les principes de ces méthodes agiles sont vraiment des principes selon les critères d'identification individuels.

D'après Fowler, ces nouvelles méthodes constituent un compromis entre aucun processus formel et trop de processus, fournissant juste le compte nécessaire pour obtenir un succès raisonnable (Fowler, 2005).

Il existe différentes sources d'information sur les méthodes rassemblées sous le thème du développement agile. Nous faisons une étude des principes de trois méthodes agiles – *XP*, *Scrum* et *DSDM* – en nous basant sur les sources d'informations considérées comme référence pour chaque méthode citée.

Le présent chapitre est organisé de la manière suivante. Nous commençons par énumérer les valeurs et les pratiques de chaque méthode. Dans le cas de la méthode *DSDM,* nous nous contentons de présenter son cycle de vie puisqu'elle ne présentc ni valeurs, ni pratiques contrairement aux méthodes *XP* et *Scrum*. Ensuite, nous effectuons l'étude de chaque principe selon les critères d'identification utilisés pour l'analyse des principes du manifeste agile. Finalement, nous faisons une analyse des résultats obtenus.

L'étude individuelle de chaque principe comportera les informations suivantes :

- ✓ l'énoncé du principe;
- ✓ une explication du principe;
- ✓ un tableau regroupant les cinq critères individuels d'identification.

5.2 eXtreme Programming (XP)

L'*eXtreme Programming* (habituellement abrégé comme XP) est une méthode de développement de logiciel agile qui se compose, comme plusieurs autres méthodes agiles, d'une collection de valeurs, de principes et de pratiques qui forment son noyau. Elle a été mise au point à la fin des années 90 par *Kent Beck*, *Ron Jeffries* et *Ward Cunningham* (Beck, 1999). Elle est assurément la méthode agile la plus connue. Elle est conçue pour des équipes de petite taille de l'ordre d'une douzaine de personnes.

Jeffries (2006) définit l'*eXtreme Programming* comme suit :

> « *Extreme Programming is a discipline of software development based on values of simplicity, communication, feedback, and courage.* » (Jeffries, 2006)

Dans son livre « *Extreme Programming Explained: Embrace Change* » Beck décrit l'*eXtreme Programming* de la manière suivante:

> « *XP is a lightweight methodology for small-to-medium-sized teams developing software in the face of vague or rapidly changing requirements.* » (Beck, 1999)

Cette déclaration met l'accent sur l'origine et l'intention de *XP*. Selon Beck, l'e*Xtreme Programming* est « léger » compte tenu de son cycle de développement simple. Toujours d'après Beck, une méthode est souvent interprétée comme un ensemble de règles à suivre qui assure le succès du projet. Ainsi, chaque équipe *XP* atteint divers degrés de succès. *XP* est appliquée avec des équipes de petite ou de taille moyenne. Beck précise que les valeurs et les principes de l'e*Xtreme Programming* sont applicables à n'importe quelle échelle, mais les pratiques doivent être ajustées lorsque de nombreuses personnes sont impliquées (Beck, 1999).

L'*eXtreme Programming* repose sur des cycles rapides de développemcnt (voir Figure 5.1). Les étapes du cycle de vie de la méthode *XP* sont les suivantes :

1. Phase d'exploration.
2. Phase de planification.
3. Phase de mise en production.
4. Phase de maintenance.

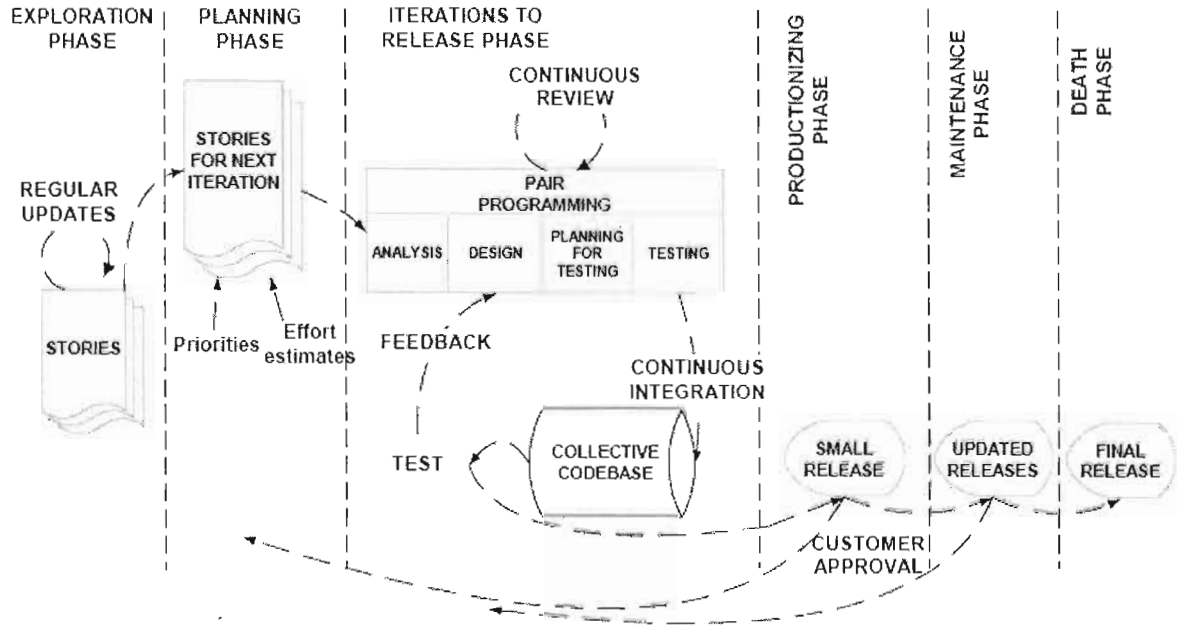

Figure 5.1 Cycle de vie de la méthode *XP* (Agile-Alliance, 2002).

La première est la phase d'exploration qui permet à l'équipe de développement de déterminer les scénarios clients qui seront fournis pendant cette itération. Ensuite, il y a la phase de planification où l'équipe transforme les scénarios en tâches à réaliser. La troisième phase est la mise en production où chaque développeur procède à la réalisation des tâches avec un binôme. Enfin, on retrouve la phase de maintenance où les tests fonctionnels sont exécutés et le produit est livré.

5.2.1 Valeurs

La méthode *eXtreme Programming* se base sur quatre valeurs qui se complètent. Elle met l'accent sur l'élément humain et sur la collaboration des membres de l'équipe en question. Les pratiques de *XP* sont appuyées par les quatre valeurs suivantes (Beck, 1999):

1. *Communication.*
2. *Simplicity.*
3. *Feed-back.*
4. *Courage.*

Première valeur

Énoncé : *Communication.*

Explication

La communication a pour but de permettre le partage de l'information entre les divers intervenants du projet. Entre les développeurs, la communication est soulignée par la programmation en binôme, entre les développeurs et les clients par les tests et les spécifications. La méthode *XP* privilégie la communication directe plutôt que l'échange de courriels ou de documents formels.

Deuxième valeur

Énoncé : *Simplicity.*

Explication

L'*eXtreme Programming* soutient l'idée de concevoir le système le plus simple possible répondant au besoin des utilisateurs actuels et après, si nécessaire, d'ajouter des fonctionnalités supplémentaires.

Comme nous l'avons mentionné dans le dixième principe du manifeste agile au chapitre précédent (section 4.3), la simplicité touche le processus lui-même de la méthode *XP*. Il faut focaliser les efforts sur les fonctions prioritaires. Ce processus est guidé par le principe suivant (*YAGNI*) : « *You ain't gonna need it* ». Ce dernier suggère aux programmeurs de n'ajouter une fonctionnalité au programme que si elle est nécessaire. En accord avec le principe *YAGNI,* Jeffries écrit :

> « *Always implement things when you actually need them, never when you just foresee that you need them.* » (Jeffries, 2006)

Jeffries souligne que le fait de créer des fonctionnalités sans en avoir besoin représente une dépense de temps de codage, de tests et de maintenance inutile (Jeffries, 2006).

Troisième valeur

Énoncé : *Feed-back.*

Explication

Le feed-back se fait à travers les tests unitaires et donne la possibilité au client d'avoir une vision du système final. Le feed-back permet aussi de repérer et intégrer les changements.

Nous remarquons l'importance de cette valeur à trois niveaux du processus *XP*. Au niveau du *story* (cycle d'un jour) : le développeur et le *Product Owner* constatent, grâce au feed-back, si ce qui a été produit est accepté ou non, sans attendre la fin du sprint. Ensuite au niveau du *sprint* (cycle d'une semaine) : le développeur, le *Product Owner* et le gestionnaire découvrent l'avancement concret du produit, ce qui est généralement plus visible qu'un avancement au niveau d'un document de conception. Enfin, au niveau du *release* (cycle d'un mois) : le développeur, le *Product Owner*, le gestionnaire et l'utilisateur vérifient concrètement la valeur apportée au produit et spécifient les *stories* qui restent pour la suite du développement.

Quatrième valeur

Énoncé : *Courage.*

Explication

Le courage réside dans le fait que l'équipe de développement accepte de changer de vision du produit et de remettre en cause son travail pour pouvoir faire un code simple. Il faut aussi accepter que les autres le remettent aussi en cause, ce qui permet de prendre le recul nécessaire.

Comme le mentionne Beck (1999), certains changements demandent beaucoup de courage. Il faut parfois changer l'architecture d'un projet, jeter du code pour en produire un meilleur ou essayer une nouvelle technique. C'est difficile, mais la simplicité, le *feed-back* et la communication rendent ces tâches possibles (Beck, 1999). Ainsi, les valeurs de *XP* sont là pour se renforcer mutuellement.

5.2.2 Pratiques

Aux quatre valeurs citées précédemment s'ajoutent douze pratiques. Beck présente les pratiques agiles comme suit :

> « *We have to have a more concrete guide leading us to practices that satisfy and embody the four values of communication, simplicity, feedback, and courage.* »
> (Beck, 1999)

Les douze pratiques de la méthode *XP* sont subdivisées en trois catégories selon leur type d'usage (Figure 5.2) :

1. des pratiques de conception et de programmation;
2. des pratiques d'équipe ou collaboratives;
3. des pratiques de processus ou de pilotage du projet.

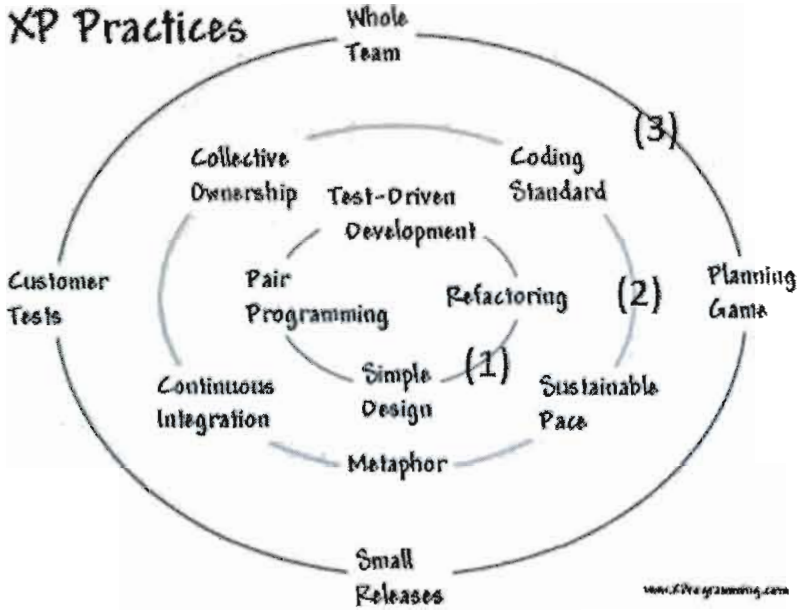

Figure 5.2 Les pratiques d'eXtreme Programming (Jeffries, 2006).

1. Les pratiques de conception et de programmation
   - « *Simple Design* » : se base sur le célèbre acronyme « *You ain't gonna need it* » qui signifie que l'attention doit être mise sur les besoins actuels du client.
   - « *Testing* » : des tests automatisés sont écrits avant le code. Ces tests sont exécutés à chaque intégration et après chaque *refactoring* du code.
   - « *Refactoring* » : permet d'améliorer la qualité du code sans modifier son comportement. Cela nécessite de mener des tests de non-régression, car le *refactoring* n'est pas sans risque.
2. Les pratiques d'équipe ou collaboratives
   - « *Metaphor* » : est utilisée pour modéliser conceptuellement le produit dans le but de simplifier les fonctionnalités. Elle est faite à l'aide du vocabulaire du client. La plupart des métaphores concernent le fonctionnement de l'application ou son architecture.

- « *Pair Programming* » : le code est toujours écrit par deux développeurs afin d'avoir une relecture du code en permanence. Les binômes changent au cours du projet pour permettre le partage des connaissances.

- « *Collective ownership* » : chaque développeur doit pouvoir modifier des lignes de code si nécessaire pour ajouter des fonctionnalités et corriger des bogues.

- « *Continuous Integration* » : le système est composé de plusieurs modules qui sont assemblés et testés un à un. Ceci permet d'obtenir un système stable et opérationnel.

- « *Coding Standards* » : les programmeurs codent dans un style uniforme, en respectant des règles de nommages des variables, méthodes, objets, classes, etc.

3. Les pratiques du processus ou de pilotage du projet
   - « *Planning Game*[7] » : vise la définition des scénarios utilisateurs prioritaires par le client. Les développeurs discutent le contenu de ces scénarios, définissent les tâches et estiment ces tâches en termes de temps/effort.

   - « *Small Releases* » : consiste à fournir rapidement des fonctionnalités utilisables au client pour avoir son feed-back. Le rythme des livraisons doit être le plus court possible pour réduire au minimum l'incompréhension entre le client et les développeurs. La satisfaction du client et la motivation des développeurs représentent les principaux effets positifs de la livraison fréquente.

   - « *40-Hour Week* » : les semaines de travail ne doivent avoir que 40 heures. Chaque développeur ne doit pas faire d'heures supplémentaires durant deux semaines de suite.

---

[7] Cette pratique a fait l'objet d'un livre écrit par Beck et Fowler intitulé « *Planning Extreme Programming* ».

- « *On-Site Customer* » : un représentant du client doit être disponible pendant la durée du projet. Il est considéré comme une ressource dédiée au projet et chargée de déterminer les besoins et de fixer les priorités.

5.2.3 Principes

Des quatre valeurs citées ci-dessus découlent cinq principes d'*eXtreme Programming*. Sur le site officiel de la méthode *XP* (Wells, 2001), nous remarquons qu'il n'y a pas la mention de principe. Wells parle de règles plutôt que de principes.

Dans son livre Beck dresse la liste de ce qu'il appelle « principe de base de la méthodologie de développement *XP* » et affirme que les valeurs sont trop vagues pour nous aider à décider des pratiques à utiliser, c'est pour cela que les valeurs sont raffinées en principes concrets que l'on peut appliquer (Beck, 1999).

Les principes de base d'*eXtreme Programming* sont les suivants :

1. *Rapid feedback*
2. *Assume simplicity*
3. *Incremental change*
4. *Embrace change*
5. *Quality work*

Dans une autre référence parue un peu plus tard, nous retrouvons la liste des principes d'eXtreme Programming suivants (Cohn, 2004) :

1. *Provide rapid feedback to its customers and lears from that feedback.*
2. *Favor simplicity and always attempts a simple solution before moving to a more complex one.*
3. *Improve the software through small, incremental changes.*
4. *Embrace change because they know they are truly adept at accommodating and adapting.*

5. *Insist that the software consistently exhibits the highest level of quality workmanship.*

Nous allons établir notre analyse des principes d'*eXtreme Programming* sur les cinq principes présentés par Cohn. Nous trouvons que ces derniers ne diffèrent pas au niveau du fond des principes de Beck, mais ajoutent de l'information.

Il existe aussi une liste de principes moins centraux et qui peuvent être utiles pour indiquer à l'équipe de développement quoi faire dans des situations spécifiques. Cette liste ne sera pas prise en considération lors de notre analyse des principes de *XP*.

Principe 1

Énoncé : *Provide rapid feedback to its customers and learn from that feedback.*

Explication

Beck explique ce principe comme suit :

> « *[...] the principle is to get feedback, interpret it, and put what is learned back into the system as quickly as possible.* » (Beck, 1999)

Le client doit être présent avec l'équipe de développement afin d'atteindre une réactivité idéale. Cette présence est aussi bien nécessaire pour la définition des besoins que pour la validation des livraisons. L'équipe de développement travaille en collaboration avec le client, cherche à comprendre ses besoins et analyse ses exigences : toutes ces actions favorisent les possibilités d'un *feed-back* rapide.

Pour mieux appliquer ce principe, Beck recommande les itérations de courte durée pour pouvoir impliquer au maximum le client (Beck, 1999). Le *feed-back* est fait dans les deux sens : un retour vers le développeur permet un changement d'orientation plus rapide et un retour vers le client lui permet de tester en permanence si le système correspond à ses attentes.

Évaluation

N° du critère	Critère satisfait	Justification
1	X	Formulation prescriptive.
2	X	Ne découle pas d'une technologie, d'une méthode, d'une technique ou d'une activité du génie logiciel.
3	X	Pas de compromis entre deux actions ou concepts.
4	X	La rétroaction est utile pour identifier des améliorations aux processus. La rétroaction est un concept du génie logiciel mentionné au neuvième chapitre « Processus du génie logiciel » du guide *SWEBOK*.
5	X	Formulation vérifiable en pratique par le retour d'information effectué dans les deux sens (vers le client et vers les développeurs).

Tableau 5.1 Grille d'analyse du principe 1 de *XP* selon les critères individuels.

Résultat

Cette proposition satisfait les cinq critères d'identification individuels des principes, par conséquent elle est retenue comme principe.

Principe 2

Énoncé : *Favor simplicity and always attempts a simple solution before moving to a more complex one.*

Explication

L'*eXtreme Programming* recommande de se pencher tout d'abord sur la solution la plus simple pour pouvoir l'améliorer par la suite. D'après Beck, le deuxième principe est le plus difficile à appliquer pour un programmeur du fait que ce dernier doit mener à terme les tâches les plus simples du moment présent puis songer à rajouter de la complexité dans un avenir où il en a besoin (Beck, 1999).

Évaluation

N° du critère	Critère satisfait	Justification
1	X	Formulation prescriptive.
2	X	Ne découle pas d'une technologie, d'une méthode, d'une technique ou d'une activité du génie logiciel.
3	X	Pas de compromis entre deux actions ou concepts.
4		Le principe n'inclut pas des concepts reliés au génie logiciel.
5		Formulation non vérifiable en pratique.

Tableau 5.2 Grille d'analyse du principe 2 de *XP* selon les critères individuels.

Résultat

Cette proposition ne satisfait pas à deux critères des cinq critères d'évaluation individuelle, ainsi elle est éliminée de notre liste de principe. Au niveau du critère no.4, nous cherchons à identifier si la proposition contient des concepts explicites du génie logiciel. Dans sa formulation cette proposition ne renferme aucun concept relié au génie logiciel, ainsi elle ne satisfait pas au critère no.4. Concernant le critère no.5, cette proposition est éliminée, car elle ne peut être vérifiée dans ses conséquences du fait que sa formulation n'indique pas clairement comment on peut appliquer la notion de simplicité.

Principe 3

Énoncé : *Improve the software through small, incremental changes.*

Explication

Procéder par des changements incrémentaux est plus fonctionnel. Ceci est applicable à plusieurs niveaux dans un projet *XP*, l'architecture et le planning changent petit à petit.

Évaluation

N° du critère	Critère satisfait	Justification
1	X	Formulation prescriptive.
2	X	Ne découle pas d'une technologie, d'une méthode, d'une technique ou d'une activité du génie logiciel.
3	X	Pas de compromis entre deux actions ou concepts.
4	X	Dans sa formulation, le principe comporte les termes « *software* » et « *change* » qui sont deux concepts du génie logiciel (selon le guide
5	X	Formulation vérifiable en pratique par la réalisation des changements incrémentaux.

Tableau 5.3 Grille d'analyse du principe 3 de *XP* selon les critères individuels.

Résultat

Cette proposition satisfait les cinq critères d'identification individuels des principes, par conséquent elle est retenue comme principe.

Principe 4

Énoncé : *Embrace change because they know they are truly adept at accommodating and adapting.*

Explication

Les besoins du client évoluent au fil du temps, l'environnement d'affaire change constamment et de nouveaux intervenants peuvent être ajoutés au cours du projet, par conséquent le processus de développement doit refléter cette réalité. Une approche de gestion de changement est proposée dans ces circonstances « *The Agile Change Management Process* » (Ambler, 2005).

Évaluation

N° du critère	Critère satisfait	Justification
1	X	Formulation prescriptive.
2	X	Ne découle pas d'une technologie, d'une méthode, d'une technique ou d'une activité du génie logiciel.
3	X	Pas de compromis entre deux actions ou concepts.
4	X	Dans sa formulation, le principe comporte le terme « *change* » qui est un concept du génie logiciel (selon le guide *SWEBOK*).
5		Formulation non vérifiable en pratique.

Tableau 5.4 Grille d'analyse du principe 4 de *XP* selon les critères individuels.

Résultat

Cette proposition ne peut être retenue comme principe. Elle ne satisfait pas au critère no.5 vu que sa formulation n'indique pas comment s'adapter aux changements qui s'opèrent durant la réalisation d'un projet agile.

Principe 5

Énoncé : *Insist that the software consistently exhibits the highest level of quality workmanship.*

Explication

Dans son livre, Beck écrit ce qui suit :

> « *We use four variables to help us think about how to control a project: cost, quality, time, and scope. They are interrelated but affect each other in strange ways.* » (Beck, 1999)

Beck explique qu'un travail est défini par quatre variables : coût, qualité, délais et taille. Il insiste sur le fait que la qualité n'est pas indépendante des autres variables. La qualité est difficile à contrôler, car il faut faire attention aux coûts. La réduction de la taille du projet augmente généralement la qualité. Prendre plus de temps peut améliorer la qualité du projet, mais cela ne respecte pas l'esprit de l'*eXtreme Programming* qui encourage les courtes itérations (Beck, 1999).

Évaluation

N° du critère	Critère satisfait	Justification
1	X	Formulation prescriptive.
2	X	Ne découle pas d'une technologie, d'une méthode, d'une technique ou d'une activité du génie logiciel.
3	X	Pas de compromis entre deux actions ou concepts.
4	X	Dans sa formulation, le principe comprend « *quality* » qui est un concept du génie logiciel (selon le guide *SWEBOK*).
5	X	Formulation vérifiable en pratique.

Tableau 5.5 Grille d'analyse du principe 5 de *XP* selon les critères individuels.

Résultat

Cette proposition satisfait les cinq critères d'identification individuels des principes, par conséquent elle est retenue comme principe.

5.2.4 Synthèse

Dans la section 5.2, nous avons appliqué les cinq critères d'évaluation individuels aux principes de l'*eXtreme Programming* dans le but de vérifier si l'on peut les considérer comme étant des principes. Il s'ensuit que seulement trois des cinq principes représentent effectivement des principes. Cependant, nous avons aussi vu qu'il existe une concordance entre les principes du manifeste agile et les pratiques d'*eXtreme Programming*. En effet, les principes du manifeste agile se concrétisent dans les pratiques des méthodes agiles. Afin de souligner ce lien important, nous avons dressé le tableau 5.6 en nous basant sur les explications fournies par les auteurs du manifeste agile.

5.2.4.1 Les principes agiles versus les pratiques de XP

Comme nous l'avons mentionné au quatrième chapitre, le manifeste agile est un ensemble de valeurs et principes considéré comme base commune de toutes les méthodes agiles. Le tableau 5.6 présente une correspondance entre les douze principes du manifeste agile analysés dans le chapitre 4 (section 4.3) et les pratiques d'*eXtreme Programming* présentées dans le présent chapitre (section 5.2.2) *(Larman et Basili, 2003)*. Chaque X indique l'existence d'un lien qui relie un principe agile avec la pratique XP correspondante.

Principe / Pratique XP	1	2	3	4	5	6	7	8	9	10	11	12
1. *Planning game.*	X	X		X								
2. *Small, frequent releases.*	X		X									
3. *System metaphors.*	X											
4. *Simple design.*									X	X		
5. *Testing.*	X	X					X		X			
6. *Frequent refactoring.*									X	X		
7. *Pair programming.*						X						
8. *Team code ownership.*									X			
9. *Continuous integration.*	X	X	X									
10. *Sustainable pace.*	X		X					X				
11. *Whole team together.*	X			X		X					X	X
12. *Coding standards.*									X			

Tableau 5.6 Les pratiques de l'*eXtreme Programming* versus les principes du manifeste agile.

Les principes de l'*eXtreme Programming* retenus sont les suivants :

- ✓ **Principe 1:** *Provide rapid feedback to its customers and learn from that feedback.*
- ✓ **Principe 2:** *Improve the software through small, incremental changes.*
- ✓ **Principe 3:** *Insist that the software consistently exhibits the highest level of quality workmanship.*

Suite à cette phase d'analyse, nous constatons que :

- ✓ le critère no.1, indiquant qu'un principe est une proposition prescriptive, n'a éliminé aucune proposition. Ces propositions étaient formulées de façon prescriptive.
- ✓ Le critère no.2, à savoir qu'un principe ne découle pas d'une technologie, d'une méthode, d'une technique ou d'une activité du génie logiciel, n'a écarté aucune proposition. Aucune proposition n'est associée à une activité du génie logiciel répertoriée dans la norme *ISO/IEC 12207*.
- ✓ Le critère no.3 n'a écarté aucun principe. Les cinq propositions ne présentent pas de dosage entre deux actions ou concepts.
- ✓ Le critère no.4 a éliminé un seul principe. Le deuxième principe ne renferme aucun concept explicite du génie logiciel.
- ✓ Le critère no.5 a écarté deux principes. Les formulations de ces principes ne permettent pas de les vérifier et les tester en pratique.

Principe \ Critère	1	2	3	4	5	Résultat
1. *Provide rapid feedback to its customers and learn from that feedback.*	+	+	+	+	+	Retenu
2. *Favor simplicity and always attempts a simple solution before moving to a more complex one.*	+	+	+	-	-	Rejeté
3. *Improve the software through small, incremental changes.*	+	+	+	+	+	Retenu
4. *Embrace change because they know they are truly adept at accommodating and adapting.*	+	+	+	+	-	Rejeté
5. *Insist that the software consistently exhibits the highest level of quality workmanship.*	+	+	+	+	+	Retenu

Tableau 5.7 Résultat de l'évaluation des principes de *XP* selon les critères individuels.

Nous avons utilisé deux types de signe entre les principes et les critères. Un signe « + » indique que le critère est satisfait par le principe considéré. Un signe « - » indique que le principe traité ne répond pas au critère.

5.3 Scrum

La méthode *Scrum* est une méthode de gestion de projets, décrite en 2001 par *Schwaber* et *Beedle* dans le livre « *Agile software developement with Scrum* » (Schwaber et Beedle, 2001). Initialement, les fondements et la formalisation de cette méthode ont été présentés par *Schwaber*, l'un des auteurs du manifeste agile, dans un article intitulé « *Scrum development process* » (Schwaber, 1995). Et bien avant cela, *Schwaber* a co-développé le processus *Scrum* avec *Sutherland* au début des années 1990 et depuis ce moment, il l'a utilisé afin d'aider les organisations travaillant sur des projets de développement complexes (Schwaber, 2004). Mais le terme Scrum apparaît pour la première fois dans un article en janvier 1986 : « *The New New Product Development Game* » écrit par Hirotaka Takeuchi et Ikujiro Nonaka, qui cite de nouvelles pratiques de développement de produits (Takeuchi et Nonaka, 1986).

Pour la rédaction de cette section, nous nous sommes basés sur les références que nous venons de citer ainsi que sur le premier ouvrage en français dédié à *Scrum* : « Le guide pratique de la méthode agile la plus populaire » écrit par Claude Aubry (Aubry, 2010). Ce livre rassemble une présentation détaillée des pratiques de *Scrum*.

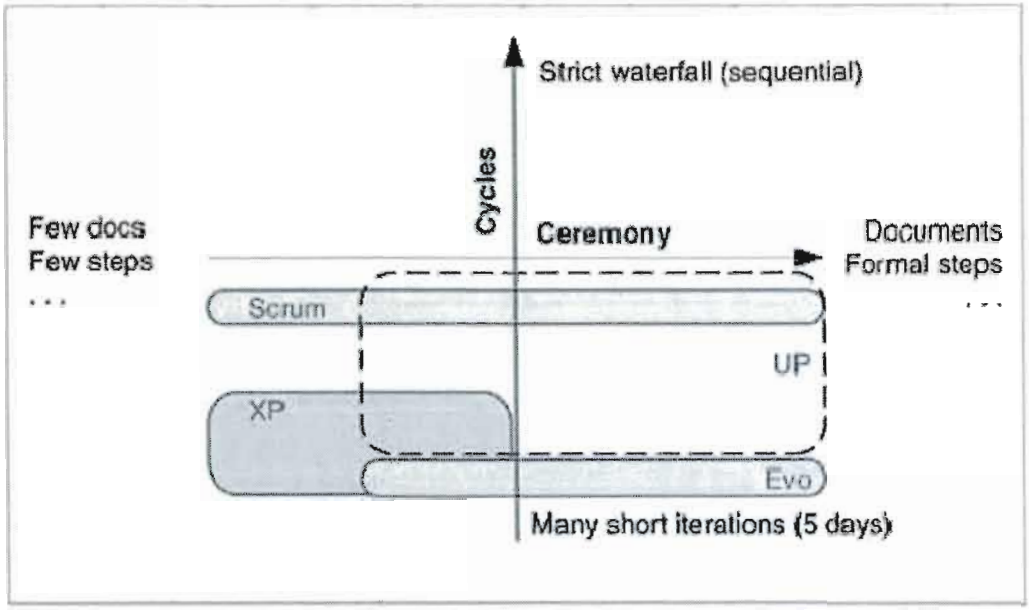

Figure 5.3 Positionnement de *Scrum* par rapport aux autres méthodes agile (Larman, 2003).

La figure 5.3 présente la position que *Scrum* occupe par rapport aux autres méthodes agiles. L'axe des ordonnées désigne le nombre de cycle et d'itérations et l'axe d'abscisse représente le nombre de documents à produire ainsi que les étapes à suivre.

5.3.1 Vue d'ensemble

Schwaber affirme qu'il est important de regarder *Scrum* dans son ensemble, et non en prenant chacun de ses éléments un à un (Schwaber, 1995). Le cycle de vie d'un projet *Scrum* est composé de trois phases comme le montre la figure 5.4 :

- ✓ Planification + Architecture
- ✓ Développement
- ✓ Test + Intégration

La planification comprend la définition des exigences définies comme des éléments d'une liste appelée « *backlog* de produit ». Elle comporte aussi la définition de la date de livraison des fonctionnalités d'une ou plusieurs itérations. Une fois que l'itération est choisie

une équipe responsable du projet est désignée. Le projet est divisé en itérations de 30 jours appelées des « *sprints* ». La phase de développement comporte l'implémentation du système. Le travail dans le *sprint* est divisé en blocs quotidiens. Chaque jour, les membres de l'équipe se réunissent pour mettre à jour le statut du *sprint* et choisissent les tâches à développer par la suite. Pendant la dernière phase de test et d'intégration, le système devrait être déployé et une documentation finale produite.

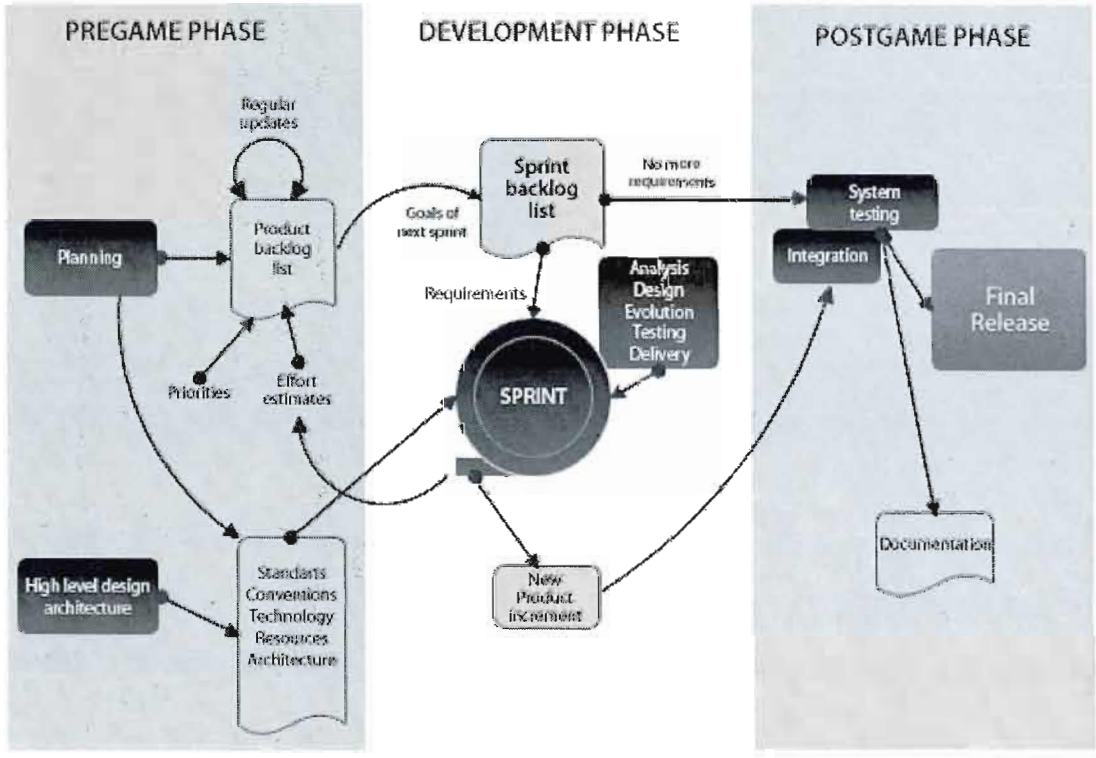

Figure 5.4 Cycle de vie de la méthode *Scrum* (Agile-Alliance, 2002).

5.3.2 Valeurs

Les cinq valeurs de *Scrum* sont (Schwaber et Beedle, 2001) :

1 *Commitment*
2 *Focus*
3 *Openness*
4 *Respect*
5 Courage

Chaque valeur est interprétée différemment : les clients et l'équipe de développement doivent s'approprier ces valeurs et décider ce que signifient ces valeurs pour eux.

Première valeur

Énoncé : *Commitment.*

Explication

Dans un projet de développement, toute relation entre le client et l'équipe doit se baser sur un engagement. Le périmètre de cet engagement est défini avant le démarrage du projet. Le client établit avec les développeurs un plan de livraison qui regroupe les différents scénarios qui seront réalisés. L'ensemble de ces scénarios constitue le périmètre du projet. De cela découlent toutes les autres valeurs : l'engagement renforce la concentration, l'ouverture, le respect et le courage.

Deuxième valeur

Énoncé : *Focus.*

Explication

Le but principal du *ScrumMaster*[8] est d'isoler l'équipe des contraintes extérieures pendant l'itération. Il est donc responsable de détecter et résoudre les problèmes. Ceci permet aux développeurs de se concentrer pleinement sur les fonctionnalités à développer pendant l'itération sans être dérangés par le client.

Troisième valeur

Énoncé : *Openness.*

Explication

Le projet doit être visible à chaque intervenant à travers des outils de gestion de projet agile. Ainsi, chaque acteur pourra voir combien de fonctionnalités ont été réalisées, le but de chaque itération et qui travaille sur quoi. Parmi les outils de gestion de projet *Scrum*[9], on peut citer : *Agilefant*, *IceScrum*, *Agilo*, *eXPlainPMT* et *XPlanner*, tous *sont open source*. Il existe une étude comparative de ces outils (Swanson, 2009).

---

[8] Le *ScrumMaster* est aussi connu sous ces différentes appellations : *Coach agile*, Facilitateur de processus et chef de projet agile.

Quatrième valeur

Énoncé : *Respect.*

Explication

Les développeurs doivent respecter les principes d'un projet *Scrum*. Le projet doit respecter l'approche de développement incrémental et itératif. *Scrum* se base sur de brèves itérations de développement appelées « *sprint* ». À la fin de chaque itération, le logiciel est testé. L'itération ne démarre que si les spécifications sont complétées. Le client ne doit pas perturber le travail de l'équipe durant une itération et l'équipe doit respecter les délais de livraison.

Cinquième valeur

Énoncé : *Courage.*

Explication

Le courage dans un projet *Scrum* se résume au courage de prendre les bonnes décisions, que ce soit de la part du client ou des développeurs. Au démarrage du projet, les développeurs doivent avoir du courage pour commencer un projet sans conception détaillée. Au cours du projet, le courage est aussi nécessaire dans le cas où l'équipe est obligée de jeter du code inutile ou trop complexe. Le client a aussi besoin de courage pour communiquer son mécontentement à chaque fin d'itération.

5.3.3 Pratiques

Les principales pratiques de *Scrum* incluent ce qui suit (Leffingwell, 2007) :

1. *Cross-functional and collocated teams of eight or fewer team members develop software in sprints.*
2. *Sprints are iterations of a fixed 30-day duration. Each sprint delivers incremental, tested functionality of value to the user.*
3. *Work within a sprint is fixed. Once the scope of a sprint is committed, no additional functionality can be added except by the development team.*
4. *The Scrum Master mentors and manages the self-organizing and self-managing teams that are responsible for delivery of successful outcomes at each sprint.*
5. *All work to be done is carried as Product Backlog, which includes requirements to be delivered, defect workload, as well as infrastructure and design activities.*
6. *The Product Backlog is developed, managed, and prioritized by the Product Owner, who is an integral member of the team and who has the primary responsibility of interfacing with the external customers.*
7. *A daily 15-minute stand-up meeting, or "daily Scrum," is a primary communication method.*
8. *Scrum focuses heavily on time-boxing. Sprints, stand-up meetings, release review meetings, and the like are all completed in prescribed times.*
9. *Scrum allows requirements, architecture, and design to emerge over the course of the project.*

Ces pratiques utilisent un vocabulaire (figure 5.5) spécifique à la méthode *Scrum*. Nous présentons dans ce qui suit quelques principales définitions des termes utilisés. Un *Sprint* est une itération de 30 jours. Le *Product backlog* représente le référentiel des exigences. Le *Sprint backlog* est un plan d'itération. Il contient la liste de tâches que l'équipe s'engage à livrer à la fin du sprint. Le *Product Owner* est le représentant de tous les intéressés : clients, utilisateurs, marketing, direction, etc. Il détermine les caractéristiques du produit, fixe la date de lancement et définit le contenu (*Product Backlog*). Le *ScrumMaster* est le chef de projet *Scrum*. Il organise les réunions et assure une collaboration étroite avec le *Product Owner*. Le *Scrum daily meeting* est une réunion quotidienne de 15 minutes (*Standing Meeting* en suivant la méthode *XP*). Le *Sprint review meeting* est réunion de revue d'itération. Le *Sprint retrospective meeting* est une réunion de bilan d'itération.

Rôles	Blocs de temps	Artefacts
• *Product Owner*	• *Planification de release*	• *Product backlog*
• *ScrumMaster*	• *Planification de sprint*	• *Plan de release*
• Équipe	• *Scrum daily meeting*	• *Plan de sprint*
	• *Sprint review meeting*	• *Burndown de sprint*
	• *Sprint retrospective*	• *Burndown de release*

Figure 5.5 Éléments de *Scrum* (Aubry, 2010).

5.3.4 Principes

Les principes de *Scrum* se retrouvent dans une première publication datant de 1986 (Takeuchi et Nonaka, 1986). Dans cet article, ces principes ont été identifiés comme étant des facteurs indispensables dans un processus de développement innovant, rapide et flexible. Par la suite, ils ont été intégrés au processus *Scrum* pour constituer les six principes fondateurs de *Scrum* :

1. *Built-in instability*
2. *Self-organizing project teams*
3. *Overlapping development phases*
4. *Multilearning*
5. *Subtle control*
6. *Organizational transfer of learning*

Selon Schwaber et Beedle, tous ces principes de base peuvent se résumer dans l'idée suivante : « Toutes les méthodes de développement basées sur la planification échouent régulièrement à produire des systèmes logiciels respectant les contraintes de coûts, de délais et de fonctionnalités prévues initialement. Plutôt que de définir à l'avance le processus de développement, offrons un moyen de le piloter en faisant en sorte de faire émerger une organisation qui soit la plus performante possible. » (Schwaber et Beedle, 2001).

Trois de ces principes se rapportent directement au processus décisionnel de l'organisation : *Built-in instability*, *Self-organizing project teams* et *Organizational transfer of learning*. Les gestionnaires du projet présentent un objectif général et donnent à l'équipe responsable la liberté pour réaliser ce but. Ceci nécessite que l'équipe s'auto-organise pour atteindre ses buts.

Nous allons procéder de la même manière que pour XP pour l'analyse individuelle des principes de *Scrum*, à savoir l'application des critères d'identification individuelle des principes.

Principe 1

Énoncé : *Built-in instability*

Traduction : Instabilité intrinsèque.

Explication

L'instabilité signifie qu'il faut donner à l'équipe un objectif précis et qui présente un défi. Takeuichi et Nonaka affirment que la créativité est redoublée s'il y a un défi.

Cette instabilité est observée au sein des équipes *Scrum*. L'organisation de l'équipe procure une certaine liberté, mais ceci crée également de la tension. Dans ces conditions, il faut appliquer le cinquième principe de *Scrum* dans le but d'empêcher l'échec du projet. Les *Planning Meetings*, les *Daily Meetings* et les *Retrospective Meetings* représentent les mécanismes de contrôle pour les équipes *Scrum*. Ces mécanismes sont facilités par le *ScrumMaster*.

Évaluation

N° du critère	Critère satisfait	Justification
1		Formulation non prescriptive.
2	X	Ne découle pas d'une technologie, d'une méthode, d'une technique ou d'une activité du génie logiciel.
3	X	Pas de compromis entre deux actions ou concepts.
4		Le principe n'inclut pas des concepts reliés au génie logiciel.
5		Formulation non vérifiable en pratique.

Tableau 5.8 Grille d'analyse du principe 1 de *Scrum* selon les critères individuels.

Résultat

Cette proposition ne satisfait pas à trois critères des cinq critères d'évaluation individuelle, ainsi elle est éliminée de notre liste de principe : elle n'est pas formulée de façon prescriptive, elle ne renferme aucun concept relié au génie logiciel et elle ne peut être vérifiée dans ses conséquences.

Principe 2

Énoncé : *Self-organizing project teams*

Traduction : Équipes auto-organisées.

Explication

C'est le principe qui est à la base de toutes les méthodes agiles, y compris *Scrum*. En effet, le manifeste agile inclut « *The best architectures, requirements, and designs emerge from self-organizing teams* » parmi ses douze principes (Beck *et al.*, 2001). L'équipe s'auto-organise[10] de manière à trouver une organisation optimale. La prise de décision est laissée à l'équipe de développement. C'est ainsi que tout le monde se sent impliqué et motivé.

Une équipe auto-organisée sort du cadre traditionnel strict où l'initiative individuelle n'est pas acceptée. Dans *Scrum*, ce n'est plus le rôle des chefs de projet d'estimer le temps nécessaire pour coder une nouvelle fonction. Le chef de projet demande à son équipe, à ses développeurs, de réaliser une estimation.

---

[10] « *Succeeding with Agile: Software Development Using Scrum* », écrit par *Mike Cohn*, décrit comment un chef de projet peut mener une équipe à s'auto-organiser.

Selon Aubry, le *ScrumMaster* fait tout pour que l'équipe prenne de l'autonomie, il l'aide à apprendre *Scrum* et à mettre en place les meilleures pratiques d'ingénierie. Il ajoute que cela demande beaucoup d'investissement, en particulier au début du projet. Si l'équipe réussit à s'auto-organiser, elle aura moins besoin du *ScrumMaster*. Il précise que dans le cas d'une petite équipe (jusqu'à trois personnes) l'équipe peut se passer d'un *ScrumMaster* (Aubry, 2010).

Évaluation

N° du critère	Critère satisfait	Justification
1		Formulation non prescriptive.
2	X	Ne découle pas d'une technologie, d'une méthode, d'une technique ou d'une activité du génie logiciel.
3	X	Pas de compromis entre deux actions ou concepts.
4	X	Dans sa formulation cette proposition renferme le terme « *Team* » est un concept du génie logiciel selon le guide *SWEBOK*.
5		Formulation non vérifiable en pratique.

Tableau 5.9 Grille d'analyse du principe 2 de *Scrum* selon les critères individuels.

Résultat

Cette proposition ne satisfait pas à deux critères des cinq critères d'évaluation individuelle, ainsi elle est éliminée de notre liste de principe. Elle n'est pas formulée de façon prescriptive, ainsi elle ne satisfait pas au critère no.1. Concernant le critère no.5, cette proposition est éliminée, car elle ne peut être vérifiée dans ses conséquences.

Principe 3

Énoncé : *Overlapping development phases.*

Traduction : Recouvrement des phases du cycle de développement.

Reformulation : *Use overlapping development phases.*

Explication

Dans un contexte d'environnement instable où les exigences changent, l'équipe doit gérer le changement. La plupart des problèmes dans le développement de produit surviennent dans la séparation des phases de développement. Dans ces conditions, il est recommandé d'appliquer ce principe qui élimine les notions traditionnelles concernant la répartition des tâches. Takeuichi et Nonaka (1986) présentent une typologie des différents modes de coordination des phases d'un projet. Le recouvrement des différentes phases du cycle de développement est mentionné comme l'un des premiers éléments novateurs de leurs observations. Ils ont constaté que des compagnies comme *Xerox* et *HP* qui utilisaient des itérations dans leur processus de développent, et ce dès les années 1980, ont obtenu de meilleurs résultats (Takeuchi et Nonaka, 1986).

Évaluation

N° du critère	Critère satisfait	Justification
1		Formulation non prescriptive.
2	X	Ne découle pas d'une technologie, d'une méthode, d'une technique ou d'une activité du génie logiciel.
3	X	Pas de compromis entre deux actions ou concepts.
4	X	Dans sa formulation, le principe comporte les termes « *development* » et « *phases* » qui représentent des concepts du génie logiciel.
5	X	La formulation du principe est vérifiable en pratique. Le principe est appliqué si l'équipe utilise une méthode itérative et travaille en phases. L'idée est de sortir au bout de deux *sprints* une version d'un logiciel

Tableau 5.10 Grille d'analyse du principe 3 de *Scrum* selon les critères individuels.

Résultat

Cette proposition n'est pas formulée de façon prescriptive, ainsi elle ne satisfait pas au critère no.1. Par contre, elle satisfait aux quatre autres critères d'évaluation individuels. Pour la retenir comme principe, nous avons tenté de la reformuler sous forme prescriptive. Nous proposons la reformulation mineure suivante : « *Use overlapping development phases.* ». Ainsi, nous pouvons retenir cette proposition comme principe selon les critères d'identification.

Principe 4

Énoncé : *Multilearning*

Traduction : Apprentissage multiple.

Explication

Les équipes doivent rester en contact avec des sources d'information extérieures pour pouvoir répondre rapidement aux changements. Les sources d'informations se présentent à des niveaux multiples (individu, groupe, organisation).

L'apprentissage multiple permet d'améliorer le niveau de l'équipe en permanence. Takeuchi et Nonaka écrivent:

> « *Learning at the individual level takes place in a number of ways. 3M, for example, encourages engineers to devote 15% of their company time to pursuing their "dream".* » (Takeuchi et Nonaka, 1986)

Évaluation

N° du critère	Critère satisfait	Justification
1		Formulation non prescriptive.
2	X	Ne découle pas d'une technologie, d'une méthode, d'une technique ou d'une activité du génie logiciel.
3	X	Pas de compromis entre deux actions ou concepts.
4		Le principe n'inclut pas des concepts reliés au génie logiciel.
5		Formulation non vérifiable en pratique.

Tableau 5.11 Grille d'analyse du principe 4 de *Scrum* selon les critères individuels.

Résultat

Cette proposition ne satisfait pas à trois critères des cinq critères d'évaluation individuelle. Elle n'est pas formulée de façon prescriptive, donc elle ne satisfait pas le critère no.1. Dans sa formulation, cette proposition ne renferme aucun concept relié au génie logiciel, ainsi elle ne satisfait pas au critère no.4. Elle ne peut être vérifiée dans ses conséquences et par suite elle ne satisfait pas le critère no.5. Ainsi, cette proposition n'est pas retenue comme principe selon les critères d'identification.

Principe 5

Énoncé : *Subtle control*

Traduction : Le contrôle subtil.

Explication

Pour appliquer ce principe, Takeuichi et Nonaka (1986) recommandent d'abord de bien choisir les membres de l'équipe, car en sélectionnant les bonnes personnes les chances de réussir un projet sont plus grandes. Ensuite, ils suggèrent de créer une communication constante avec le client. Selon eux, il est important d'établir un système d'évaluation et de

récompense pour les membres de l'équipe et, surtout, guider l'équipe d'une manière l'encourageant à devenir autonome (Takeuchi et Nonaka, 1986).

Dans un projet *Scrum* même si une équipe est considérée comme autonome, elle reste toujours pilotée par des règles qui permettent aux chefs de projet de contrôler le processus de développement.

Évaluation

N° du critère	Critère satisfait	Justification
1		Formulation non prescriptive.
2	X	Ne découle pas d'une technologie, d'une méthode, d'une technique ou d'une activité du génie logiciel.
3	X	Pas de compromis entre deux actions ou concepts.
4		Le principe n'inclut pas des concepts reliés au génie logiciel.
5		Formulation non vérifiable en pratique.

Tableau 5.12 Grille d'analyse du principe 5 de *Scrum* selon les critères individuels.

Résultat

Cette proposition est éliminée de notre liste de principe. Elle n'est pas formulée sous forme prescriptive, donc elle ne satisfait pas le critère no.1. Elle ne contient pas de concepts explicites du génie logiciel, ainsi elle ne satisfait pas au critère no.4. Elle ne peut être vérifiée dans ses conséquences du fait que sa formulation.

Principe 6

Énoncé : *Organizational transfer of learning*

Traduction : Transfert des connaissances dans l'organisation.

Explication

Selon Takeuichi et Nonaka, la meilleure manière pour transférer les connaissances au sein de l'équipe est de mélanger les membres des nouvelles équipes avec des membres des équipes précédentes (Takeuchi et Nonaka, 1986).

Le choix d'une technologie ou d'un produit est aussi considéré comme une source de diffusion des bonnes pratiques. En effet, les apprentissages réalisés par les équipes sont systématiquement diffusés dans l'organisation. La propagation des bonnes pratiques est donc un autre facteur de réussite de l'équipe.

Selon Aubry, une équipe *Scrum* doit couvrir avec l'ensemble de ses membres toutes les activités nécessaires pour obtenir un produit à la fin d'un sprint. Il souligne qu'une équipe qui ne possède pas assez de compétences techniques en ingénierie de logiciel doit faire une mise à niveau. Cela a des impacts sur la gestion du projet et la qualité du produit (Aubry, 2010).

Évaluation

N° du critère	Critère satisfait	Justification
1		Formulation non prescriptive.
2	X	Ne découle pas d'une technologie, d'une méthode, d'une technique ou d'une activité du génie logiciel.
3	X	Pas de compromis entre deux actions ou concepts.
4	X	Le sens donné au principe est relié à la gestion de projet qui représente un concept du génie logiciel selon le guide *SWEBOK*.
5		Formulation non vérifiable en pratique.

Tableau 5.13 Grille d'analyse du principe 6 de *Scrum* selon les critères individuels.

Résultat

Cette proposition est éliminée de notre liste de principe, car elle ne satisfait pas à deux critères des cinq critères d'évaluation individuelle.

5.3.5 Synthèse

Suite à cette phase d'analyse, nous avons retenu un seul principe parmi six « *Overlapping development phases.* ».

- ✓ Le critère no.1, indiquant qu'un principe est une proposition prescriptive, a éliminé six propositions. Ces propositions étaient formulées de façon descriptive. Le troisième principe a été reformulé comme suit « *Use overlapping development phases* » puisqu'il satisfait aux quatre autres critères. Ainsi il a été retenu comme principe.

- ✓ Le critère no.2, à savoir qu'un principe ne découle pas d'une technologie, d'une méthode, d'une technique ou d'une activité du génie logiciel, n'a écarté aucune proposition. Aucune proposition n'est associée à une activité du génie logiciel répertoriée dans la norme *ISO/IEC 12207*.

- ✓ Le critère no.3 n'a écarté aucun principe. Les cinq propositions ne présentent pas de dosage entre deux actions ou concepts.

- ✓ Le critère no.4 a éliminé cinq principes. Ces principes ne renferment aucun concept explicite du génie logiciel.

- ✓ Le critère no.5 a écarté cinq principes. Les formulations de ces principes ne permettent pas de les vérifier et les tester en pratique.

Principe \ Critère	1	2	3	4	5	Résultat
1. *Built-in instability.*	-	+	+	-	-	Rejeté
2. *Self-organizing project teams.*	-	+	+	+	-	Rejeté
3. *Overlapping development phases.*	-	+	+	+	+	Retenu*
4. *Multilearning.*	-	+	+	-	-	Rejeté
5. *Subtle control.*	-	+	+	-	-	Rejeté
6. *Organizational transfer of learning*	-	+	+	+	-	Rejeté

Tableau 5.14 Résultat de l'évaluation des principes de *Scrum* selon les critères individuels.

* avec reformulation.

Dans cette section, nous avons analysé six principes pour n'en conserver qu'un seul énoncé. Le troisième principe a été révisé afin de le reformuler sous forme prescriptive. Ainsi, on retient la reformulation mineure suivante « *Use overlapping development phases* ».

5.4 Dynamic System Development Method (DSDM)

La méthode *Dynamic System Development Method*, ou méthode *DSDM*, est une autre des méthodes dites agiles. *DSDM* a été développée par un consortium de sociétés initialement constitué d'entreprises du Royaume-Uni. Cette méthode est construite sur la méthode de développement *RAD* (développement rapide d'applications). Elle décompose le processus de développement de projet en cinq phases comme l'illustre la figure 5.4 :

1. Cycle de l'étude de faisabilité
2. Cycle d'étude métiers
3. Cycle du modèle fonctionnel itératif
4. Cycle de conception et de développement itératif
5. Cycle d'implémentation

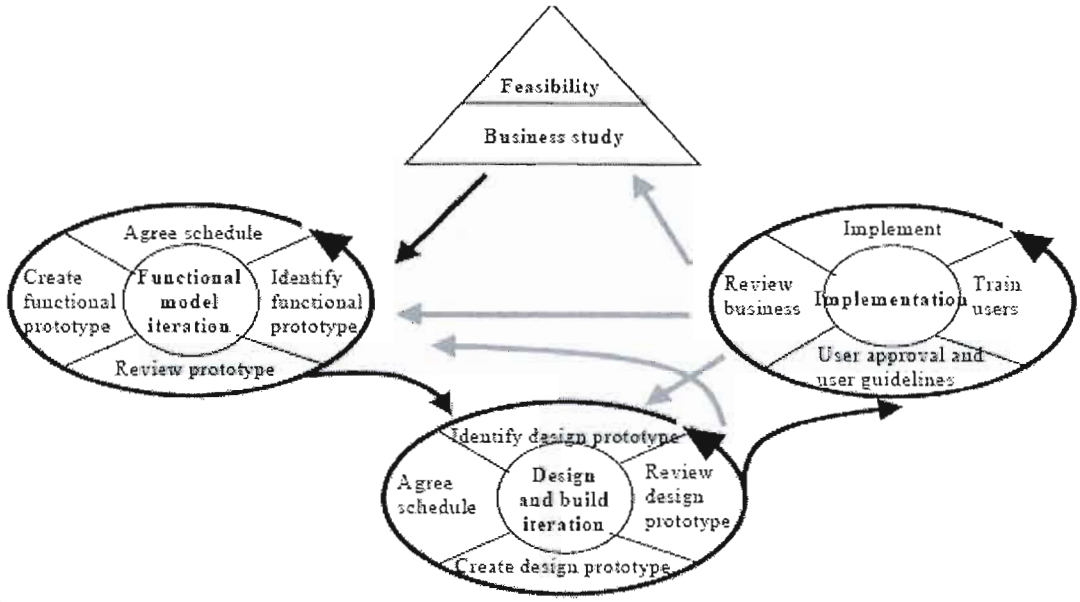

Figure 5.5 Cycle de vie de la méthode *DSDM* (Agile-Alliance, 2002).

L'étude de faisabilité permet de vérifier si la méthode *DSDM* est appropriée au projet. L'étude métier est une courte série d'ateliers de travail ayant comme but la compréhension des domaines métiers concernés par le développement. Elle débouche sur un diagramme de haut niveau représentant l'architecture système et un plan de projet. Le reste du processus est

formé de trois cycles imbriqués : le cycle du modèle fonctionnel itératif produit l'analyse de la documentation et des prototypes; le cycle de conception et de développement itératif produisent l'ingénierie du système en vue d'une utilisation opérationnelle; le cycle d'implémentation produit le déploiement pour l'utilisation opérationnelle.

La méthode est basée sur un ensemble de principes incluant une participation forte des utilisateurs, une priorisation des fonctionnalités, des livraisons fréquentes, des équipes autonomes, une réduction de la documentation et une intégration des tests tout au long du processus.

Selon Fowler, la méthode *DSDM* est remarquable pour son infrastructure, digne d'une méthode plus mature, tout en suivant les principes des méthodologies agiles (Fowler, 2005).

5.4.1 Principes

La méthode *DSDM* est assez peu documentée. À partir des quelques livres et des documents accessibles, nous avons pu extraire un certain nombre d'informations, dont celles nécessaires à l'explication des principes.

DSDM s'appuie sur neuf principes mentionnés dans le livre référence de Jennifer Stapleton « *DSDM: Dynamic Systems Development Method* » (Stapleton, 1997). En consultant ces principes, nous avons réalisé qu'ils sont proches de ceux du manifeste agile. Selon Highsmith, ces principes sont assez proches des principes agiles du manifeste qu'une explication n'est pas nécessaire sauf pour le sixième principe (Highsmith, 2002).

Dans ce qui suit, nous allons, comme dans les autres méthodes agiles, procéder à l'étude des principes de *DSDM* qui sont les suivants :

1. *Active user involvement is imperative.*
2. *DSDM teams must be empowered to make decisions.*
3. *The focus is on frequent delivery of products.*
4. *Fitness for business purpose is the essential criterion for acceptance of deliverables.*
5. *Iterative and incremental development is necessary to converge on an accurate business solution.*
6. *All changes during development are reversible.*
7. *Requirements are baselined at a high level.*
8. *Testing is integrated throughout the life cycle.*
9. *A collaborative and cooperative approach between all stakeholders is essential.*

Principe 1

Énoncé : *Active user involvement is imperative.*

Explication

La participation active des utilisateurs est un des principes clés des méthodes agiles. La présence de ce principe parmi les principes de *DSDM* ne fait que renforcer cette idée. L'équipe doit travailler ensemble à établir et préciser clairement les exigences, définir les priorités, identifier les tâches nécessaires pour satisfaire les besoins et estimer les charges.

Il est difficile d'avoir des utilisateurs directement impliqués dans les projets de développement, mais leur présence et leur implication ne peuvent qu'être bénéfiques pour le projet. Cette implication fait en sorte que les exigences et les besoins des clients sont communiqués de façon claire à l'équipe de développement. Dans un projet *DSDM* comme dans toutes les autres méthodes agiles, les clients sont considérés comme des membres à part entière de l'équipe de projet.

Évaluation

N° du critère	Critère satisfait	Justification
1		Formulation non prescriptive.
2	X	Ne découle pas d'une technologie, d'une méthode, d'une technique ou d'une activité du génie logiciel.
3	X	Pas de compromis entre deux actions ou concepts.
4	X	L'utilisateur est un concept du génie logiciel selon le guide *SWEBOK*.
5	X	Ce principe est vérifiable en pratique par le fait que les clients ne sont pas considérés comme des simples fournisseurs d'informations, mais bien comme des acteurs à part entière de l'équipe.

Tableau 5.15 Grille d'analyse du principe 1 de *DSDM* selon les critères individuels.

Résultat

Cette proposition ne satisfait pas au critère no.1 étant donné qu'elle n'est pas formulée sous forme prescriptive. Afin de retenir ce principe parmi la liste des principes, nous l'avons reformulé de la façon suivante : « *Imperatively involve active user.* »

Principe 2

Énoncé : *DSDM teams must be empowered to make decisions.*

Explication

Dans un projet, une équipe est formée de développeurs et de clients. Cette équipe doit être autorisée à prendre des décisions concernant la modification des besoins et des fonctionnalités à développer. L'équipe doit avoir un pouvoir de prise de décision concernant l'évolution des besoins ce qui lui permet de se sentir impliquée dans le projet.

Évaluation

N° du critère	Critère satisfait	Justification
1	X	Formulation prescriptive.
2	X	Ne découle pas d'une technologie, d'une méthode, d'une technique ou d'une activité du génie logiciel.
3	X	Pas de compromis entre deux actions ou concepts.
4	X	« *Teams* » (*users and developers*) est un concept du génie logiciel selon le guide *SWEBOK*.
5	X	Formulation vérifiable en pratique par l'autorité donnée à l'équipe pour prendre des décisions liées aux exigences, le choix des fonctionnalités à développer dans un incrément donné, le classement des priorités, etc.

Tableau 5.16 Grille d'analyse du principe 2 de *DSDM* selon les critères individuels.

Résultat

Cette proposition est retenue comme principe, puisqu'elle répond aux cinq critères d'évaluation individuels.

Principe 3

Énoncé : *The focus is on frequent delivery of products.*

Explication

L'équipe doit être en mesure de livrer le plus souvent possible afin de permettre un feed-back rapide de la part du client. Avoir des délais de livraison courts permet de mieux définir quelles sont les activités qui permettront d'atteindre les résultats prioritaires. Les livraisons fréquentes permettent aussi de s'assurer que les erreurs sont détectées rapidement. Ceci s'applique au code aussi bien qu'aux documents.

Selon Aubry, il existe trois usages d'une version produite (Aubry, 2010) :

- ✓ La version est produite pour chercher à minimiser les risques liés à la technologie et à la capacité de l'équipe à intégrer différents composants. Elle n'est pas livrée au client. Cela est fréquent au début du projet.
- ✓ La version est produite pour être évaluée par des utilisateurs sélectionnés. Ces utilisateurs pourront évaluer la facilité d'utilisation des fonctionnalités et en proposer de nouvelle.
- ✓ La version est produite pour être mise en production ou en exploitation par ses utilisateurs finaux. C'est ce qu'il faut viser puisque chaque nouvelle version apporte de la valeur.

Évaluation

N° du critère	Critère satisfait	Justification
1		Formulation non prescriptive.
2	X	Ne découle pas d'une technologie, d'une méthode, d'une technique ou d'une activité du génie logiciel.
3	X	Pas de compromis entre deux actions ou concepts.
4	X	Livrer le produit aux clients est un concept du génie logiciel selon le guide *SWEBOK*. Le produit fait référence au logiciel.
5	X	Dans un projet *DSDM,* ce principe est vérifiable par le biais des livraisons fréquentes de logiciels.

Tableau 5.17 Grille d'analyse du principe 3 de *DSDM* selon les critères individuels.

Résultat

Cette proposition ne satisfait pas au critère no.1 étant donné qu'elle n'est pas formulée sous forme prescriptive. Afin de retenir ce principe dans la liste des principes, nous l'avons reformulé de la façon suivante : « *Focus on frequent delivery of products.* »

Principe 4

Énoncé : *Fitness for business purpose is the essential criterion for acceptance of deliverables.*

Explication

L'équipe de développement doit pouvoir livrer un produit en adéquation avec les besoins métiers du client. Dans un projet *DSDM,* l'élaboration d'un système complet n'est pas une priorité : la satisfaction des besoins métiers est le critère le plus essentiel. DSDM suggère de satisfaire aux besoins d'affaires d'abord. L'adaptation au changement du cycle de développement permet de lancer sur le marché un produit qui sera entièrement adopté par le public cible.

> « *If you can innovate better and faster–you create change for your competitors. If you can respond quickly to competitive initiatives, new technology and customers' requirements–you create change for your competitors. If you are slower, less innovative, less responsive–you are doomed to survival strategies in a sea of chaos imposed by others.* » (Highsmith, 2002)

Highsmith, l'un des promoteurs du développement de logiciel agile, souligne que vu le contexte dans lequel évoluent les entreprises, elles doivent faire face à de nombreux changements technologiques et s'y préparer. Dans ces conditions, le changement est considéré comme un avantage concurrentiel pour le client, puisque le critère principal pour l'acceptation d'un livrable est de fournir un système qui répond aux besoins.

Évaluation

N° du critère	Critère satisfait	Justification
1		Formulation non prescriptive.
2	X	Ne découle pas d'une technologie, d'une méthode, d'une technique ou d'une activité du génie logiciel.
3	X	Pas de compromis entre deux actions ou concepts.
4	X	« *Fitness for business purpose* » est un concept du génie logiciel selon le guide *SWEBOK*. Construire un logiciel « approprié à l'usage » est un attribut important pour obtenir l'acceptation du client.
5	X	La vérification du principe est faite en comparant si l'application est en adéquation avec le besoin métier du client.

Tableau 5.18 Grille d'analyse du principe 4 de *DSDM* selon les critères individuels.

Résultat

Cette proposition ne satisfait pas au critère no.1 étant donné qu'elle est formulée sous forme descriptive. Par contre, elle satisferait aux quatre autres critères, une reformulation mineure pourra être appliquée afin de la rendre sous forme prescriptive. Nous n'avons pas

trouvé de reformulation mineure appropriée à ce principe. Ainsi, nous ne le considérerons pas comme principe.

Principe 5

Énoncé : *Iterative and incremental development is necessary to converge on an accurate business solution.*

Explication

La nature itérative et incrémentale du processus de développement d'un projet *DSDM* est basée sur le feed-back des clients, tout comme dans les autres méthodes agiles.

Évaluation

N° du critère	Critère satisfait	Justification
1		Formulation non prescriptive.
2	X	Ne découle pas d'une technologie, d'une méthode, d'une technique ou d'une activité du génie logiciel.
3	X	Pas de compromis entre deux actions ou concepts.
4	X	Le modèle itératif et incrémental est un concept du génie logiciel relié aux modèles de cycle de vie de logiciel selon le guide *SWEBOK*. Ces processus de cycle de vie de logiciel permettent d'augmenter la qualité du
5	X	Formulation vérifiable en pratique par le fait que chaque itération ajoute de nouvelles fonctionnalités jusqu'à ce que les exigences d'affaires soient accomplies.

Tableau 5.19 Grille d'analyse du principe 5 de *DSDM* selon les critères individuels.

Résultat

Cette proposition ne satisfait pas au critère no.1 étant donné qu'elle n'est pas formulée sous forme prescriptive. Nous l'avons reformulé comme suit : « *Converge to an accurate business solution through iterative and incremental development.* ». Ainsi, elle est retenue comme principe selon les critères d'identification.

Principe 6

Énoncé : *All changes during development are reversible.*

Explication

Toute modification effectuée durant le développement doit être réversible. Stapleton explique que répondre au changement exige que les configurations de système changent pendant le développement de n'importe quel incrément dû aux changement de priorités dans les exigences (Stapleton, 1997).

Plusieurs outils logiciels soutiennent ce principe en maintenant une configuration dynamique des projets (gestion de configuration). Ainsi, à un point quelconque, des changements faits dans les documents ou dans le code peuvent être renversés ou annulés. L'équipe peut craindre l'action de renverser les changements en supprimant les travaux précédents, mais puisque *DSDM*[11] procède par petits incréments, la perte de travail est limitée. Grâce à ce principe, l'équipe n'a plus peur de faire des erreurs, car ils peuvent être rapidement défaits.

---

[11] Le *DSDM Manual* fournit des conseils sur la façon dont les changements peuvent être réversibles. `http://www.dsdm.org/version4`

Évaluation

N° du critère	Critère satisfait	Justification
1		Formulation non prescriptive.
2	X	Ne découle pas d'une technologie, d'une méthode, d'une technique ou d'une activité du génie logiciel.
3	X	Pas de compromis entre deux actions ou concepts.
4	X	Dans sa formulation, le principe comprend le terme « *changes* » qui est un concept du génie logiciel (selon le guide *SWEBOK*).
5	X	Formulation vérifiable en pratique par le fait que le code ou les documents peuvent être mis à un état antérieur.

Tableau 5.20 Grille d'analyse du principe 6 de *DSDM* selon les critères individuels.

Résultat

Cette proposition ne satisfait pas au critère no.1 étant donné qu'elle n'est pas formulée sous forme prescriptive. Nous proposons la reformulation suivante : « *Ensure that all changes during development are reversible.* ». Cette reformulation guide l'action et reprend les termes de la proposition. Ainsi, elle est retenue parmi la liste des principes.

Principe 7

Énoncé : *Requirements are baselined at a high level.*

Explication

Selon Stapleton, l'application de ce principe se concrétise par une définition globale des exigences. Au début du projet, les exigences sont définies à l'aide d'un schéma fixant les grandes lignes du projet. Cette définition sert à déterminer les objectifs et le périmètre du système, permettant ainsi une étude approfondie des besoins à un stade ultérieur. D'autres modifications peuvent être effectuées en cours du développement mais le périmètre ne doit pas être modifié sauf de façon limitée (Stapleton, 1997).

Dans un projet *DSDM*, toutes les exigences identifiées doivent être vues à un niveau élevé et dans un format visuel, par exemple, en utilisant des séquences de captures d'écran (*story-board*), des visuels, des diagrammes de séquence. L'identification des exigences se fait dans le cadre d'une activité conjointe de l'équipe qui permet à tous les membres de l'équipe de comprendre les exigences et de comprendre ce qui est nécessaire.

Évaluation

N° du critère	Critère satisfait	Justification
1		Formulation non prescriptive.
2	X	Ne découle pas d'une technologie, d'une méthode, d'une technique ou d'une activité du génie logiciel.
3	X	Pas de compromis entre deux actions ou concepts.
4	X	Dans sa formulation, le principe comprend le terme « *requirements* » qui est un concept du génie logiciel selon le guide *SWEBOK*.
5	X	Formulation vérifiable en pratique en établissant les exigences pour chaque élément à développer.

Tableau 5.21 Grille d'analyse du principe 7 de *DSDM* selon les critères individuels.

Résultat

Cette proposition ne satisfait pas au critère no.1 étant donné qu'elle n'est pas formulée sous forme prescriptive. Afin de retenir ce principe parmi la liste des principes, nous l'avons reformulé de la façon suivante : « *Baseline requirements at a high level.* »

Principe 8

Énoncé : *Testing is integrated throughout the life cycle.*

Explication

Les tests sont présents durant toutes les phases développement pour garantir le bon fonctionnement de l'application à chaque itération. Dans un projet DSDM, les tests ne sont pas traités comme une activité séparée. Pendant que le système est développé incrémentalement, il est également testé par les développeurs et les utilisateurs pour s'assurer que le développement avance non seulement dans la bonne direction côté clients, mais du côté technique aussi. Les développeurs s'engagent dans l'écriture de tests unitaires automatiques pour valider leur code.

Selon Aubry, il faut donc procéder de la façon suivante : le développeur commence d'abord par écrire les tests unitaires d'un composant ce qui lui permet de réfléchir au comportement attendu de ce composant. Ensuite, il écrit le code pour que les tests passent avec succès. Aubry précise que le fait d'écrire le test avant permet de rester simple au niveau du code. Ainsi, le développeur peut continuer à rajouter de nouveaux tests puis le code minimal pour qu'ils passent (Aubry, 2010).

Évaluation

N° du critère	Critère satisfait	Justification
1		Formulation non prescriptive.
2	X	Ne découle pas d'une technologie, d'une méthode, d'une technique ou d'une activité du génie logiciel.
3	X	Pas de compromis entre deux actions ou concepts.
4	X	Dans sa formulation, le principe comprend le terme « *testing* » et « *life cycle* » qui sont des concepts du génie logiciel (selon le guide *SWEBOK*).
5	X	Formulation vérifiable en pratique par le biais des tests.

Tableau 5.22 Grille d'analyse du principe 8 de *DSDM* selon les critères individuels.

Résultat

Cette proposition ne satisfait pas au critère no.1 étant donné qu'elle n'est pas formulée sous forme prescriptive. Nous procédons à sa reformulation sous la forme suivante : « *Integrate testing throughout the life cycle.* ». Ainsi, elle est retenue parmi la liste des principes.

Principe 9

Énoncé : *A collaborative and cooperative approach between all stakeholders is essential.*

Explication

La méthode *DSDM* se base sur une étroite collaboration de tous les membres de l'équipe et des intervenants. Les intervenants au projet doivent montrer un véritable esprit de collaboration.

Dans les méthodes traditionnelles de gestion de projet, c'est le chef de projet qui décide et gère le projet. Les membres de l'équipe ne sont que des exécutants. En développement agile, on favorise plutôt l'esprit d'équipe.

Stapleton (1997) affirme: « *Developers cannot divine what is needed without support from the end users.* » Il souligne aussi que les utilisateurs font partie de l'équipe. Ils ont un grand impact sur l'avancement de l'équipe. L'équipe demande des précisions à l'utilisateur lors de l'inspection du produit. Cette collaboration permettra de maximiser la valeur ajoutée au produit.

Évaluation

N° du critère	Critère satisfait	Justification
1		Formulation non prescriptive.
2	X	Ne découle pas d'une technologie, d'une méthode, d'une technique ou d'une activité du génie logiciel.
3	X	Pas de compromis entre deux actions ou concepts.
4	X	Dans sa formulation, le principe comprend le terme « *stakeholders* » qui est un concept du génie logiciel selon le guide *SWEBOK*.
5	X	Ce principe est vérifié par la pratique par le biais de la gestion de l'équipe projet.

Tableau 5.23 Grille d'analyse du principe 9 de *DSDM* selon les critères individuels.

Résultat

Cette proposition ne satisfait pas au critère no.1 étant donné qu'elle n'est pas formulée sous forme prescriptive. La reformulation mineure appropriée à ce principe est la suivante : « *All stakeholders must maintain a collaborative and cooperative approach between each other.* ». Ainsi, nous pouvons la considérer comme principe selon les critères d'identification.

5.4.2 Synthèse

L'analyse des principes de la méthode *DSDM* nous conduit à retenir six principes parmi neuf.

- ✓ Le critère no.1, indiquant qu'un principe est une proposition prescriptive, a éliminé sept propositions. Ces principes satisfont aux quatre autres critères d'évaluations, mais ils sont formulés de façon descriptive. Par la suite, une reformulation qui guide l'action peut être appliquée à chacun d'entre eux. Pour qu'une reformulation soit acceptée, elle devrait être mineure. Dans le cas du quatrième principe, nous n'avons pas trouvé une reformulation mineure appropriée à ce principe. Il doit être entièrement réécrit. Ainsi il ne peut pas être retenu comme principe selon les critères d'identification. Par contre, le reste des principes étaient des candidats pour des reformulations mineures.

- ✓ Le critère no.2, à savoir qu'un principe ne découle pas d'une technologie, d'une méthode, d'une technique ou d'une activité du génie logiciel, n'a écarté aucune proposition. Aucune proposition n'est associée à une activité du génie logiciel répertoriée dans la norme *ISO/IEC 12207*.

- ✓ Le critère no.3 n'a écarté aucun principe. Les cinq propositions ne présentent pas de dosage entre deux actions ou concepts.

- ✓ Le critère no.4 n'a éliminé aucun principe. Tous les principes étudiés renferment des concepts explicites du génie logiciel.

- ✓ Le critère no.5 n'a écarté aucun principe. Les formulations de ces principes permettent de les vérifier et les tester en pratique.

La liste des principes reformulés est la suivante :

1. « *Active user involvement is imperative.* ». Cette proposition est reformulée comme suit : « *Imperatively involve active user.* ».

2. « *The focus is on frequent delivery of products.* ». La reformulation mineure qui reprend les termes de la proposition est la suivante : « *Focus on frequent delivery of products.* ».

3. « *Iterative and incremental development is necessary to converge on an accurate business solution* ». La reformulation de la proposition sous forme prescriptive est la suivante : « *Converge to an accurate business solution through iterative and incremental development.* ».

4. « *All changes during development are reversible.* ». Cette proposition est reformulé comme suit : « *Ensure that all changes during development are reversible.* ».

5. « *Requirements are baselined at a high level.* ». Cette proposition est reformulée dans ces termes : « *Baseline requirements at a high level* ».

6. « *Testing is integrated throughout the life cycle.* ». Nous avons reformulé cette proposition comme suit : « *Integrate testing throughout the life cycle* ».

7. « *A collaborative and cooperative approach between all stakeholders is essential.* ». La reformulation mineure *est : « All stakeholders must maintain a collaborative and cooperative approach between each other.* ».

Les principes de la méthode *DSDM* retenus sont les suivants :

- ✓ **Principe 1 :** *Imperatively involve active user.*
- ✓ **Principe 2 :** *DSDM teams must be empowered to make decisions.*
- ✓ **Principe 3 :** *Focus on frequent delivery of products.*

- ✓ **Principe 4 :** *Converge to an accurate business solution through iterative and incremental development.*
- ✓ **Principe 5 :** *Ensure that all changes during development are reversible.*
- ✓ **Principe 6 :** *Baseline requirements at a high level.*
- ✓ **Principe 7 :** *Integrate testing throughout the life cycle.*
- ✓ **Principe 8 :** *All stakeholders must maintain a collaborative and cooperative approach between each other.*

Critère / Principe	1	2	3	4	5	Résultat
1. *Active user involvement is imperative.*	-	+	+	+	+	Retenu*
2. *DSDM teams must be empowered to make decisions.*	+	+	+	+	+	Retenu
3. *The focus is on frequent delivery of products.*	-	+	+	+	+	Retenu*
4. *Fitness for business purpose is the essential criterion for acceptance of deliverables.*	-	+	+	+	+	Rejeté
5. *Iterative and incremental development is necessary to converge on an accurate business solution.*	-	+	+	+	+	Retenu*
6. *All changes during development are reversible.*	-	+	+	+	+	Retenu*
7. *Requirements are baselined at a high level.*	-	+	+	+	+	Retenu*
8. *Testing is integrated throughout the life cycle.*	-	+	+	+	+	Retenu*
9. *A collaborative and cooperative approach between all stakeholders is essential.*	-	+	+	+	+	Retenu*

Tableau 5.24 Résultat de l'évaluation des principes de *DSDM* selon les critères individuels.

* avec reformulation.

CONCLUSION

« *Une conclusion, c'est quand vous en avez assez de penser.* »

Herbert Albert Fisher

Dans le présent chapitre, nous présentons ce qui a été réalisé, ce qui a été appris, et ce qui pourrait être fait dans des travaux futurs.

Travaux accomplis

Dans les deux chapitres précédents, nous avons analysé 32 supposés principes agiles pour finalement ne retenir que 19 principes (tableau suivant) :

- ✓ sept principes du manifeste agile sont retenus parmi douze;
- ✓ trois principes de l'*eXtreme Programming* sont retenus parmi cinq;
- ✓ un seul principe de *Scrum* est retenu parmi six;
- ✓ huit principes de *DSDM* sont retenus parmi neuf.

Ces principes satisfont aux critères d'identification individuels. Parmi les principes retenus, huit principes ont dû être reformulés légèrement sous une forme prescriptive.

Manifeste agile
❖ *Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.*
❖ *Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.*
❖ *Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.*
❖ *Business people and developers must work together daily throughout the project.*
❖ *Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.*
❖ *Measure the progress primarily through working software.*
❖ *Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.*
eXtreme Programming
❖ *Provide rapid feedback to its customers and learn from that feedback.*
❖ *Improve the software through small, incremental changes.*
❖ *Insist that the software consistently exhibits the highest level of quality workmanship.*
Scrum
❖ *Use overlapping development phases.*
Dynamic Software Development Method
❖ *Imperatively involve active user.*
❖ *DSDM teams must be empowered to make decisions.*
❖ *Focus on frequent delivery of products.*
❖ *Converge to an accurate business solution through iterative and incremental development.*
❖ *Ensure that all changes during development are reversible.*
❖ *Baseline requirements at a high level.*
❖ *Integrate testing throughout the life cycle.*
❖ *All stakeholders must maintain a collaborative and cooperative approach between each other.*

Tableau Principes retenus.

L'objectif de notre recherche était, dans un premier temps, d'identifier les principes introduits dans la littérature sur les méthodes agiles et, dans un deuxième temps, de vérifier si ces principes satisfont aux critères d'identification définis par Normand Séguin (Séguin, 2006). Pour ce faire, il a fallu tout d'abord sélectionner les sources des principes que nous voulions analyser. Ensuite, il nous a fallu comprendre la méthodologie de recherche présentée par Séguin (2006) et en extraire les éléments adaptés au contexte de notre recherche. Notre choix s'est porté sur le manifeste agile et sur trois méthodes agiles. Le manifeste agile est la base de toutes les méthodes agiles, par conséquent l'analyse de ses douze principes (chapitre 4) constitue, dans une première étape, une bonne assise pour entreprendre notre étude des principes agiles. L'analyse des principes de trois méthodes agiles reconnues (chapitre 5) vient, dans une deuxième étape, compléter notre évaluation des principes agiles.

La définition de plusieurs termes s'est avérée importante : concept, principe, valeur, pratique et méthode. Ces définitions forment un fondement à notre cadre conceptuel (chapitre 1). Elles soutiennent l'analyse des principes. La norme *ISO/IEC 12207* et le guide *SWEBOK* sont aussi des composants fondamentaux de ce cadre. La définition du terme principe nous a servi à vérifier, notamment, si un principe agile est formulé sous forme prescriptive : un principe doit spécifier une action à faire. La norme *ISO/IEC 12207* regroupe l'ensemble des activités du génie logiciel. Elle nous a permis de nous assurer qu'un principe agile n'est pas simplement une activité du génie logiciel. Le guide *SWEBOK* identifie et décrit les principaux concepts du génie logiciel. Il nous a été utile pour déterminer si un principe agile contient un ou plusieurs concepts du génie logiciel.

La revue de littérature (chapitre 2) a révélé que les origines du développement agile de logiciel remontent au-delà de l'apparition du manifeste agile. L'approche agile, au sens du manifeste agile, est apparue au début des années 2000. Mais les premières méthodes et techniques agiles remontent au milieu des années 1980 avec l'introduction du modèle en spirale (Boehm, 1988).

À notre connaissance, notre recherche est la seule à avoir analysé les principes du génie logiciel au niveau du développement agile. Cette recherche a présenté le développement agile d'un côté purement théorique. Les résultats obtenus s'appuient sur des critères d'identification introduits et justifiés dans le travail de Normand Séguin portant sur les principes du génie logiciel. De plus, le travail de recherche décrit dans ce mémoire soutient l'hypothèse émise par Séguin (2006): dans le cas où une liste de principes serait publiée, elle pourra être facilement traitée à l'aide de la méthode de recherche qu'il a proposée. En effet, la définition de ce qu'est un principe et la spécification de critères d'identifications permettent de poursuivre la recherche sur les principes de développement de logiciel.

Travaux futurs

À l'issue des travaux menés pour ce mémoire, un certain nombre de pistes de recherche mériteraient, à notre sens, d'être explorées :

- ✓ Classer les 19 principes retenus selon trois catégories proposées par Séguin (2006) : individu, produit et processus. Ensuite, appliquer les deux critères d'ensembles, présentés brièvement à la section 3.4, aux principes retenus.

- ✓ Vérifier le degré de couverture de la discipline de développement agile par les principes agiles retenus.

- ✓ Analyser les principes de l'*Agile Modeling* (*AM*) et du l'*Agile Project Management*. Le présent travail n'a pas tenu compte de ces principes.

- ✓ Analyser les principes d'autres méthodes agiles comme *PUMA* et *Crystal*.

BIBLIOGRAPHIE

Abran, Alain, Pierre Bourque, Robert Dupuis et James W. Moore. 2004. *Guide to the Software Engineering Body of Knowledge — SWEBOK*: IEEE Computer Society Press. En ligne. <http://www.swebok.org>.

Agile-Alliance. 2002. «Agile Alliance». En ligne. <http://www.agilealliance.org>.

Agile-Québec. 2005. «Agile Québec». En ligne. <http://www.agilequebec.ca>.

Ambler, Scott W. 2002. «Agile Modeling». En ligne. <http://www.agilemodeling.com>.

--------. 2005. «Agile Requirements Change Management». En ligne. <http://www.agilemodeling.com/essays/changeManagement.htm>.

--------. 2008. «Answering the "Where is the Proof That Agile Methods Work" Question». En ligne. <http://www.agilemodeling.com/essays/proof.htm>.

Aubry, Claude. 2010. *Scrum : le guide pratique de la méthode agile la plus populaire* : Dunod.

Beck, Kent. 1999. *Extreme Programming Explained: Embrace Change*: Addison-Wesley Professional.

Beck, Kent, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland et Dave Thomas. 2001. «Manifesto for Agile Software Development». En ligne. <http://agilemanifesto.org>.

Boehm, Barry W. 1988. «A spiral model of software development and enhancement». *Computer*, vol. 21, no 5, p. 61-72.

Cockburn, Alistair. 2001. *Agile Software Development*: Addison-Wesley Professional.

Cohn, Mike. 2004. *User Stories Applied: For Agile Software Development*: Addison Wesley Longman Publishing Co., Inc.

Fowler, Martin. 2005. «The New Methodology». En ligne. <http://martinfowler.com>.

Grand-dictionnaire-terminologique. 1997. Semantix. En ligne. <http://www.granddictionnaire.com/btml/fra/r_motclef/index1024_1.asp>.

Highsmith, Jim. 2001. «History: The Agile Manifesto». En ligne. <http://www.agilemanifesto.org/history.html>.

--------. 2002. *Agile software development ecosystems*: Addison-Wesley Longman Publishing Co., Inc.

Jeffries, Ronald.E. 2006. «What is Extreme Programming?». En ligne. <http://www.xprogramming.com/xpmag/whatisxp.htm>.

Kniberg, Henrik. 2007. *Scrum and XP from the Trenches: Enterprise Software Development*: Lulu.com.

Larman, Craig. 2003. *Agile and Iterative Development: A Manager's Guide*: Pearson Education.

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

Leffingwell, Dean. 2007. *Scaling Software Agility: Best Practices for Large Enterprises (The Agile Software Development Series)*: Addison-Wesley Professional.

Martin, James. 1991. *Rapid Application Development*: Macmillan Pub. Co. ; Collier Macmillan Canada ; Maxwell Macmillan International.

Martin, Robert Cecil. 2003. *Agile Software Development: Principles, Patterns, and Practices*: Prentice Hall PTR.

Merriam-Webster, Inc. 1990. *Webster's basic English dictionary*: Merriam-Webster.

Nagel, R, et N Devia. 1999. «Agile Manufacturing». In *the 5th International Conference on Manufacturing Technology* (Beijing, China), sous la dir. de. Beijing, China.

Peter, Measey. 2007. «Agile (DSDM Atern) at Rolls-Royce». En ligne. <http://www.projchallenge.com/presentations_archive.cfm#20079>.

Power, Gus. 2006. «Values, Practices & Principles». En ligne. <http://www.energizedwork.com/weblog/2006/12/values-practices-principles>.

Schwaber, K., et J. Sutherland. 1995. «The Scrum Papers: Nut, Bolts, and Origins of an Agile Framework». In *OOPSLA' 95*, p. 219.

Schwaber, Ken. 1995. «Scrum development process ». In *Annual ACM Conference on Object Oriented Programming Systems, Languages, and Applications (OOPSLA)*, p. 117-134.

--------. 2004. *Agile Project Management with Scrum* 1 edition ; Microsoft Press.

Schwaber, Ken, et Mike Beedle. 2001. *Agile Software Development with Scrum*: Prentice Hall PTR.

Séguin, Normand. 2006. «Inventaire, analyse et consolidation des principes fondamentaux du génie logiciel». École de Technologie Supérieure (Canada).

Shore, James, et Shane Warden. 2007. *The art of agile development*: O'Reilly.

Stapleton, Jennifer. 1997. *DSDM: The Method in Practice*: Addison-Wesley Longman Publishing Co., Inc.

Swanson, Brad. 2009. «Comparing Open Source Agile Project Management Tools». En ligne. <http://olex.openlogic.com/wazi/2009/comparing-open-source-agile-project-management-tools/>.

Takeuchi, Hirotaka, et Ikujiro Nonaka. 1986. «The New New Product Development Game». *Harvard Business Review Article*, p. 10.

Vickoff, Jean-Pierre. 1999. *Réingénierie du développement d'application, Méthodes, processus, techniques et pratiques de conduite de projet sous contraintes:* Gartner Group.

Wells, Don. 2001. «Extreme Programming: A gentle introduction». En ligne.
<http://www.extremeprogramming.org>.