MASTER EN SCIENCES INFORMATIQUES

Elaboration d'une démarche de qualité adaptée aux exigences des méthodologies Open source

Renaud, Steven

Award date:
2006

Facultés Universitaires Notre-Dame de la Paix, Namur
Institut d'informatique
Année académique 2005-2006

Elaboration d'une démarche de qualité adaptée aux exigences des méthodologies **Open Source**

Steven Renaud
<steven@run.be>

Mémoire présenté en vue de l'obtention du grade de licencié en informatique

Résumé

L'objet de cette étude est l'élaboration d'une démarche de qualité dans le cadre du développement de logiciels Open Source. Cette étude est organisée en trois parties :

1. Formalisation d'une méthodologie de développement Open Source de type "Agile"
2. Recherche de pratiques de qualité
3. Mise en conformité des pratiques de qualité grâce au modèle CMMi

Mots clés : Open Source, Agile, Qualité Logicielle, CMMi

Abstract

The purpose of this report is the elaboration of a quality approach in Open Source software development. This study is organized in three parts :

1. Formalization of an Open Source software development methodology considered as Agile
2. Exploration of quality-oriented practices
3. Standardization of the quality-oriented practices with the CMMi model

Keywords : Open Source, Agile, Software Quality, CMMi

Je tiens à remercier mon promoteur, Monsieur Naji Habra, pour avoir cru en moi au long de ces deux années de recherches et pour ses nombreux conseils avisés.
Je tiens également à remercier ma famille pour m'avoir soutenu au long de ces années de travail et pour avoir réussi à trouver les arguments pour me convaincre de persévérer lorsque je perdais confiance.
Et enfin, à toutes les personnes qui, de près ou de loin, m'ont aidé à mener à bien ce mémoire, j'adresse mes plus vifs remerciements.

Table des matières

Table des figures

Introduction

Déjà souvent dénoncée pour ses manquements dans la gestion des coûts, des délais et de la fiabilité, l'industrie du logiciel doit de plus, actuellement, faire face à une augmentation de la complexité logicielle à laquelle de nombreux experts prédisent l'application de la très remarquable loi de Moore[1].

Les méthodologies "prédictives" ont été conçues pour résoudre les problèmes récurrents de maîtrise de du cycle de vie de développement de logiciel. Elles tentent de réduire l'incertitude en prédisant l'entièreté du projet le plus tôt possible. Elles ont cependant montré leurs limites. Métaphoriquement, ces méthodologies prédictives de développement peuvent être représentées comme un tir à l'arbalète vers une cible : malgré les qualités du tireur et la précision du geste, le moindre coup de vent ou le déplacement de la cible durant le tir auront de fortes chances d'engendrer l'échec du lancer. La prédiction à long terme ne permet pas de gérer les aléas de la complexité logicielle.

C'est pour pallier aux risques d'échecs dus à l'extrême rigidité des méthodologies prédictives qu'apparaisent les méthodologies de développement de logiciel adaptatives. Celles-ci ont comme particularité fondamentale la capacité d'adaptation du projet tout au long de son cycle de vie. Ces méthodologies peuvent s'apparenter à un trajet en voiture avec une bonne carte entre les mains. Après un rapide plan de route, le conducteur peut prendre le volant. Tout au long de son trajet, il sera capable de s'adapter à l'apparition inopinée d'une déviation ou d'un changement de destination grâce au ré-ajustement du plan de route initial. C'est dans ce contexte adaptatif que sont apparues les méthodologies dites "Agile" et les logiciels Open Source.

L'Open Source est avant tout un type de logiciel qui octroie des droits et des devoirs aux différents acteurs du logiciel par le biais d'une licence. Néanmoins, le succès de nombreux projets mettant en avant des logiciels Open Source a mis en évidence des bribes d'une méthodologie de développement de type Open Source qui, de par sa nature, fait partie intégrante des méthodologies dites adaptatives.

Afin de faire face à l'augmentation de la complexité de la construction logicielle, de nombreuses organisations utilisant des méthodologies prédictives ont fait le choix de la certification de qualité. La certification la plus utilisée dans le monde est CMMi. Elle est le fruit du rapprochement de différents standards.

C'est précisément le modèle CMMi que cette étude va tenter d'utiliser pour valider un certain nombre de pratiques de qualité issues des méthodologies de développement Open Source.

[1] Voir l'article du "Courrier International" [Tri03]

Ma contribution personnelle à cette étude se décline en trois axes distincts :

- *la formalisation d'une méthodologie de développement Open Source* sur base de la littérature des théories agilistes et Open Source.
- *la recherche des pratiques de qualité* sur base de la littérature agiliste, d'une analyse empirique de différents projets Open Source et de mon expérience professionnelle.
- *la validation des pratiques de qualité* en se basant sur le modèle CMMi.

Ces trois points correspondent aux trois parties de cette étude.

La première partie est une recherche théorique sur les méthodologies adaptatives ayant pour but de mettre en évidence une méthodologie de développement Open Source. Le *chapitre 1* définit les méthodologies Agile et en particulier sa représentation la plus populaire, l'Extreme Programming. Le *chapitre 2* fait le point concernant les concepts définissant l'Open Source et formalise une méthodologie de développement qui, à plusieurs égards, peut être qualifiée d'Agile.

Dans la deuxième partie, douze pratiques de qualité sont mises en évidence sur base de quatre sources : théorie Agile, recherche au sein de projets Open Source, théorie Open Source, expérience professionnelle. Le *chapitre 4* est une introduction aux concepts de qualité et d'assurance qualité. Le *chapitre 5* exprime quatre pratiques issues du monde Agile, clairement orientées qualité et qui peuvent être utilisées dans l'Open Source. Le *chapitre 6* est une étude de l'outillage utilisé dans l'Open Source, de la théorie des métriques et de différents projets Open Source, pour mettre en évidence des pratiques de qualité. Enfin, le *chapitre 7* formalise les différents concepts de qualité définis dans les chapitre précédents, pour élaborer un ensemble de 12 pratiques de qualité applicables à la méthodologie de développement Open Source.

La troisième partie, a pour objectif de valider les différentes pratiques mises en évidence dans la partie précédente, avec le modèle d'assurance qualité CMMi. Le *chapitre 8* définit le modèle CMMi. Le *chapitre 9* tente de réconcilier CMMi avec les différentes pratiques mises en évidence dans la deuxième partie.

Première partie

Méthodologies Adaptatives

Chapitre 1

Les méthodologies "Agile"

"Pour réussir, il ne suffit pas de prévoir, il faut aussi savoir improviser"

(Isaac Asimov)

1.1 Le manifeste Agile

Le concept de "méthodologie Agile" a vu le jour en 2001, au terme d'un séminaire extraordinaire de 2 jours, réunissant des méthodologistes de différents horizons mus par les mêmes ambitions. Le résultat de ces recherches fut le "Manifeste pour le Développement Agile de Logiciels[1]" . Ce document a pour but de rassembler les valeurs communes de la communauté Agile.

Les 4 valeurs de ce manifeste, sont :

Les individus et les interactions doivent primer sur les processus et les outils

Les "Agilistes" mettent l'accent sur les êtres humains en tant qu'individus et se basent sur l'expertise d'équipes de développement soudées qui communiquent plutôt que sur des procédures et outils performants.

Les logiciels fonctionnels doivent primer sur la documentation exhaustive

L'écriture et la maintenance d'une documentation pléthorique[2] sont extrêmement consommatrices de ressources. Un document qui n'est pas mis à jour devient vite inutile et risque même de fournir le résultat inverse en

[1] http ://www.agilemanifesto.org/

[2] En nombre excessif, surabondant (Le petit Larousse illustré)

FIG. 1.1 – Manifeste Agile [dAdMa]

donnant de fausses informations. Les partisans des théories "Agile" préfèrent donc une documentation succincte sur l'architecture générale du système régulièrement mise à jour ainsi qu'une documentation continue du code.

La collaboration avec le client doit primer sur la négociation contractuelle

Il s'agit de considérer le client comme un partenaire qui participe au développement en fixant les objectifs et en donnant un feed-back continu sur l'évolution du logiciel. Cela demande une grande maturité de la part du client ainsi que du prestataire de services afin qu'une relation de confiance puisse s'établir entre eux. Une bonne connaissance de la réalité opérationnelle du projet par les juristes du dossier est également exigée.

La réponse au changement doit primer sur le suivi d'un plan rigide

Cette idée repose sur le principe que personne ne peut connaître avec précision l'entièreté des besoins dès les premières phases d'un projet. La méthodologie Agile tente d'absorber les différences entre les spécifications initiales et le résultat final durant toute la durée du processus de développement. Cela demande des outils particuliers et une coopération de tout instant avec le client.

1.2 Caractéristiques des projets "Agile"

D'une manière générale, le but est d'augmenter le niveau de satisfaction du client en étant plus réactif au changement. Le travail de développement est facilité grâce à un développement itératif et incrémental basé sur une collaboration réciproque.

1.2.1 Itératif et incrémental

Itératif, c'est le fait de réaliser un peu d'une activité (comme tester, coder, modéliser,...) et puis de passer à la suivante et ainsi de suite. C'est différent d'une approche sérialisée qui consiste à effectuer l'entièreté d'une tâche, avant de passer à la suivante.

Incrémental : Organisation du système en une série de livraisons plutôt qu'en une seule.

1.2.2 Adaptatif

Les méthodologies prédictives ont pour objectif d'établir, dès le début du projet, un planning précis et détaillé de l'entièreté du projet et ce, sur une longue période de temps. Cela suppose que les spécifications restent immuables. Ces méthodologies peuvent fonctionner sur des projets de petite taille mais sont trop réfractaires au changement pour pouvoir être appliquées avec succès sur des projets plus importants.

Dans un environnement adaptatif, la planification est souvent considérée comme un paradoxe, puisque les résultats sont par nature imprévisibles. Si, dans un processus de développement prédictif, les écarts par rapport au planning sont des erreurs qui se doivent d'être corrigées, dans un environnement adaptatif, ces écarts nous guident vers la bonne solution. Le terme de planification est souvent remplacé par le terme spéculation, considéré comme plus approprié.

Dans un environnement adaptatif, utilisateurs et développeurs doivent continuellement se remettre en question. Chaque cycle d'itération doit apporter son lot d'apprentissage et celui-ci doit être utilisé pour adapter les cycles d'itération suivants.

Plans et Design changent au fur et à mesure des progrès du développement, de l'augmentation des connaissances et de l'affinement des spécifications.

1.2.3 Collaboratif

Dans un environnement aussi imprévisible, le défi consiste à trouver une équipe de développement qui communique et collabore efficacement afin de s'accommoder de l'incertain. L'attitude du management doit d'avantage aider et encourager cette collaboration afin que les individualités puissent elles-

mêmes apporter des solutions créatives, plutôt que de leur dicter ce qu'elles doivent faire.

1.3 Les 12 principes derrière le manifeste

Satisfaire le client en livrant très tôt et régulièrement des versions fonctionnelles de l'application

Les méthodes "Agile" recommandent de livrer très tôt (dès les premières semaines) une première version rudimentaire. Et ensuite de livrer fréquemment des versions auxquelles les fonctionnalités s'ajoutent progressivement. Le client, quant à lui, se doit de donner un feedback pour permettre : soit de continuer le développement, soit d'opérer les changements nécessaires le plus tôt possible.

Accueillir les changements à bras ouverts

Ceci implique de devoir produire des systèmes flexibles de sorte que l'impact d'une évolution des spécifications soit aussi minime que possible. Chaque changement d'exigence doit être perçu comme une amélioration vers une meilleure compréhension du système.

Livrer le plus souvent des versions opérationnelles

Il faut garder comme objectif de livrer le plus rapidement possible une solution qui satisfasse les besoins. La fréquence de livraison devrait être comprise entre deux semaines et deux mois.

Coopération entre le client et l'équipe de développement tout au long du projet

Il doit exister une interaction permanente entre le client et l'équipe de développement afin d'affiner le projet.

Axer le projet autour des individus

Dans un projet Agile, les personnes sont considérées en tant qu'individu et non en tant que ressource interchangeable. Elles constituent le facteur clé du succès du projet.

Le premier vecteur de communication est la parole

Le premier mode de communication à l'intérieur d'une équipe Agile est le dialogue et le non-écrit.

Fonctionnement de l'application comme indicateur d'avancement du projet

Le degré d'avancement du projet est évalué en fonction du pourcentage des fonctionnalités effectivement mises en place.

Rythme de travail durable

L'équipe doit adopter un rythme de travail soutenable pour préserver la qualité de son travail tout au long du cycle de vie du projet.

Attention particulière à l'excellence technique et à la conception

Un code robuste et solide améliore l'agilité ; chaque membre de l'équipe doit s'efforcer de produire le code le plus clair, le plus propre et le plus robuste possible.

La simplicité est essentielle

L'équipe a comme consigne de construire le système le plus simple répondant aux besoins actuels afin que celui-ci soit adaptable demain. Les consignes sont : maximisation de la quantité de travail à ne pas faire et non-anticipation des besoins futurs.

Remise en question permanente

Consciente d'être au centre d'un environnement changeant, une équipe agile doit ajuster continuellement son organisation, ses règles et son fonctionnement.

1.4 Extrême programming (XP)

1.4.1 Introduction

Une des spécificités de XP est de réduire les coûts dus aux changements, on peut donc se permettre de prendre des décisions le plus tard possible en attendant les résultats concrets issus du développement. Les spécifications sont écrites au fur et à mesure du projet en enchaînant des itérations rapides de spécifications, développement et livraison.

Plutôt qu'une méthodologie à appliquer mécaniquement, l'Extreme Programming par delà ses quatre valeurs et ses pratiques, tente de mettre l'accent sur le réajustement permanent du projet. Réajustement en ce qui concerne la réalisation du logiciel mais également pour tout ce qui touche au processus suivi par l'équipe. Une équipe XP doit donc être capable de remettre sans cesse en question ses méthodes de travail.

FIG. 1.2 – Principes "Agile" [dAdMb]

1.4.2 Valeurs XP

Les 4 valeurs prônées par XP sont :
- Communication
- Simplicité
- Feedback
- Courage

1.4.3 Pratiques XP

Pratiques pour développer juste : intégration du client dans l'équipe de développement

- *Livraisons fréquentes* : Une version minimale doit être livrée le plus rapidement possible. Après cela, on essayera de trouver un juste milieu entre "releases" fréquentes et fonctionnalités complètes. Des livraisons fréquentes permettent d'obtenir un feed-back continu et rapide.
- *Planification itératives* : Une planification est établie en début de projet et est en permanence revue et remaniée.
- *Client sur site* : Des représentants du client sont intégrés à l'équipe de développement pour définir les tests fonctionnels et donner un feedback de l'évolution du projet.
- *Rythme durable* : L'équipe détermine le rythme de travail lui permet-

tant de fournir un travail de qualité tout au long du projet. Afin de ne pas être surchargée, l'équipe ne doit pas faire d'heures supplémentaires plus de deux semaines de suite, sinon elle doit redéfinir son planning.

Pratiques pour développer vite en acceptant les changements en cours de projet

- *Unit Test* : Ensemble de tests écrits avant le code. Ils permettent de vérifier le code et d'assurer la non régression de l'application, autorisant ainsi des changements sans risques.
- *Code simple et organisé* : Afin de rester le plus clair possible, l'équipe fera usage de la méthodologie KISS(Keep It Simple and Stupid). Elle ne devra également produire qu'une documentation minimale, c'est-à-dire pas plus que ce qui est imposé par le client.
- *Test de recette* : Tests définis selon les critères du client(si possible automatisés) permettant de vérifier fréquemment le bon fonctionnement de l'application.
- *Programmation Just-In-Time* : On ne développe rien qui ne soit utile tout de suite et l'on optimise seulement à la fin du projet.

Pratiques de cohésion de l'équipe de développement

- *Responsabilité collective du code* : Chacun des membres de l'équipe de développement doit se sentir responsable de toutes les parties du code. A ce titre, il peut et doit pouvoir travailler sur toutes les parties du code.
- *Programmation par binôme* : Deux programmeurs par ordinateur : l'un (le driver) est au clavier et écrit le code, l'autre (le partner) veille au code écrit, décèle les problèmes éventuels, fait des suggestions. Ces couples sont appelés à être régulièrement changés.
- *Règles de codage* : Respectées par tous les codeurs.
- *Intégration continue* : L'intégration de nouveaux développements doit se faire le plus souvent possible (tous les jours ou chaque fois qu'une tâche est finie) afin d'avoir une version toujours à jour.

1.4.4 Les acteurs XP

Programmeur

Il est le "Gardien du Code". A ce titre, il écrit, connaît, modifie, sauvegarde, gère les différentes versions, et s'occupe de la transformation en executable du code. Il maintient un code clair, structuré, compréhensible aussi bien par la machine que par l'homme. Pour réaliser ses objectifs, le programmeur devra s'assurer à tout moment que son code fait vraiment ce

qu'on attend de lui en le testant sans relâche. Seul le test permet de déterminer que le programme fonctionne, et toute partie de code non testée doit être considérée comme ne fonctionnant pas. Pour pouvoir assurer qu'il teste ce qui doit l'être, il doit à la fois savoir écouter et comprendre le client mais aussi l'aider à définir ce dont il a réellement besoin.

Pratiques XP associées au programmeur :
- Programmation en binôme
- Tests unitaires
- Conception simple
- Remaniement du code
- Responsabilité collective du code
- Règle de codage
- Intégration continue
- Rythme durable

Client

L'une des particularités de XP est la présence du client sur le site de développement. Un contact direct s'effectue donc entre le programmeur et le client. Cette relation de proximité a pour but d'installer une relation de confiance entre ces deux protagonistes. Le client écrit les exigences fonctionnelles, les valide et fixe les priorités.

Pratiques XP associées au Client :
- Planning game (Planification itérative comprenant la rédaction des scénarios client et les séances de planification)
- Test de recette
- Intégration continue
- Rythme durable

Testeur

C'est le bras droit du client. Il l'aide à écrire les tests fonctionnels de recette. En ce sens il est le garant de ce que fera réellement le logiciel. C'est également lui qui met en place les outils de test et qui les maintiendra tout au long de la durée du projet.

Pratiques XP associées au Testeur :
- Scénarios client et suivi des tests
- Test de recette
- Intégration continue
- Rythme durable

Tracker

C'est lui qui s'occupe du "feedback". Il vérifie les estimations de l'équipe, il s'occupe du "feedback" pour des projets futurs, il planifie les itérations et

vérifie que les objectifs peuvent être atteints avec les ressources et le temps alloués. En aucun cas, il ne prend de décision de son propre chef, il est seulement le révélateur. S'il pense avoir détecté un problème potentiel, il s'en réfère au Coach qui envisagera des solutions.

Manager

Il s'agit du supérieur hiérarchique des programmeurs. Il suit l'avancement du projet, fournit les moyens matériels(bureaux, ordinateurs, salaires,...) et humains (engagement de nouveaux programmeurs,testeurs,...) et demande des comptes à son équipe. Enfin, il s'inquiétera de la satisfaction du client.

Pratiques XP associées au Manager :

- Scénarios client
- séances de planification
- métriques (avancement des tests de recette, vélocité du projet) de la planification itérative
- Rythme durable

Coach

Le coach est le garant du processus. Il vérifie que chacun joue son rôle et que les pratiques XP sont bien respectées. Le coach doit être omniprésent en début de projet(organisation et animation des réunions, mise en confiance du manager, aide à la réalisation des premiers tests client,...). Néanmoins son objectif absolu est que l'équipe fonctionne sans lui.

Remarque

Certains rôles peuvent être pris en charge par la même personne. D'autres sont par nature incompatibles. La figure 1.3 montre les différentes compatibilités des différents rôles XP. Le client ne peut en aucun cas endosser le rôle du programmeur. Il existe par contre une grande adéquation entre les rôles de client et de testeur. Certains rôles ne sont pas conseillés mais ne sont pas non plus défendus ; par exemple, il est préférable de ne pas fusionner les rôles de programmeur et de testeur.

La figure 1.4 résume et schématise le rôle de chacun des intervenants d'une équipe XP.

1.4.5 Phases du projet

- *Planning* : Réunion entre le client et les développeurs. Les développeurs estiment le temps nécessaire pour achever les "customers stories" et le client définit les objectifs.
- *Small release* : Un premier système simple est produit.

	Programmeur	Client	Testeur	Tracker	Manager	Coach
Program-meur		✘	~	~	✘	~
Client	✘		✔	✘	✘	✘
Testeur	~	✔		✘	✘	✘
Tracker	~	✘	✘		~	~
Manager	✘	✘	✘	~		✘
Coach	~	✘	✘	~	✘	

FIG. 1.3 – Compatibilité des rôles d'une équipe XP

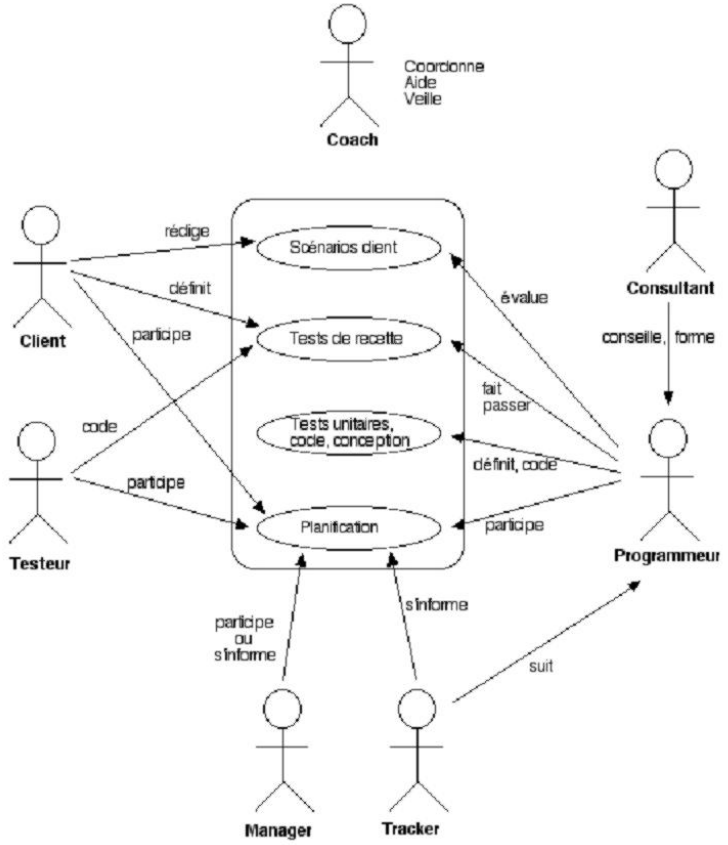

FIG. 1.4 – Résumé des rôles d'une équipe XP

- *Simple Design* : Création d'une première architecture la plus simple possible. Seuls les besoins réels sont modélisés. Complexité superflue et extra-code sont éliminés.
- *Testing* : Écriture des tests unitaires par le client (et un aidant ...) et développement de ceux-ci par les développeurs ou par les testeurs.
- *Pair programming* : Deux développeurs sur une seule machine.
- *Intégration continue* : Nouvelles "releases" le plus souvent possible.

1.4.6 Conclusion sur XP

Cette méthode est critiquée. Elle ne serait que la somme relookée de méthodes largement utilisées et qui ont fait leurs preuves. De plus il faut mettre en place TOUTES les pratiques si l'on veut faire de l'XP, ce qui est plutôt contraignant dans le cadre de théories prétendument "Agile".

1.5 D'autres méthodologies

1.5.1 Rational Unified Process (RUP)

On retrouve la griffe d'IBM dans ce titre puisque cette méthode a été réalisée par Rational Software[3]. RUP fait partie des méthodologies qui ont comme objectif de mener des projets de taille assez importante même si ses auteurs affirment qu'il est possible de l'adapter pour des petits projets. RUP propose une méthodologie rigoureuse et formelle qui couvre l'ensemble du processus de création du logiciel. Sa mise en application demande un investissement assez important (plusieurs centaines de pages de documentation). Une formation préalable semble indispensable pour se lancer dans cette méthodologie.

Ce modèle a un coût d'investissement très important et, en un sens, il peut être considéré comme pas tellement "Agile" car trop rigide. Il s'agit ici cependant d'un véritable outil complet de gestion de projets qui fournit un cadre générique qu'il faut adapter à chaque application. Il est centré sur une architecture incrémentale et itérative ; le développement s'articule autour du modèle objet et de cas d'utilisation, et favorise UML en tant que langage de modélisation. RUP affecte à chacun un rôle très clair. Il est formé d'un découpage en quatre phases : initiale (objectifs et cas d'utilisation), élaboration (architecture logicielle), construction (développement), transition (déploiement). Chaque phase se découpe en itérations. Chaque itération a une durée de deux semaines à six mois et fournit une version de l'application.

Les principales pratiques liées à RUP sont :

- un développement itératif et piloté par les risques (les cas d'utilisation sont pris en compte en fonction des priorités)
- les besoins du client sont pris en compte et formalisés avec précision

[3] Rationnal Software fait partie d'IBM

- l'architecture est fondée sur des composants objets
- les modèles sont visuels, et formalisés avec UML
- la qualité du logiciel est vérifiée régulièrement par des tests systématiques et par son intégration continue
- les changements sont contrôlés avec rigueur

Moins "extrémiste" que XP, RUP donne en même temps une apparence de plus grande rigueur, ce qui peut rassurer certains clients habitués aux documentations exhaustives.

1.5.2 Scrum

Mise au point par Ken Schwaber, cette méthodologie met l'accent sur le facteur humain et s'apparente en certains points à une méthodologie de ressources humaines. Cette méthodologie est parfois mise en oeuvre avec d'autres méthodologies "Agile" pour donner à l'équipe de développement un capital de prise en compte du facteur humain.

Théoriquement, Scrum peut se marier avec XP en fournissant aux développeurs "agiles" un cadre de prise en compte du facteur humain. On peut alors envisager des projets plus importants qu'avec le simple XP. Comme RUP, il s'agit d'un processus incrémental, mais bien moins formalisé : Scrum ne propose aucune pratique de développement, juste des pratiques de management. Il s'agit en fait d'un cadre de gestion de projets bien adapté aux méthodes de développement "Agile", comme XP.

Les principales caractéristiques de Scrum sont :

- identifier les changements très tôt
- donner toute confiance aux développeurs
- réaliser des itérations de 30 jours ("sprints") pour laisser le temps aux programmeurs de coder. Chaque itération a un objectif bien précis ("backlog") et fournit de nouvelles fonctionnalités testées
- organiser des réunions tous les jours pour encadrer les équipes et recentrer les objectifs

1.6 Conclusion sur les développements "Agile"

Les principaux objectifs des méthodologies "Agile" sont : satisfaction du client, simplicité et vitesse. On parle de méthodologie "Agile" [ASRW02] lorsque le développement est

- *Incrémental* : Cycle est constitué de rapides successions de petites "releases" logicielles
- *Collaboratif* : Collaboration entre le client et les développeurs, accent mis sur la communication
- *Simple et directe* : Méthode facile à apprendre et à modifier ; elle est également bien documentée
- *Adaptatif* : Changements acceptés tout au long du cycle de vie

XP	Développement guidé par le client. Équipes réduites. Les développeurs sont au premier plan. Binôme. Builds journaliers. Amélioration constante de la qualité. Adaptativité aux modifications.	Focalisation sur l'aspect individuel du développement au détriment d'une vue globale et des pratiques de management ou de formalisation. On risque ainsi de manquer de contrôle et de structuration en laissant les développeurs trop libres et de dériver par rapport aux fonctions de l'application.
RUP	Processus complet assisté par des outils. Exhaustif. Rôles bien définis. Modélisation.	Lourd, largement étendu. Convient pour les gros projets qui génèrent beaucoup de documentation.
SCRUM	Petites équipes, itération de 30 jours. Réunions journalières	La mise en oeuvre du développement n'est pas précisée, seule compte la gestion des ressources humaines.

TAB. 1.1 – Comparatif de différentes méthodologies "Agile"

Chapitre 2

Le modèle "Open Source"

"Show me the source"

(Linus Torvalds)

2.1 Historique

L'histoire du logiciel libre n'est pas aussi récente qu'on pourrait le penser. Dans les années 60 et 70, pour des raisons de compatibilité matérielle, il était indispensable de pouvoir reconfigurer et recompiler le code du logiciel en fonction du "hardware". La possession du code source du logiciel était dès lors indispensable.

Les années 70 furent marquées par l'arrivée de sociétés ayant pour but de vendre du logiciel. La très grande diffusion des micro-ordinateurs durant les années 80 et 90 n'a fait que renforcer cette position. La plupart des développeurs étaient engagés par des sociétés privées qui ne prenaient en compte que l'aspect commercial du logiciel.

En réaction à cette situation, Richard Stallman lance un projet de re-dynamisation des logiciels libres. Le but principal de ce projet est de recréer un ensemble d'outils de programmation . Cette initiative débouchera sur la création de la "Free Software Foundation" (FSF). C'est dans cet élan que se crée également la licence GPL devant remplacer la notion de copyright.

L'amélioration des canaux de communication et plus particulièrement le grand déploiement d'Internet a favorisé l'augmentation du nombre de personnes ayant un intérêt idéologique ou économique pour l'Open Source .

Les succès incontestables de Linux écrit par Linus Torvalds, alors jeune étudiant en informatique de 21 ans, et des succès plus récents comme Apache ou encore Perl, Emacs ou Latex, Netscape et Mozilla, ont montré que cette méthodologie de développement était viable, aussi bien économiquement qu'humainement . Elle prend de nos jours de plus en plus d'ampleur.

2.2 Définition

Le concept d'Open Source et de Logiciel libre évoquent les mêmes valeurs. Néanmoins, le débat entre les deux fait rage. On leur porte même des valeurs philosophiques. Alors que la FSF[1], qui défend le concept de "logiciel libre", insiste avant tout sur la dimension sociale, l'OSI[2], défendant le concept "Open Source" insiste, quant à elle, sur la dimension économique.

2.2.1 Le logiciel libre

Appartient au logiciel libre tout logiciel dont la licence octroie les quatre libertés suivantes aux utilisateurs :

- Liberté d'exécution du programme, pour tous les usages.
- Liberté d'étude du fonctionnement du programme et de l'adaptation selon ses propres besoins. Pour ce faire, l'accès au code source est une condition essentielle.
- Liberté de redistribution de copies.
- Liberté d'amélioration du logiciel et de publication de ses améliorations. L'accès au code source est ici requise.

En outre, la FSF a créé le concept de "copyleft" qui consiste en une licence qui reprend les termes ci-dessus et qui impose que ceux-ci soient repris à l'identique lors de la redistribution.

2.2.2 L'Open Source

L'appellation "Open Source" implique plus que la simple diffusion du code source. Pour qu'un programme soit caractérisé d'Open Source, sa licence doit répondre aux critères suivants :

Libre redistribution

La licence ne doit pas empêcher de vendre ou donner le logiciel en tant que composant d'une distribution, d'un ensemble contenant des programmes d'origines diverses. La licence ne doit pas exiger que cette vente soit soumise à l'acquittement de droits d'auteur ou de royalties.

Cela signifie que l'on peut faire autant de copies du logiciel qu'on le souhaite et les vendre ou les donner, sans devoir donner de l'argent à qui que ce soit pour bénéficier de ce privilège.

Code Source

Le programme doit inclure le code source et la distribution sous forme de code source comme sous forme compilée doit être autorisée. Quand une

[1] Free Software Foundation
[2] Open Source Initiative

forme d'un produit n'est pas distribuée avec le code source correspondant, il doit exister un moyen clairement indiqué de télécharger le code source depuis l'Internet, sans frais supplémentaires. Le code source est la forme la plus adéquate pour qu'un programmeur modifie le programme. Il n'est pas autorisé de proposer un code source rendu difficile à comprendre. Il n'est pas autorisé non plus de proposer des formes intermédiaires, comme ce qu'engendre un pré-processeur ou un traducteur automatique.

Le code source est un préliminaire nécessaire à la correction ou à la modification d'un programme. L'intention est ici de faire en sorte que le code source soit distribué aux côtés de la version initiale de tous les travaux qui en dériveront.

Travaux dérivés

La licence doit autoriser les modifications, les travaux dérivés et les redistributions, sous les mêmes conditions que celles qu'autorise la licence du programme original.

L'intention est ici d'autoriser tout type de modification et la redistribution sous les mêmes conditions de licence. Il ne s'agit nullement d'une obligation, mais la possibilité de redistribution doit être offerte, que cela soit selon les mêmes conditions ou selon des conditions différentes. Les différentes licences traitent ce problème de manières différentes : BSD autorise la privatisation des modifications, GPL l'interdit.

Intégrité du code source de l'auteur

La licence ne peut restreindre la redistribution du code source sous forme modifiée que si elle autorise la distribution de "fichiers patchs[3]" aux côtés du code source dans le but de modifier le programme au moment de la construction.

Cette clause permet d'imposer que les modifications soient bien distinctes du travail de l'auteur initial qui pourrait craindre que l'utilisateur du logiciel "modifié" puisse penser que le travail est uniquement son oeuvre (cas de bogues, défaillances intentionnelles, malveillances...)

La licence doit explicitement permettre la distribution de logiciels construits à partir du code source modifié. La licence peut exiger que les travaux dérivés portent un nom différent ou un numéro de version distinct de ceux du logiciel original.

Par exemple, seul Netscape a le droit de distribuer une nouvelle version du programme nommée Netscape navigator(tm), les autres distributions doivent porter un autre nom, comme Mozilla ou autre chose.

[3] Fichier comprenant la correction ou la modification d'un programme

Pas de discrimination entre les personnes ou les groupes

La licence ne peut opérer aucune discrimination à l'encontre de personnes ou de groupes de personnes.

Ce qui signifie que le logiciel doit pouvoir être utilisé par tous indifféremment des us, coutumes ou croyances. L'utilisation du logiciel ne pourrait donc être interdite à des peuples qui, par exemple, ne respecteraient pas certaines convictions ou seraient en guerre.

Pas de discrimination entre les domaines d'application

La licence ne peut pas limiter le champ d'application du programme selon l'usage qui en est fait.

Par exemple, elle ne peut pas interdire l'utilisation du programme pour faire des affaires ou dans le cadre de la recherche génétique ; elle ne peut pas non plus apporter des restrictions aux cliniques pratiquant l'avortement ou l'euthanasie.

Non spécificité de la licence à un produit

Les droits attachés au programme ne doivent pas dépendre du fait que le programme fait partie d'une distribution logicielle spécifique. Si le programme est extrait de cette distribution et utilisé ou distribué selon les conditions de la licence du programme, toutes les parties auxquelles le programme est redistribué doivent bénéficier des droits accordés lorsque le programme est au sein d'une distribution originale de logiciels.

Cela signifie que l'on ne peut contraindre un produit identifié en tant qu'Open Source à être utilisé en tant que partie d'une distribution particulière de Linux, ou autre(Windows...). Il doit garder les mêmes spécificités, même séparé de la distribution logicielle avec laquelle il a été fourni.

Non-contamination de la licence à d'autres logiciels

La licence ne peut pas apposer de restrictions à d'autres logiciels distribués avec le programme qu'elle couvre.

Ainsi, la licence ne peut pas exiger que tous les programmes distribués grâce au même médium soient des logiciels "open-source".

2.3 Licences

2.3.1 Définitions

Droit d'auteur

Les droits d'auteur ont été créés pour protéger la mise en forme des idées d'un créateur ainsi que leur expression. Pour qu'une oeuvre soit protégée par les droits d'auteur, celle-ci se doit d'être originale. Il est intéressant de noter que la protection de ce droit ne requiert aucune formalité. L'oeuvre est protégée du seul fait de sa création. L'auteur bénéficie alors de droits patrimoniaux et de droits moraux. Les droits patrimoniaux permettent à l'auteur d'obtenir une rémunération en échange de l'utilisation de son oeuvre. Les droits moraux représentent l'expression du lien qui unit l'auteur à son oeuvre (respect de son nom, de sa qualité et de son oeuvre), Ce deuxième droit est perpétuel, inaliénable et imprescriptible.

Brevet

La notion de brevet protège le concept de l'oeuvre. C'est le contrat entre la société et l'inventeur. L'inventeur dispose ainsi du droit exclusif d'exploitation de son oeuvre durant un certain laps de temps. En contrepartie, l'inventeur s'engage à divulguer son invention au public.

Licence

La licence se définit comme un contrat entre deux parties, le détenteur des droits patrimoniaux et l'utilisateur. Ce document contractuel définit les droits et devoirs de l'utilisateur et les obligations du titulaire, comme par exemple, l'existence (ou la non-existence) d'une garantie de bon fonctionnement.

Les textes des licences Open Source ne sont (en général) pas couverts par la définition de l'Open Source. Une licence n'offrirait qu'une valeur symbolique si chacun pouvait la modifier.

Copyleft ou Gauche d'Auteur

Le copyleft utilise les lois du copyright non pour privatiser le logiciel, mais plutôt pour le rendre libre de façon perpétuelle. Afin de protéger les libertés fondamentales de tous les utilisateurs, le copyleft énonce que quiconque redistribue le logiciel (avec ou sans modification) doit le redistribuer sous les mêmes termes que la licence originale. Grâce au copyleft, le code ainsi que les libertés attenantes deviennent juridiquement insécables.

Concepts	Signification
Droits d'auteur	Concerne les oeuvres de l'esprit et protège la mise en forme des idées et l'expression de l'auteur, celui-ci bénéficie de droits patrimoniaux et de droits moraux
Brevet	Protège les concepts de l'oeuvre. C'est un contrat entre la société et l'inventeur. L'inventeur dispose ainsi d'un droit exclusif d'exploitation sur son invention pendant vingt ans. En contre partie, l'invention doit être divulguée au public. Sa couverture est nationale.
Licence	C'est un contrat entre le titulaire et l'utilisateur. Il transfère ainsi des droits et des devoirs du titulaire à l'utilisateur. Le titulaire y définit aussi ses obligations.

TAB. 2.1 – Droits d'auteur, Brevet et Licence

La licence GPL est un exemple de licence copyleftée. A contrario, les licences X, BSD et Apache sont des licences non-copyleftées.

2.3.2 Le domaine public

Un programme du domaine public est un programme sur lequel son auteur a délibérément choisi de ne pas faire valoir ses droits. Tout le monde a le droit de l'utiliser comme s'il lui appartenait. On peut également lui assujettir une nouvelle licence en ôtant cette version modifiée du domaine public.

2.3.3 GPL : La licence publique générale de GNU[4]

Les termes de la GPL[5] autorisent toute personne à recevoir une copie d'un travail sous GPL. Chaque personne qui adhère aux termes et aux conditions de la GPL a la permission de modifier le travail, de l'étudier et de redistribuer le travail ou un travail dérivé. La GPL indique explicitement qu'un travail sous GPL peut-être (re)vendu. La GPL n'autorise pas à rendre les modifications secrètes. Les copies, ainsi que les modifications, devront être redistribuées également sous la licence GPL.

[4]GNU (acronyme récursif de « GNU's Not UNIX ») est un projet de développement d'un système d'exploitation semblable à UNIX et qui serait un logiciel libre (http ://www.gnu.org).

[5]disponibles à l'adresse électronique suivante : http ://www.gnu.org/licenses/gpl.html

2.3.4 LGPL : La licence publique générale de GNU pour les bibliothèques

Cette licence est dérivée de la GPL et a été mise au point pour les bibliothèques de logiciel. Sa différence fondamentale est la possibilité d'incorporer un programme couvert par GPL à un programme propriétaire [6]. On peut convertir un programme couvert par la LGPL en un programme couvert par la GPL. La réciproque n'est, quant à elle, pas possible.

2.3.5 Les licences X, BSD et Apache

Ces licences sont très proches les unes des autres mais sont très différentes des licences GPL et LGPL dans le sens où ces licences offrent plus de libertés. La permission la plus importante étant certainement la possibilité de rendre secrètes ses modifications. En d'autres termes, un programme couvert par ces licences peut être modifié et ensuite commercialisé dans une version binaire, sans distribuer le code source correspondant et sans appliquer la licence à ces modifications. Cela reste néanmoins de l'Open Source, car la définition de l'Open Source n'exige pas que les modifications soient couvertes par la licence originale.

2.3.6 Les licences publiques de Netscape et Mozilla

La NPL (Netscape Public License) a été développée par Netscape lorsqu'elle a rendu son produit "Open Source". La NPL a ceci de particulier qu'elle renferme des privilèges particuliers dont seul Netscape peut profiter. Netscape a, par exemple, le droit de placer sous une autre licence des modifications apportées à leur logiciel par un tiers. Netscape peut rendre ces modifications secrètes, les améliorer et refuser de communiquer le résultat[7].

La MPL (Mozilla Public License) ressemble très fort à la NPL. Mais ne contient pas la clause permettant à Netscape de placer des modifications extérieures sous une autre licence.

Aussi bien la NPL que la MPL autorisent à rendre toutes modifications secrètes.

[6] Un programme propriétaire est défini par la licence GPL comme : "Tout programme dont la licence vous donne moins de droit que ne vous en donne la GPL"

[7] Cette clause est due au fait qu'à l'époque de l'élaboration de cette licence, Netscape était sous contrat avec des sociétés qui ont exigé d'obtenir une version non Open Source des produits de Netscape.

2.4 Le modèle de développement Open Source

2.4.1 Le paradigme du bazar

Eric S. Raymond est un programmeur de logiciels libres. Il a écrit de nombreux ouvrages sur le monde Open Source dont l'essai sur "The Cathedral and the Bazar [Ray98]" qui a fortement contribué à populariser le monde Open Source. Il a formalisé le paradigme du bazar en se basant sur sa propre expérience lors du développement du logiciel libre "Fetchmail".

Un travail de groupe planétaire

Des milliers de développeurs travaillant aux 4 coins du monde (voir figure 2.1), simultanément sur un projet. Ce modèle de développement distribué est appelé "bazar" à cause de son absence de formalisme.

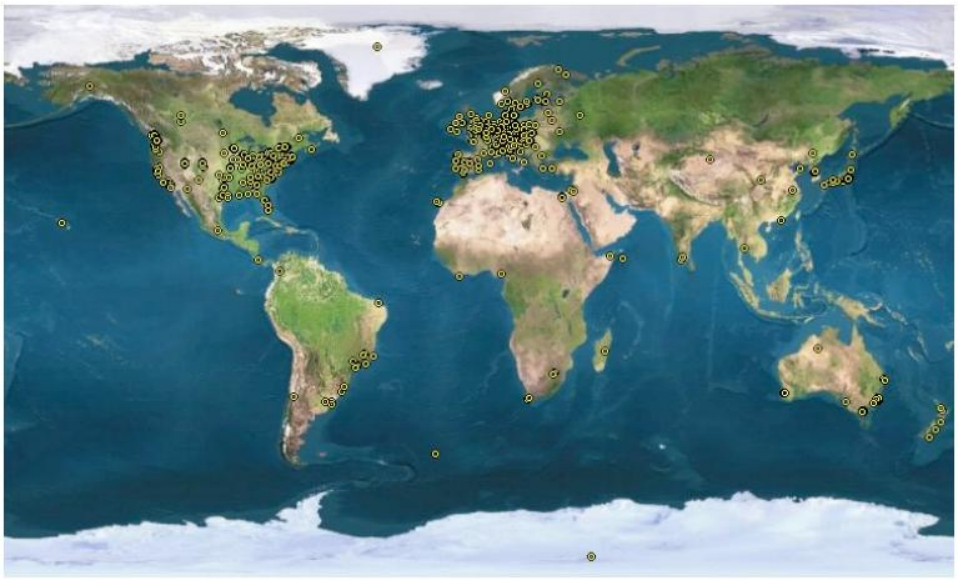

FIG. 2.1 – Répartition géographique des contributeurs au projet Debian (Source : debian.org/devel/developers.loc)

Un programme préexistant

Aucun logiciel ne naît entièrement du "bazar", le concepteur rend public le code source d'un programme préexistant. Celui-ci peut provenir d'une version de test (comme ce fut le cas pour Linux) ou encore venir du monde commercial (avec Mozilla par exemple).

Distributions rapides et fréquentes

"Distribuez tôt et mettez à jour souvent." Autrement dit, adoption d'une démarche itérative et incrémentale.

Débogage complet et rapide

Le développement se base sur l'émulation qui existe entre les différents développeurs. Celle-ci s'obtient grâce à des mises à jour fréquentes reprenant les ajouts faits par chaque développeur. A mesure que le logiciel s'améliore, le nombre d'utilisateurs s'agrandit avec de nouveaux besoins, ainsi que la découverte et la correction de nouveaux bogues.

C'est le concept de révision par ses pairs illustré par la désormais célèbre maxime[8] : *"Étant donné un ensemble de bêta-testeurs et de co-développeurs suffisamment grand, chaque problème sera rapidement isolé et sa solution semblera évidente à quelqu'un"*.

Coordination informelle

Le projet est coordonné par un petit nombre de personnes. Les coordinateurs organisent les prochains axes du projet et publient les changements de version majeurs.

Évangélisation constante

La communauté d'utilisateurs qui se crée autour du logiciel assure également le support technique par le biais des forums de discussion mais aussi par la rédaction et la traduction des documents techniques.

Utilisateurs devenant co-développeurs

Grâce à la disponibilité du code, les utilisateurs peuvent déboguer le logiciel, y ajouter des fonctionnalités et publier leur modification. Ils rentrent alors dans la communauté des co-développeurs.

2.4.2 Les acteurs

Project leaders

Ce sont les garants de la pérennité du projet. Ils peuvent être les initiateurs du projet, les reprenneurs d'un projet en déclin, ou élus par leurs pairs pour un période donnée. Il peut s'agir d'une seule personne, On parle alors souvent de "dictateur bénévole". Comme par exemple Linus Torvalds pour le projet Linux ou encore Jork Janke pour Compiere. Si cette tâche est dévolue à un ensemble de personnes, on parlera alors plutôt de "sénat

[8] Appelée loi de Linus

impérial". On retrouve, par exemple, cette organisation dans les projets de la fondation Apache.

Développeurs volontaires

Il existe plusieurs profils de volontaires :

1. Les membres seniors ou "coeur de développement". Ce sont des développeurs expérimentés, pouvant avoir une expertise dans la problématique. Ce sont les développeurs qui possèdent le plus d'autorité. En plus du développement, ils peuvent guider les nouveaux développeurs, approuver les contributions,...
2. Les développeurs périphériques. Ils produisent du code et le soumettent. Ce sont des développeurs passionnés par le sujet qui décident de donner un peu de temps au projet, ou encore des utilisateurs fréquents du logiciel.
3. Les contributeurs occasionnels. Ils soumettent du code occasionnellement. Souvent, ce code est le résultat d'un développement en réponse à des besoins qui leur sont propres.

Utilisateurs du logiciel

Ils effectuent des tests, identifient les bogues et délivrent des rapports de problèmes.

Posters (post)

Ils participent fréquemment aux forums, mais ne codent pas. Ils peuvent également écrire ou traduire des documents techniques et fonctionnels.

Ces acteurs font partie de la communauté des utilisateurs actifs, car ils contribuent au développement du logiciel. A l'inverse, on nommera utilisateurs passifs, les utilisateurs qui ne désirent pas contribuer aux développements mais qui utilisent néanmoins le logiciel. (Un grand nombre d'utilisateurs passifs permettent d'élever la notoriété du projet et sont donc quelque peu actifs puisqu'ils aideront à faire connaître le logiciel et donc aideront à trouver des utilisateurs actifs).

2.4.3 Les phases du projet

Phase de développement initial

Le départ du projet, c'est de répondre à un besoin. Un initiateur ou une société, veut implémenter un logiciel. Dans le but de partager le travail, celui-

ci décide de publier le résultat de son développement ("release" restreinte, ou projet déjà bien avancé).

Phase d'essor

Durant cette phase, des utilisateurs ont décidés non seulement d'utiliser le logiciel (résultant de la phase 1) mais également de l'améliorer et/ou de l'adapter à leur besoin. Le code du logiciel subit alors une augmentation du nombre de fonctionnalité et une amélioration de la qualité.

Phase d'organisation

Cette phase débute lorsqu'apparaît une équipe autour du développeur initial pour coordonner les changements, découvrir les nouveaux besoins des utilisateurs,... A ce stade, les coûts de maintenance sont alors répartis dans toute la communauté de développeurs, mais de nouveaux coûts inhérents à l'organisation du code, ainsi qu'au maintien de sa cohérence apparaissent.

2.5 Conclusion sur les méthodologies Open Source

L'open source est un type de logiciel défini par une licence octroyant le droit à l'utilisateur d'obtenir le code source en même temps que le programme. Les contraintes (caractère distribué, contributions volontaires,...) liées au développement de tels logiciels ont engendré un nouveau modèle de développement. Ce modèle identifie clairement les différents acteurs et les différentes phases du projet. Il définit également des pratiques de management. Ce modèle peut donc être défini comme une méthodologie de développement. Elle sera tout au long de cette étude nommée "méthodologie Open Source".

Chapitre 3

Conclusion de la première partie

Pour faire face aux problème engendrés par la rigidité des méthodologies prédictives, des méthodologistes ont formalisé un nouveau type de méthodologie de développement : les méthodologies "Agile". Une méthodologie de développement peut être caractérisée d'Agile (voir section 1.6) si elle est : incrémentale, coopérative, simple et directe, et adaptative.

Parallèlement à l'avènement des ces méthodologies, est apparu un nouveau type de logiciels. Ces logiciels se distinguent des logiciels classiques car ils sont distribués avec leur source et sont le résultat de contributions de toute une communauté gravitant autour du projet. Les droits et les devoirs des différents acteurs sont définis dans une licence accompagnant le logiciel.

L'implémentation de tels logiciels a engendré une nouvelle méthodologie de développement de logiciel : la méthodologie Open Source. Cette méthodologie est :

Incrémentale
La méthodologie de développement Open Source est composée de 3 étapes : initial, essor et organisation. Dans chacune de ces étapes, il est également recommandé de réaliser des distributions fréquentes. Ce cycle de vie peut être qualifié d'incrémental.

Coopérative
Le développement Open Source est par définition coopératif puisqu'il s'articule autour d'une communauté travaillant ensemble pour réaliser un objectif commun. C'est l'émulation qui va se développer au sein de la communauté qui va permettre au projet de se développer avec succès.

Simple et directe
Afin de faciliter l'arrivée de nouveaux contributeurs, la méthode est accessible à tous avec un minimum d'effort. La standardisation des outils utilisés dans les différents projets est un exemple de cette volonté de simplifier l'intégration.

Adaptative

L'adaptativité peut être définie selon deux points de vue :

- *Adaptativité du logiciel* : obtenue grâce la disponibilité de son code source ainsi qu'a la forte modularité[1] de celui-ci découlant d'une modélisation spécifique.
- *Adaptativité de la communauté* : les règles qui régissent la communauté permet l'arrivée et le départ de nouveaux acteurs, l'ajout et le retrait de contributions.

Elle fait donc partie de la famille des méthodologies "Agile".

L'existence d'une méthodologie de développement ne garantit aucunement la qualité du produit ni de son application. Même si des exemples de projets réussis sont considérés comme des projets de haute qualité, de nombreux autres projets n'atteignent jamais un stade de maturité. La suite de cette étude va se focaliser sur l'aspect qualité de la méthodologie de développement en se basant sur les théories "Agile" et open Source, ainsi que sur des projets existants pour mettre en évidence des pratiques de qualité.

[1] Considéré par Comino, Manenti et Parisi[CMP05] comme un facteur prépondérant dans la réussite d'un projet Open Source

Deuxième partie

Pratiques "qualité"

Chapitre 4

Introduction à la qualité logicielle

4.1 De la qualité...

La qualité peut être définie comme "une caractéristique ou un attribut de quelque chose". Tout comme l'attribut d'un objet, la qualité fait référence à une caractéristique mesurable, à une chose que l'on est capable de comparer par rapport à un standard. La qualité d'un logiciel, bien que plus proche de l'entité intellectuelle que de l'objet, peut aussi être mesurée. La norme ISO/IEC 9126 définit trois aspects de la qualité logicielle :

- *Qualité du Produit* : Elle est évaluée en mesurant les attributs internes (généralement des mesures du produit à des phases intermédiaires de son cycle de vie), ou externes (mesure du comportement du code lorsqu'il est exécuté).
- *Qualité des processus* : C'est la qualité de l'ensemble des processus du cycle de vie du logiciel.
- *Qualité d'utilisation* : Représente la vue subjective des utilisateurs. Deux utilisateurs différents pourront avoir deux vues différentes. Les métriques relatives à la qualité d'utilisation mesurent comment le produit rencontre les besoins de l'utilisateur pour réaliser certains objectifs en terme d'efficience, productivité, sécurité et satisfaction.

L'évaluation et l'amélioration de la qualité des processus augmentent la qualité du produit, et l'augmentation de la qualité du produit contribue à une meilleure qualité d'utilisation (voir figure 4.1).

Chacun de ces aspects de la qualité peut être évalué en mesurant des attributs spécifiques. A chacun de ces attributs correspondent des métriques logicielles définissant les méthodes de mesures.

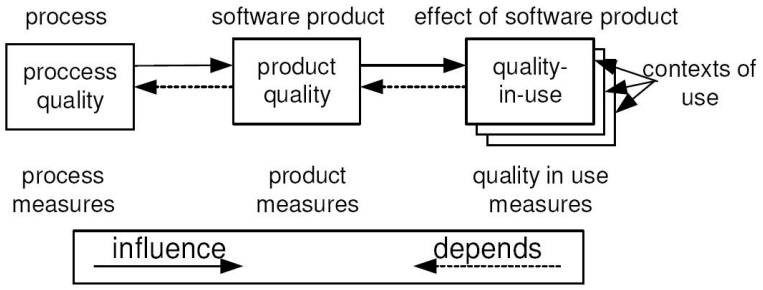

FIG. 4.1 – Les 3 aspects de la qualité logicielle[PH00]

4.2 ...à l'assurance qualité

L'assurance qualité est la mise en oeuvre des activités appropriées permettant à une organisation de donner confiance à ses clients aussi bien externes qu'internes en satisfaisant les exigences de la qualité. Cette mise en oeuvre consiste en la formalisation de "Qui fait quoi ?", "Avec qui ?" et "Comment ?" en se basant sur des méthodes préventives et d'anti-dysfonctionnement.

Lors de travaux effectués au lendemain de la seconde guerre mondiale, Deming démontra que l'obtention de la qualité est dépendante de l'implication de tous les acteurs de l'entreprise : les clients, les fournisseurs, les actionnaires et la collectivité.

L'amélioration de la qualité peut être vue comme un processus itératif prolongeable à l'infini reposant sur un cycle de 4 actions représenté par la "roue de Deming" (voir figure 4.2). Ce cycle est nommé modèle PDCA (Plan - Do - Check - Act) :

Plan Prévoir

Do Faire ce qui est prévu

Check Vérifier et démontrer que l'on a effectivement fait ce qui était prévu

Act Analyser les causes des écarts et améliorer le processus afin de faire mieux la fois suivante

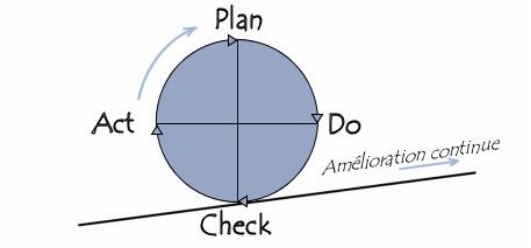

FIG. 4.2 – Modèle PDCA de Deming

La reconnaissance de cette recherche d'amélioration constante peut être obtenue par la certification qui est une reconnaissance par un tiers indépendant. Cette certification s'effectue par rapport à des standards préétablis. Parmi ces standards existent entre autres : la famille ISO 9000[1] ou IEEE[2] ou plus spécifique au logiciel : CMMi décrit au chapitre 8.

[1] International Standard Organisation. Organisme international de normalisation. (http ://www.iso.org)

[2] Institute of Electrical and Electronics Engineers. Organisation ayant pour but de promouvoir la connaissance dans le domaine de l'ingénierie électrique (http ://www.ieee.org/)

Chapitre 5

Qualité dans un monde Agile

5.1 Introduction

Ce chapitre a pour but de mettre en évidence des pratiques de qualité applicables à la méthodologie Open Source en se basant sur des pratiques de qualité de la théorie des méthodologies "Agile".

Au même titre que les méthodologies de développement traditionnelles, les théories de développement "Agile" sont basées sur l'usage de bonnes pratiques de développement. Certaines de celles-ci sont clairement orientées vers la création et la livraison de logiciels de grande qualité. Les quatre pratiques de qualité suivantes sont les plus importantes dans le monde "Agile" :

- Refactoring
- Développement piloté par les tests (TDD)
- Tests remplaçant les documents traditionnels
- Modélisation Agile (AMDD)

5.2 Refactoring

Le "refactoring" est une discipline consistant à restructurer le code source dans le but d'améliorer son design, le rendre plus évident à travailler. L'aspect critique du "refactoring" est de sauvegarder l'aspect sémantique initial.

Le "refactoring" ne correspond pas à l'ajout ou la suppression d'une fonctionnalité. Il se borne tout simplement à l'amélioration de la qualité. Le "refactoring" n'est pas fini tant que le code source ne fonctionne pas de nouveau comme avant (en terme de fonctionnalité).

Beaucoup de développeurs (et en particulier les "XP'ers") considèrent le refactoring comme une pratique essentielle de développement. Ils re-travaillent leur code impitoyablement parce qu'ils estiment qu'ils sont bien plus performant lorsqu'ils travaillent sur un code de grande qualité.

Lors de l'ajout d'une caractéristique au programme, la première question d'un agiliste est de savoir si le code est du meilleur design possible pour ajouter cette caractéristique. Dans la négative, la première tâche sera d'améliorer le design et ensuite d'ajouter la fonctionnalité.

5.3 Développement piloté par les tests (TDD)

Le développement orienté test(TDD[1]) est la pratique de programmation consistant à écrire le test qui vérifiera la fonctionnalité à implémenter avant de la coder. Les avantages sont multiples :

- Phase d'analyse avant l'écriture du code (Design just in time)
- Assurance que le code est testé
- Encouragement à garder un code de meilleure qualité possible (refactoring continu) pour être sûr qu'il continuera à passer tous les tests

La méthodologie à suivre peut être décrite en 6 étapes :

- Écrire le test
- Exécuter le test
- Modifier le code
- Exécuter le test
- Si le test a échoué retourner à l'étape 3
- Dès que le test est réussi, le développement est fini

5.4 Les tests à la place des documents traditionnels

Lorsque l'agiliste devient infecté par les tests, il peut en détourner leur usage premier en leur donnant de nouvelles fonctionnalités. Par exemple, les tests d'acceptation peuvent être considérés comme étant un document d'analyse de besoin (si un test d'acceptation met en évidence un critère que le système doit mettre en valeur, alors il s'agit clairement d'un besoin).

Une autre idée qui circule également auprès des agilistes est que les tests unitaires correspondent à la représentation de l'analyse détaillée. Dans l'approche TDD, les tests unitaires sont écrits avant le code source, c'est-à-dire que le test unitaire est la représentation de ce que le code doit réaliser, autrement dit, le test unitaire devient effectivement la spécification de l'analyse détaillée. Cette philosophie peut réduire considérablement la somme de travail dévolu à l'agiliste.

La version 2 de l'AM[2] inclut une pratique appelée "Source d'Information Unique"[3], conseillant de ne sauver l'information qu'à un seul endroit, et idéalement à la meilleure place possible. La meilleure place pour conserver des informations techniques est généralement considérée comme étant dans

[1] Test Driven Development
[2] Agile Modeling : Technique de modélisation agile
[3] Single Source Information

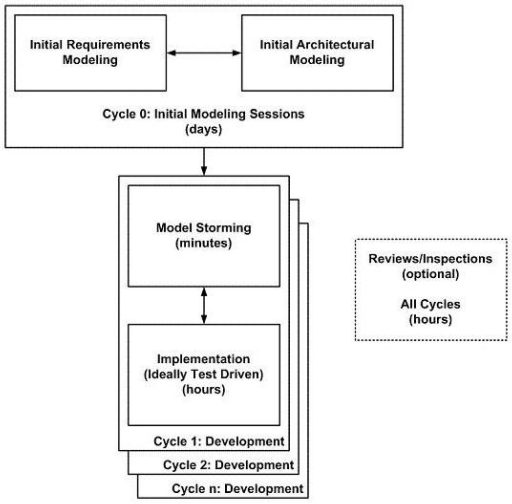

FIG. 5.1 – AMDD - Cycle de vie

le code et dans les tests, plutôt que dans une documentation volumineuse (pléthorique) rarement mise à jour.

5.5 Modélisation Agile (AMDD)

On dit souvent que les agilistes ne modélisent pas. C'est faux, ils modélisent, mais ils n'écrivent pas énormément de documentation car ils estiment que ce n'est pas un bon investissement en temps.

Le cycle de vie de modélisation Agile est représenté à la figure 5.1. A chaque boite correspond une activité de développement. Le cycle 0 : "Initial Modeling" arrive au tout début du projet et dure environ une semaine. Le but de cette étape est d'identifier les besoins initiaux et une première architecture potentielle du système. Il n'est pas nécessaire d'avoir une documentation recouvrant l'entièreté du domaine d'application pour commencer à développer. Les développeurs doivent néanmoins avoir une bonne idée de ce vers quoi ils vont et déjà une petite idée de comment ils vont le faire.

Les détails arrivent après, lors de courtes sessions itératives de "brainstorming" où la modélisation est réalisée en temps réel. Ces séances peuvent être réalisées avec les commanditaires, ou juste effectuées grâce à un outil de modélisation. La réalisation est libre et personnelle mais ne devrait pas durer plus d'une vingtaine de minutes.

AMDD fonctionne pour tout type de projet (aussi bien grand que petit). Au besoin quelqu'un peut prendre en charge l'architecture générale afin de s'assurer que les équipes travaillent bien ensemble dans la même direction.

5.6 Conclusion sur les pratiques de qualité "Agile"

Ce qui est nouveau, ce n'est pas tant ces techniques qui sont bien connues de nombreux méthodologistes. Mais, plutôt, leur soudaine grande popularité. Il semble que les agilistes ont enfin écouté ce que prônent les experts qualité depuis des décennies, à savoir :

1. Écriture et maintenance d'un code de la meilleure qualité possible
2. Réalisation d'une documentation centralisée et à jour.
3. Modélisation continue

L'agilité de la méthodologie Open Source a été prouvée dans la première partie. Ces pratiques de qualité devraient donc également lui être applicables.

Chapitre 6

Qualité dans l'Open Source

6.1 Introduction

Tout comme le chapitre précédent, ce chapitre poursuit l'objectif de mettre en évidence différents facteurs de qualité utilisés ou utilisables dans l'Open Source. Cette recherche se base sur :

- Des études concernant l'outillage utilisé dans l'Open Source
- La discipline des métriques
- L'étude de projets Open Source

6.2 Étude de l'outillage dans l'Open Source

6.2.1 Gestion des versions

Le contrôleur de versions peut être comparé à un entrepôt contenant toujours la dernière version des sources du projet. Il rend les sources accessibles aux personnes ayant le droit de les acquérir. Le contrôleur de versions offre souvent des fonctionnalités d'utilisation non-connectées, de gestion des conflits, de numérotation de versions permettant la récupération de chaque version des sources. Il est également courant de pouvoir configurer un tel outil pour la notification de tout changement permettant une révision par ses pairs.

CVS et Subversion sont les deux outils de gestion de versions les plus représentatifs dans le monde Open Source.

6.2.2 Environnement de développement

Ce sont les outils les plus utilisés par les développeurs. Ils intègrent énormément de fonctionnalités tantôt très génériques, tantôt très précises grâce

à la possibilité d'ajout de plugiciel[1].

"Eclipse" et "Netbeans" sont sans conteste les plus utilisés au sein de la communauté Open Source.

6.2.3 Tests

D'après Roger Pressman [Pre97], le test logiciel est l'élément critique de l'assurance qualité logicielle. De nombreux outils largement répandus dans la communauté Open Source facilitent l'écriture, l'automatisation, l'exécution et la planification de tels tests. Dans un monde "Agile", la bonne réalisation de l'ensemble des tests est la garantie que le système fonctionne bien comme il le devrait. Ces outils sont par exemple JUnit, Cactus ou encore JStyle

On retrouve plusieurs types de tests :

Test unitaire

Le but des tests unitaires est de vérifier la conformité de *chaque* partie du code par rapport à sa spécification. On parle généralement de "white box testing" et de "black box testing", selon que ces tests prennent respectivement en compte la structure interne du logiciel ou non. Dans un monde "Agile", ces tests représentent généralement la spécification technique de la partie de code testée.

Test d'intégration

Si les tests unitaires permettent de vérifier que chaque composant réalise les résultats attendus individuellement, les test d'intégration assurent que tous ces composants continueront à fonctionner correctement lorsqu'ils seront mis ensemble.

Test de validation

Si les tests d'intégration permettent de vérifier que le logiciel, dans son ensemble, répond à ses spécifications, les tests de validation vont, eux, permettre de valider les résultats obtenus dans un environnement aussi proche que possible de l'environnement d'exploitation.

Test de non-régression

Les tests de non-régression servent à vérifier qu'il n'y a pas de dégradation des résultats et des performances entre les différentes versions.

[1] module d'extension(plug-in)

6.2.4 Système de "Build"

Ces outils, ont comme premier objectif, la "construction" automatisée du logiciel. Mais ces outils intègrent dorénavant d'autres fonctionnalités, comme leur planification (toutes les nuits, ou après chaque modification), l'intégration de tests automatisés ou de métriques, et la notification des résultats. Cette combinaison permet d'assurer que le logiciel ne comprend pas d'erreur de compilation et qu'il est conforme aux tests qui ont été écrits.

On retrouve parmi ces outils Maven ou Ant.

6.2.5 Travail collaboratif

Au XVIIIème siècle, Adam Smith soulevait la problématique du travail collaboratif représentée par l'image économique de la fabrique d'épingles : dix ouvriers travaillant chacun de leur côté produiront un nombre incomparablement moins élevé d'épingles que s'ils parviennent à collaborer ensemble dans les différentes étapes de fabrication.

Contrairement aux théories "Agile" qui préconisent l'usage de la parole et du dialogue comme vecteur de communication, les membres des communautés Open Source ont été contraints de s'adapter à un environnement distribué. Cet environnement nécessitait des outils accessibles à tous ses membres (en terme de coûts, de facilité d'utilisation,...) et également des outils minimisant l'administration de ceux-ci (facilité de mise en oeuvre, de maintenance,...). Les forums, wikis[2], mailing-list et FAQ, ont très vite été intégrés comme outils de communication.

L'usage de ces outils permet la sauvegarde tout au long de leur utilisation de tous les échanges d'informations transitant par leur biais, formant ainsi une base de données semi-ordonnée représentant la "mémoire du projet".

6.2.6 Support technique

Ces outils offrent une centralisation des demandes venant des utilisateurs, qu'il s'agisse de rapport de bogues, de demande de fonctionnalité ou de demande de support. Leur simplicité d'utilisation (généralement une simple interface web) les rend (pratiquement) universellement accessibles.

Bugzilla ou Scarab sont les standards actuellement utilisés dans l'Open Source.

[2] Un wiki est un site Web dynamique permettant à tout individu d'en modifier les pages à volonté. Son nom vient du terme hawaiien wiki wiki, qui signifie "rapide" ou "informel". (http ://fr.wikipedia.org/wiki/Wiki)

6.3 Métriques

Les leçons venant d'autres disciplines suggèrent que les métriques peuvent jouer un rôle déterminant en ingénierie logicielle. Le concept de mesure est en effet utilisé dans la majorité des disciplines (électricité, ingénierie, construction, physique,...).

En tant que concept plutôt intellectuel, le logiciel est plus difficilement mesurable que ne peut l'être un objet physique. C'est sans doute pourquoi, en ingénierie logicielle, les métriques sont encore largement sous estimées.

Les métriques existantes recouvrent néanmoins énormément d'aspects : l'estimation des coûts et des plannings, les mesures de productivité et de complexité ou encore le contrôle qualité. Les métriques sont susceptibles d'intéresser tous les acteurs intervenant dans l'ingénierie logicielle.
Les managers s'attarderont davantage sur :

- Le coûts des différentes phases du développement du logiciel
- La productivité de l'équipe de développement
- la répétition des mesures pour établir les facteurs qui influencent les autres métriques
- L'évaluation de l'efficacité de certains modèles et outils de l'ingénierie logicielle

Les programmeurs, quant à eux, les utiliseront dans les domaines suivants :

- Le contrôle de la qualité du système au fil des étapes de développement.
- La spécification de la qualité et des performances en des termes objectivement mesurables.
- La mesure de différents attributs afin de réaliser des prédictions pour les suivants

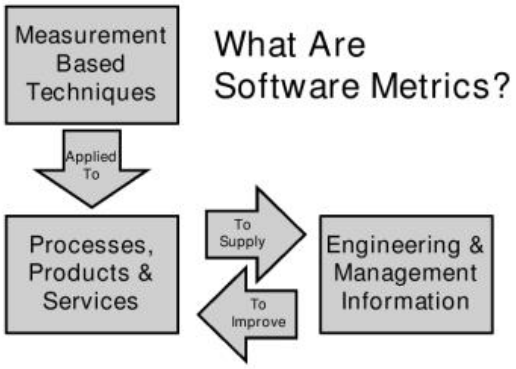

FIG. 6.1 – Que sont les métriques logicielles ?

Les métriques logicielles peuvent être classées en trois familles en fonction de leurs objectifs de mesure :

- Les métriques orientées produit
- Les métriques orientées processus

– Les métriques orientées utilisateur

On notera que certaines métriques appartiennent à de multiples catégories.

6.3.1 Métriques produit

Elles désignent les caractéristiques du produit, la taille, le niveau de qualité ou encore la complexité. Ces mesures peuvent en général être dérivées d'un outil automatisé.

LOC (Line Of Code)

Cette métrique consiste à mesurer le nombre de lignes de code que compte le programme. Elle nous apporte une information sur la taille du produit et est souvent considérée comme un dénominateur idéal pour former un ratio de comparaison. Ainsi, on comptera le nombre de lignes de commentaires par ligne de code, le nombre de bogues par ligne de code,...

La définition d'une ligne de code a évolué au fur et à mesure des années, et de l'utilisation de langages de programmation différents qui impliquent (ou imposent) des façons différentes de programmer. Ainsi le nombre de lignes de code peut être calculé comme étant la somme :

– des lignes exécutables.
– des lignes exécutables et de définition de données.
– des lignes exécutables, de définition de données et de commentaires.
– des lignes physiquement visibles à l'écran.
– des instructions (terminées par un délimiteur logique).

Mesure de complexité

Mesure cyclomatique de Mc Cabe

Mesure développée en 1976 par Mc Cabe. Ce système permet de mesurer le nombre de chemins conditionnels d'un programme et d'y attribuer une valeur de complexité. La mesure cyclomatique d'un programme se calcule par la formule :

$$nombre(arcs) - nombre(noeuds) + 2$$

où un *arc* correspond au chemin entre deux noeuds et un *noeud* correspond à une partie de programme ne contenant pas d'instructions conditionnelles.

Mesure de complexité selon Henry et Kafura

Cette méthode permet de prendre en compte le couplage entre les différents modules d'un programme. On définit :

Fan in : Nombre de modules en entrée du module + nombre de données globales en entrée.

Fan out : Nombre de modules en sortie du module + nombre de données

globales en sortie.
Complexité :

$$longueur * (Fan\ in * Fan\ out)^2$$

où la longueur peut être tout simplement une LOC, une mesure cyclomatique de Mc Cabe, ou tout autre mesure de longueur.
Après avoir validé leur mesure sur UNIX, Henry et Kafura ont constaté qu'une augmentation de leur mesure de complexité entraîne une augmentation des erreurs.

Mesure de complexité adaptée aux langages orientés objet
De par leur particularité objet (encapsulation, héritage,...), les langages de développement orienté objet ont leurs propres métriques de complexité. Le but est de calculer la complexité d'un objet ou d'un module.
"Profondeur d'héritage (DIT)" : Cette mesure représente la longueur du chemin depuis la classe racine. Elle va croître avec le nombre d'héritage.
"Nombre d'enfants (NOC)" : Cette mesure détermine le nombre de successeurs directs d'une classe. Pour un NOC important, il est préférable de tester au maximum les classes parentes.
"Interdépendance des objets" : Cette mesure détermine le nombre de classes qu'une classe particulière référence et qui la référence. Plus cette valeur est grande, plus sa réutilisabilité est faible.
"Calcul de complexité par méthodes" Cette méthode consiste à calculer la complexité d'une classe à partir de la complexité (locale et héritée) de ses méthodes. Ainsi une classe possédant n méthodes M1, M2,..., Mn de complexité C1, C2,..., Cn aura pour mesure :

$$WMC = \sum_{i=1}^{n} Ci$$

Plus WMC est grand, moins l'application est utilisable.

6.3.2 Métriques processus

Elles sont généralement utilisées pour améliorer le développement et la maintenance du logiciel. On retrouve dans cette catégorie, par exemple, le nombre de défauts résolus lors de la phase de développement ou le temps de réponse pour corriger un bogue.

Elles regroupent la plupart des métriques intervenant dans l'amélioration du processus de développement. Elles sont généralement moins formalisées et leurs pratiques peuvent différer très largement de l'une à l'autre. En réalité, le but principal de ces métriques est de gérer les erreurs découvertes lors des tests. Néanmoins, il existe une quantité d'autres métriques qu'il est possible d'utiliser tout au long du processus de développement.

Densité des défauts durant les tests

Il s'agit du nombre total de défauts connus divisé par la taille de l'entité logicielle testée :

$$\frac{\textit{Nombre de défauts connus}}{\textit{Taille}}$$

où la taille est généralement exprimée en KLOC[3] et le nombre de défauts connus est la somme des défauts identifiés dans l'entité logicielle durant un certain laps de temps (depuis la création de l'entité ou depuis la dernière livraison au client).

Cette métrique est utilisée pour comparer le nombre de défauts dans des entités logicielles différentes, ou dans la même entité logicielle sur des versions différentes. Cela, dans le but de connaître l'impact d'une campagne de qualité. La comparaison obtenue avec une livraison précédente doit toujours tenir compte de la qualité des tests effectués pour découvrir ces défauts. Si la qualité des tests est au moins aussi bonne et que le nombre de défauts par KLOC tend à diminuer, c'est que la qualité du produit augmente.

Schéma de découverte des erreurs durant les tests

Il désigne les caractéristiques de détection des défauts lors de la phase de test. A densité égale de défauts découverts durant les tests, les caractéristiques de détection peuvent révéler des niveaux de qualité différents. Voici quelques métriques associées à ce point :

- Le nombre de défauts découverts durant la phase de test par intervalle de temps.
- Le nombre de défauts valides reportés durant la phase de test par intervalle de temps.
- La taille du journal des défauts reportés (au plus la taille du journal augmente, au plus le nombre de corrections augmente, ce qui implique que la stabilité du système va être affaiblie et donc également sa qualité).

Taux de Correction des défauts

Désigne le pourcentage de défauts corrigés dans l'application. Il peut être défini par

$$\frac{\textit{Nombre de défauts découvert lors d'une phase de développement}}{\textit{Nombre de défauts latents dans le système}} * 100\%$$

Le nombre de défauts latents étant ignoré, celui-ci est généralement estimé de la manière suivante :

Nombre de défauts corrigés+nombre total de défauts qui existaient déja à ce stade

[3]KLOC (Kilo Line Of Code) : 1000 Lignes de code

Cette métrique peut être effectuée pour l'entièreté du cycle de développement, ou pour une phase donnée. Plus cette métrique est élevée, meilleure est la phase de développement car cela signifie que peu de défauts sont reportés à la phase suivante.

6.3.3 Métriques utilisateur

Ces métriques mesurent les attributs relatifs à la perception subjective de qualité des utilisateurs.

Métrique de satisfaction du client

Souvent calculée sous forme de sondage sur une échelle de cinq valeurs : "Très Satisfait", "Satisfait", "Neutre", "Insatisfait", "Très insatisfait". Les résultats du sondage, sont généralement exprimés sous quatre formes :

- Pourcentage des clients totalement satisfaits
- Pourcentage des clients satisfaits (Satisfait et Très satisfait)
- Pourcentage des clients insatisfaits (Insatisfait et Très Insatisfait)
- Pourcentage des clients non satisfaits (Neutre, Insatisfait et Très Insatisfait)

La deuxième formulation des résultats est particulièrement utilisée. En pratique la réduction du nombre de clients non satisfaits est privilégiée au détriment de l'amélioration de la qualité du produit. Il est également possible de pondérer la valeurs. Par exemple en attribuant une note de 5 à la valeur 1, 4 à la valeur 2,... et 1 à la valeur 5. Ainsi, si la moitié des client est très satisfaite et l'autre moitié est neutre, la moyenne donnera un taux de satisfaction égal à "Satisfait".

6.4 Étude au sein de projets existants

6.4.1 Apache : Une organisation bien structurée

Apache est une fondation sans but lucratif américaine dont les objectifs sont de :

- Fournir l'infrastructure nécessaire à des projets collaboratifs et ouverts de développement de logiciel.
- Créer une entité légale habilitée à recevoir les dons et pouvant assurer que ceux-ci seront utilisés pour le bénéfice public.
- Fournir un cadre légal protégeant les contributeurs volontaires de poursuites judiciaires.
- Protéger la marque Apache.

La structure de la fondation est composée d'un conseil d'administration responsable de la gestion de la fondation et de la visibilité au niveau des affaires en concordance avec les objectifs, ainsi que de différents comités de

gestion des projets responsables de la gestion d'un ou de plusieurs projets au sein de la fondation.

Les différents contributeurs sont organisés au sein d'une hiérarchie selon une méritocratie très stricte : User, Developer, Committer, PMC Member et ASF Member.

Un grand nombre de règles sont documentées et accessibles auprès du portail de la fondation.

6.4.2 Compiere : Modélisation constante et mainmise sur le CVS[1]

Compiere est un ERP[4] Open Source hébergé par Sourceforge[5]. Il fait partie des dix projets les plus actifs depuis de nombreuses années. Compiere a été écrit par une seule personne et n'accepte que très peu de contributions extérieures. La communauté entourant le projet est constituée de partenaires créant des projets parallèles pour, par exemple, ajouter une traduction ou de nouvelles fonctionnalités. Régulièrement le concepteur de Compiere organise des rencontres avec ses partenaires afin de déterminer ensemble les nouvelles fonctionnalités qui seront intégrées au sein du "package" standard. Les objectifs et leurs priorités sont également discutés au sein de ces réunions.

6.4.3 ERP5 : Procédure d'assurance qualité

ERP5 est un projet Open Source. Afin de garder la consistance du projet, les différents contributeurs doivent respecter certaines conventions :

- convention des noms, des titres, des modules,...
- convention de développement
- convention d'utilisation du CVS[1]

Ces conventions sont formalisées au sein de guides et accessibles à tous.

Outre le respect des conventions, un plan d'assurance qualité est également formalisé. Ce plan prévoit :

- le respect des conventions formalisées dans les guides du développeur
- les procédures de test
- les règles d'accessibilité au code (lecture, écriture, modification)
- l'utilisation de métriques (par exemple, pour déterminer les parties de code nécessitant du "refactoring")

[1] concurent Version System : voir section 6.2.1

[4] "ERP (Enterprise Resource Planning) : applications dont le but est de coordonner l'ensemble des activités d'une entreprise autour d'un même système d'information.

[5] http ://www.sourceforge.net

6.4.4 Subversion : Guidance des nouveaux arrivants

Subversion est un outil de gestion de versions (voir section 6.2.1 au même titre que CVS[1]. Hébergé par Tigris[6], Subversion propose de nombreuses documentations explicatives pour permettre aux nouveaux arrivants de s'intégrer au processus de développement et également s'assurer de la standardisation des contributions.

6.4.5 Conclusion

En plus d'être une aide au développement, les outils utilisés dans l'Open Source sont généralement basés sur l'amélioration de la qualité intrinsèque du produit. L'automatisation de tâches, les fonctionnalités offertes et les règles de travail qui découlent de leur utilisation sont un premier pas implicite vers la création d'une assurance qualité.

Les métrique sont des outils permettant la mesure des attributs du logiciel, de son processus de développement et de son utilisation auprès des utilisateurs. Un programme de métriques basé sur les objectifs de l'organisation doit aider à mesurer et à communiquer les progrès réalisés. Les métriques peuvent aider l'organisation à améliorer ses produits logiciels, processus et services tout en gardant le cap sur ses objectifs.

L'analyse de différents projets a mis en valeur le fait que la qualité dans les projets Open Source est bien un objectif récurrent. Il apparaît également que les moyens pour y parvenir sont bien différents et peu standardisés.

Le chapitre suivant va synthétiser et formaliser un ensemble de pratiques de qualité à partir des observations effectuées au cours de cette seconde partie.

[6] http ://www.tigris.org

Chapitre 7

Formalisation des pratiques rencontrées dans les différents projets Open Source

7.1 Introduction

Cette partie à mis en valeur différentes notions relative à la qualité applicable aux méthodologies open Source. Ce dernier chapitre va les synthétiser et les formaliser pour fournir une librairie effectivement utilisable de pratiques orientées qualité.

7.2 Organisation : l'allégorie du centre commercial

D'après la théorie du bazar, il suffirait de mettre un logiciel à disposition de tous, pour voir affluer les contributeurs. De là découlerait un logiciel plus fonctionnel, sécurisé, stable, innovant et pérenne. En d'autres termes, un logiciel de meilleure qualité.

Néanmoins, cette vision *romantique*[Vis05] des développements Open Source n'est pas conforme à la réalité. Bien que quelques exemples viennent à première vue accréditer la thèse du bazar, comme Apache, Linux ou autre Perl, la majorité des projets Open Source ne compte qu'un développeur (voir fig 7.1). Dans ce cas, le principe de révision par ses pairs ne fonctionne pas et la correction des erreurs n'est plus parallélisée.

Distribuer les sources d'un projet ne suffit pas à rendre Open Source son modèle de développement. Le manque d'ouverture des projets se traduit parfois par l'absence d'outils permettant la collaboration de nouveaux contributeurs.

D'après Robert Visseur[Vis05], il existe entre la cathédrale et le bazar,

	One	Two	Three or four	Five or six	Between 7 and 15	More than 16
Number of projects	57406	13451	8583	3046	2728	586
Percentage	66.9%	15.7%	10.0 %	3.5%	3.2 %	0.7%

FIG. 7.1 – Nombre de développeurs par projet dans l'Open Source[CMP05]

un modèle dit du "centre commercial", qui conserverait les avantages de modularité, de souplesse et d'intégration des diverses contributions, mais qui en plus posséderait une certaine organisation.

Cette organisation doit être structurée avec des rôles, une hiérarchie et des critères de nomination bien définis. Cette structuration conduit à une meilleure répartition des rôles, en fonction des aspirations et compétences de chacun et inspire une certaine confiance. L'organisation peut être démocratique (voir section 6.4.1) ou autoritaire (voir section 6.4.2). Ce qui est important c'est que le cadre soit bien défini. L'organisation doit également être ouverte sur l'extérieur ce qui implique aussi certaines contraintes dans les choix techniques (comme le langage de programmation, l'architecture ou encore le support du multi-linguisme).

7.3 The Wall

Métaphoriquement, le projet Open Source doit être entouré d'une muraille. Cette muraille a pour mission de maximiser le flux d'informations sortant et simultanément limiter et contrôler le flux d'informations entrant. Toutes les procédures critiques ("commit", gestion de la documentation, gestion de l'infrastructure, "build", tests...) du projet sont réalisées à l'abri des murailles du projet.

Ce type d'approche permet aux leaders du projet d'apporter le niveau de rigueur voulu au projet du côté serveur (Server Side), tout en laissant une complète liberté aux contributeurs dans leur espace personnel (Client Side).

7.4 Communication

L'objectif est d'aider les membres de la communauté à travailler ensemble et d'instaurer un climat de confiance entre les responsables et le monde extérieur. Les moyens d'y parvenir sont :

- L'établissement de canaux de communication.
- La formalisation des règles de la communauté
- Le choix de la licence

Un bon climat de confiance entre la communauté et le monde extérieur influence directement la taille de la communauté qui est primordiale[1].

7.5 Guidance

Au delà d'idéologies personnelles, l'utilisation d'outils Open Source relativement standardisés dans les projets Open Source facilite l'arrivée de nouveaux participants en leur permettant de se créer un environnement de développement à moindre coût et avec moins d'effort.

Les portails de projets Open Source et les projets eux-mêmes ont réalisé un effort particulier pour faciliter l'intégration de nouveaux arrivants en mettant à leur disposition différents documents et en leur offrant une aide leur permettant de s'adapter aux exigences des différents projets.

7.6 Outillage Open Source

Cette composante a un impact direct sur les composantes décrites précédemment(Organisation, The Wall, Communication et Guidance) et a pour but l'amélioration ou l'automatisation de celles-ci.

7.6.1 Les outils liés à la communication

Ces outils ont pour principal objectif de permettre aux différents acteurs du projet de communiquer entre eux en réduisant les frontières qu'impose un développement à caractère distribué. Ces outils possèdent deux avantages :

1. L'intégration des fonctionnalités de régulation de l'information (système d'authentification, audit, notification,...) conforme entre autres à la pratique du projet protégé par sa muraille.
2. La sauvegarde de toute information transitant par son biais formant ce que l'on pourrait appeler la "mémoire de l'organisation". Cette information est semi-structurée dans le sens où elle est regroupée par sujet et que le fil de la discussion a, en général, pour but la résolution du problème. Le challenge des projets open source est donc d'arriver à transformer ces bribes d'informations en documents de design structurés[2].

Ces outils sont les forums, mailing-lists, wikis, présentations flash, portails web, documents électroniques,...

[1] Voir loi de Linus section 2.4.1

[2] Le *Hacker's Guide to Subversion* en est un exemple(%http ://subversion.tigris.org/hacking.html)

7.6.2 Les outils liés à l'infrastructure

Ces outils permettent aux différents acteurs de travailler ensemble au sein du projet.

1. Outils liés à la disponibilité du code et des informations : CVS, portails de projets
2. Outils liés à l'écriture du code : éditeurs de code comprenant souvent des plugiciels augmentant le nombre de fonctionnalités dans des domaines bien précis
3. Outils liés à la qualité continue du code : procédures automatisées de "build", tests et mise en production, notification des résultats, de métriques,...
4. Outils liés à la correction des défauts : rapporteur de bogues,...

Parmi ces outils on retrouve : Subversion, Sourceforge, Eclipse, JUnit, Maven, Scarab,...

7.7 Modèle d'évaluation de la qualité

Cette pratique permet d'évaluer objectivement la qualité par des modèles basés sur l'utilisation de métriques. Deux modèles d'évaluation et de sélection de logiciels Open Source sont ici présentés. Leur objectif premier est de réaliser un audit des différentes solutions Open Source existantes pour un domaine précis en fonction de leur qualité. Pour y parvenir, ces différents modèles proposent une démarche permettant de définir objectivement la qualité des solutions testées. A titre d'exemple, deux modèles sont rapidement explicités.

7.7.1 Modèle de Polancic et Horvat

Le modèle de Polancic et Horvat [PH00] est un modèle d'évaluation et de sélection d'un logiciel Open Source en fonction de sa maturité. La première phase de ce modèle a pour ambition d'évaluer objectivement la qualité en se basant sur l'hypothèse que la qualité d'un logiciel Open Source dépend très fortement des attributs de son processus de développement. Ce modèle d'évaluation fait référence aux métriques proposées par le standard ISO 9126-2 :2003. Certaines données sont fournies par le portail web, d'autres proviennent du projet.

Le modèle consiste à calculer, pondérer et normaliser un ensemble de mesures. Leur somme fournira un indice de maturité du projet. Plusieurs facteurs sont pris en compte :

Le *classement* qui détermine comment les utilisateurs classent le projet pondéré par le nombre de votant.

Metric name	Purpose of the metrics	Measurement, formula and data element computations	Interpretation of measured value	Input to measurement	Metric weight
Rating	How do user rate the projects work	x=(v / (v+m)) * R + (m / (v+m)) * C; R = average for the project (mean) = (Rating); v = number of votes for the project; m = minimum votes required to be listed in the top 20 (currently 20); C = the mean vote across the whole report	0 <= x <= 10	Open source portal	High
Vitality	How vital is a project	x=((announcements * age) / (last_announcement))	0 <= x <= 100	Open source portal	Medium
Popularity	How popular is a project	((record hits + URL hits) * (subscriptions + 1))^(1/2)	0 <= x <= 100	Open source portal	Medium
Lifespan	How old is the project	x=n; n= number of days counted from project initialization	0<=x	Project home page	Low
URL Hits	How many users had interest in project	x=a; a=number of URL hits to a specified project, recorded by a specific source	0<=x	Open source portal	Low
Subscribers	How many subscribers has a project	x=a; a=number of persons interested in a project, recorded by a specific source	0<=x	Open source portal	Medium
Number of developers	How many developers has a project	x=a; a=number of developers (a person, who contributes isolated code patches, as wel as a continued contribution of code)	0<=x	Project home page	Medium
Downloadable files	How many project files were already downloaded	x=a; a= number of downloaded files	0<=x	Project home page	Medium
Bug ratio	What is the relation between open and all bugs	X=1-a/b; a=Number of open bugs, b= Total number of bugs	0<=x<=1	Project home page	Medium
Status of the project	What is the status of development of a project	x=a; a={Planning=1, Pre-alpha=2, Alpha=3, Beta=4, Stable=5, Mature=6}	1<=x <= 6	Project home page	High

FIG. 7.2 – Métriques du modèle de Polancic et Horvat

La *vitalité* qui détermine la vitalité du projet est calculée grâce à la formule suivante :

$$\frac{(annonce * age)}{dernière\ annonce}$$

La *popularité* définie par la formule suivante :

$$\sqrt[2]{((recordhit + URLhit) * (souscription + 1))}$$

La *durée de vie* définie par le nombre de jours écoulés depuis l'initialisation du projet.

Le *Nombre de hits* représente le nombre d'utilisateurs intéressés par le projet. Sa mesure est le nombre d'accès au site durant une période donnée.

Le nombre *d'abonnés* au projet enregistré par une source déterminée

le nombre de *développeurs* qui contribuent au projet, que cette contribution ait été unique, occasionnelle ou continue.

L'*activité* calculée selon le nombre de fichiers téléchargés.

Le *ratio de bogues* définit par $1 - \frac{\text{Nombre de bogues ouverts}}{\text{Nombre de bogues total}}$

Le *statut du projet* calculé en associant un nombre au statut du projet : 1 = Planification, 2 = Pré-Alpha, 3 = Alpha, 4 = Bêta, 5 = Stable, 6 = Mature.

7.7.2 Modèle d'évaluation d'aptitude industrielle pour les logiciels libres (BRR)

Le BRR[LS05] (Business Readiness Rating) est une proposition de standard ouvert pour faciliter l'évaluation et l'adoption des logiciels libres. Ce modèle s'articule en 4 phases.

Premièrement, l'évaluation rapide des différents composants à analyser.

Deuxièmement, l'estimation de l'utilisation cible. Cette deuxième phase consiste à estimer les douze catégories suivantes en fonction de leur importance. Seules les sept catégories notées les plus importantes seront retenues dans l'évaluation. Un coefficient sera ensuite assigné à chacune des catégories retenues afin que la somme de ces coefficient totalise 100%.

- *Fonctionnalité* Évalue l'adéquation du logiciel avec les exigences de l'utilisateur
- *Convivialité* Quelle est la qualité de l'interface utilisateur ? Quelle est la facilité d'utilisation du logiciel pour l'utilisateur final ? Avec quelle facilité peut-on installer, configurer, déployer et maintenir le logiciel ?
- *Qualité* Quelle est la qualité de la conception, du code et des tests ? À quel point sont-ils complets et sans erreur ?
- *Sécurité* Comment les exigences de sécurité sont elles prises en compte par le logiciel ? quel est son niveau de sécurité ?
- *Performance* Est-ce que le logiciel fonctionne bien ?
- *Extensibilité* Comment le logiciel tient-il la charge ?
- *Architecture* Le logiciel est-il bien architecturé ? Quel est son niveau de modularité, portabilité, flexibilité, extensibilité, ouverture et quelle est sa facilité d'intégration ?
- *Support* Le composant logiciel est-il bien maintenu ?
- *Documentation* Quelle est la qualité de la documentation associée au logiciel ?
- *Adoption* Le composant est-il largement adopté par la communauté, le marché et l'industrie ?
- *Communauté* Quel est le niveau de vivacité et d'activité de la communauté relativement à ce logiciel ?
- *Professionnalisme* Quel est le niveau de professionnalisme du processus de développement et de l'organisation du projet dans son ensemble ?

Dès que les catégories sont choisies, des métriques leur sont attribuées. Celles-ci ne sont pas définies, elle sont dépendantes des choix de mesure de l'évaluateur en fonction de l'environnement des logiciels à évaluer. Ces métriques sont associées à une ou plusieurs catégories et également pondérée par un coefficient de telle sorte qu'au sein de chaque catégorie, la somme des coefficients totalise 100%.

Troisièmement, la collecte et le traitement des données.

Quatrièmement, déduire des résultats un coefficient d'évaluation.

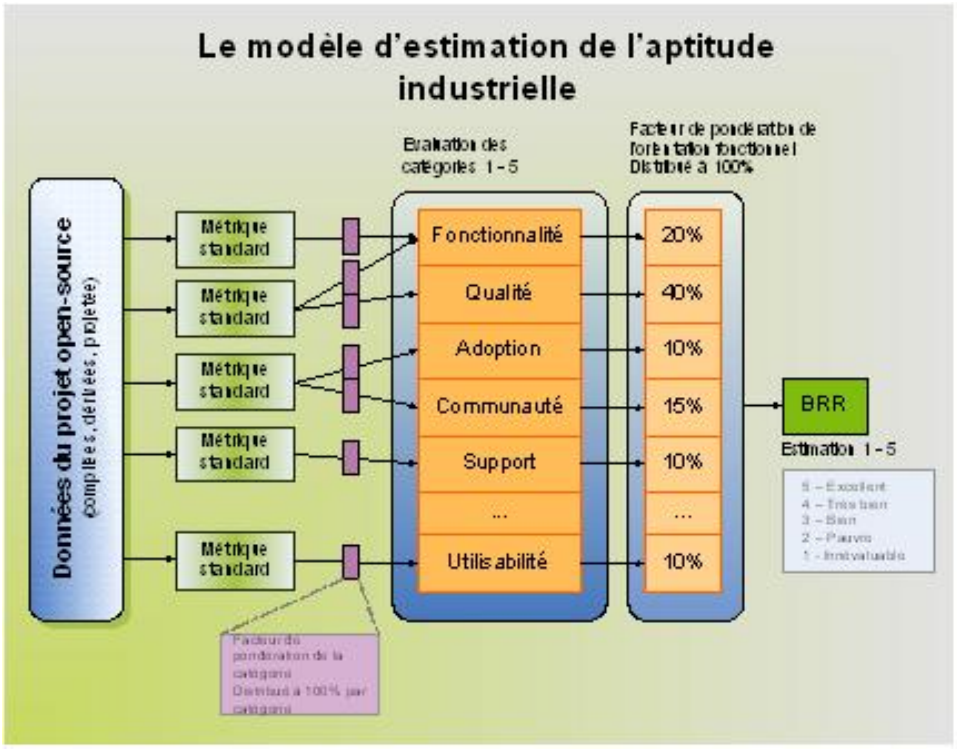

FIG. 7.3 – Modèle d'estimation de l'aptitude industrielle

7.8 Inventaire des différentes pratiques qualité

La formalisation des différentes notions de qualité a permit d'élaborer une librairie de douze pratiques applicables à la méthodologie Open Source.

7.8.1 Pratiques issues des méthodologies "Agile"

De par sa nature "Agile", le développement Open Source suit les pratiques de qualité des méthodologies "Agile". A savoir :

- 1.1 Code de la meilleure qualité possible
- 1.2 Documentation centralisée et à jour
- 1.3 Modélisation continue
- 1.4 Développement itératif et incrémental

7.8.2 Pratiques issues du "bazar"

Modèle de référence de la majorité des projets Open Source, le paradigme du bazar propose les pratiques suivantes :

- 2.1 Travail collaboratif
- 2.2 Révision par ses pairs
- 2.3 Parallélisation du développement
- 2.4 Parallélisation du débogage

7.8.3 Pratiques additionnelles

La recherche empirique des pratiques présentes dans certains projets Open Source a également mis en évidence les pratiques suivantes :

- 3.1 Allégorie du centre commercial
- 3.2 The Wall
- 3.3 Communication
- 3.4 Guidance
- 3.5 Outillage Open Source
- 3.6 Modèle de validation et d'évaluation de qualité

Ces pratiques ont pour objectif d'améliorer la qualité du logiciel, la qualité des différents processus de son cycle de vie ainsi que la qualité liée à son utilisation. Leur application constitue un premier pas vers l'élaboration d'une démarche de qualité.

Pour compléter cette étude, il reste à déterminer quelle est la part d'amélioration de la qualité que peut apporter l'adoption de ces pratiques. C'est le modèle CMMi qui sera utilisé pour effectuer cette validation dans la partie suivante.

Troisième partie

Validation CMMi

Chapitre 8

Capability Maturity Model integration(CMMi)

8.1 Introduction

Au milieu des années 80, le "Département of Defense" (DOD) américain lance un appel d'offre pour la création d'un modèle lui permettant d'évaluer les performances de ses fournisseurs. Ce projet voit lentement le jour et en 1991, est publiée la première version d'un modèle d'évaluation des capacités logicielles : CMM (Capability Maturity Model) mis au point conjointement par SEI et Mitre corporation. Le Software Engineering Institute était le centre de recherche financé par l'armée américaine, tandis que Mitre corporation était une ASBL. Ce modèle (CMM) regroupe un ensemble de pratiques réparties par secteur clé ainsi qu'une méthode d'évaluation du niveau de maturité de l'entreprise sur une échelle de 1 à 5. Il ne faudra attendre que deux ans pour voir une nouvelle version de ce modèle, baptisée CMM 1.1.

Ce modèle va vite dépasser le cadre strict du développement logiciel. Et bientôt de nouveaux modèles complémentaires voient le jour comme, par exemple, SE-CMM (Software Engineering), People-CMM(gestion des ressources humaines). Le modèle CMM initial sera alors renommé SW-CMM (Software) pour éviter toute confusion. Un bon nombre de projets viennent compléter le tableau, citons à titre d'exemple SPICE (Software Process Improvement and Capability determination) de l'organisme ISO.

Après cette période euphorique, succéda une période plus trouble engendrée par la prolifération de modèles. Il fut alors décidé, dans un souci d'harmonisation, de tenter de rassembler tant bien que mal ces divers modèles dans un modèle global baptisé CMMi (Capability Maturity Model integration)

Ce modèle est devenu le standard d'évaluation et d'amélioration de la maturité

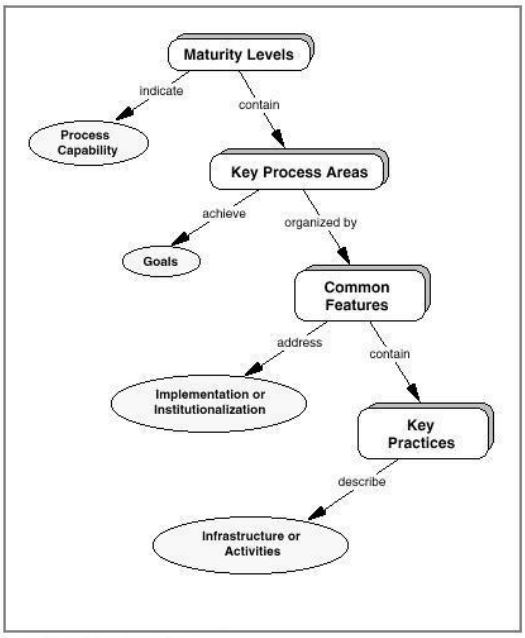

FIG. 8.1 – CMMi - Structure

8.2 Le modèle

CMMi décrit les principes et les pratiques qui sont à l'origine de la maturité dans le développement logiciel. CMMi est un modèle évolutif destiné aux organisations de développement logiciel qui veulent améliorer la maturité de leurs processus chaotiques vers des processus disciplinés et matures. Le modèle CMMi s'articule autour de secteurs clé. A chacun de ces secteurs clé sont associés des objectifs, et à chaque objectif sont également associées un ensemble de pratiques définissant à la fois ce qui doit être réalisé ainsi que les documents à établir. Une pratique est généralement définie par une simple phrase, suivie d'une description plus détaillée, ainsi que des exemples et des conseils d'application. Certains objectifs et pratiques sont appelés génériques s'ils sont communs à tous les secteurs clé ; dans le cas contraire, ils sont appelés spécifiques.

Le modèle complet prévoit vingt-cinq secteurs clé. Néanmoins,il existe des modèles allégés pour des applications du modèle dans des domaines particuliers. Ces secteurs clé peuvent être regroupés en quatre catégories : Gestion des processus, Gestion de projet, Ingénierie et Support. C'est la réalisation de ces pratiques et objectifs qui déterminera si la réalisation d'un secteur clé est atteinte.

Enfin, le modèle CMMi prévoit également deux types de représentation : continue ou étagée. La représentation continue utilise les niveaux de capacité pour mesurer l'amélioration des processus, tandis que la représentation

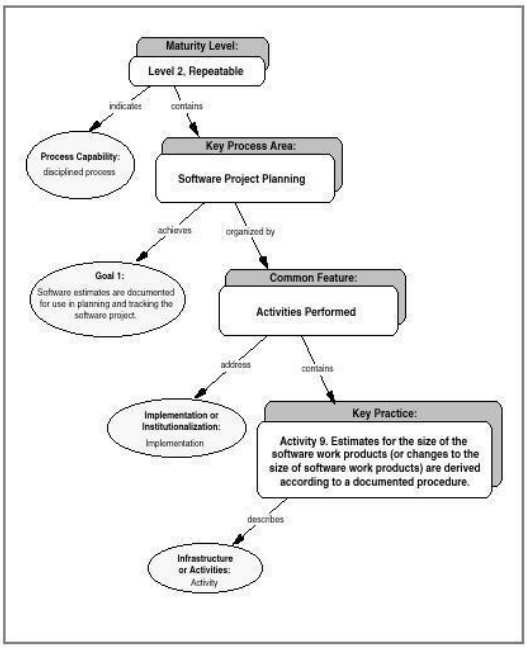

FIG. 8.2 – CMMi - Exemple de pratique

étagée utilise les niveaux de maturité de l'organisation. La vision induite par l'une ou l'autre de ces représentations est véritablement différente. Elle détermine de façon décisive comment les niveaux de maturité ou les niveaux de capacité seront atteints. Les structures plus complexes choisiront la représentation étagée qui, même si elle laisse moins de liberté, fournit une bonne marche à suivre, plus facile à mettre en oeuvre dans des environnements complexes, puisqu'il suffit de valider ou d'invalider chaque secteur clé. Les petites structures, de complexité moindre choisiront, quant à elles, la représentation continue.

Tous les secteurs clé sont communs aux deux modes de représentation.

8.3 Représentation étagée

Dans cette représentation, c'est le niveau global de l'organisation qui est analysé. Le modèle CMMi définit cinq niveaux de maturité permettant de caractériser le profil de maturité d'une organisation sur une échelle de 1 à 5 où le cinquième niveau représente le niveau de maturité le plus important. Un niveau de maturité n'est atteint que lorsque tous les secteurs clé associés à ce niveau de maturité sont atteints.

Cette vision est centrée sur l'organisation puisque la cible de cette représentation est la maturité globale de l'organisation. Elle donne également une marche à suivre claire concernant les objectifs à atteindre et décrit en termes

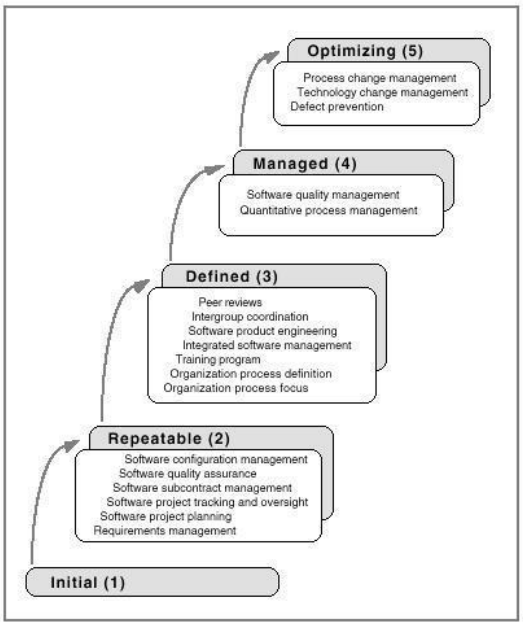

FIG. 8.3 – CMMi - Niveaux de maturité et secteurs clé

généraux comment l'organisation doit améliorer ses processus et évoluer vers le niveau supérieur.

Les avantages de cette représentation sont un bonne guidance et une facilité de mise en oeuvre, tandis que son principal point faible est son manque d'agilité dans son évaluation.

Les 5 niveaux de maturité

Niveau 1 : 'Initial'

Le premier niveau a pour but d'établir une base de comparaison pour l'évaluation des processus d'un niveau supérieur. A ce stade, l'organisation n'a pas ou peu de procédures formalisées. La capacité de développement est assez aléatoire, et la bonne réalisation d'un projet dépend uniquement des aptitudes personnelles des acteurs du projet. Il est pratiquement impossible de réaliser un planning fiable, de déterminer un budget ou encore d'assurer la qualité du produit puisque ces facteurs sont dépendants d'efforts individuels. Durant les périodes de crise, les bribes de méthodologie sont oubliées et la procédure maître devient le "coding-testing".

Niveau 2 : 'Répété' ou 'Reproductible'

A ce niveau quelques procédures de gestion de projets existent afin de pouvoir déterminer les coûts, le planning et la qualité. Une discipline de développement est également mise en place. La capacité de développement est définie comme étant disciplinée car les expériences des projets réussis sont répétées, même si le domaine d'application peut différer et donc révéler quelques surprises. Il en résulte que le projet est sous le contrôle effectif de plans réalistes basés sur l'expérience acquise des précédents projets.

Niveau 3 : 'Défini'

Au troisième niveau de maturité, les différents processus sont documentés, standardisés et intégrés au cycle de développement de processus. La capacité de développement est définie comme étant stable et consistante car standardisée et approuvée. Coûts, planning et fonctionnalités sont totalement sous contrôle. La qualité du logiciel est pistée.

Niveau 4 : 'Maîtrisé'

A ce niveau, l'organisation se fixe des objectifs quantitatifs et qualitatifs. Les résultats sont évalués et quantifiés. Quand les limites à respecter sont dépassées, une attention particulière est posée afin de modifier cette situation.

Niveau 5 : 'Optimisé'

Le processus d'amélioration du logiciel est continuellement soutenu par un grand nombre de retours d'expérience sur des projets antérieurs. A ce stade, l'organisation peut être caractérisée comme étant en amélioration continue.

8.4 Représentation continue

Cette représentation détermine le profil de capacité de l'organisation, non plus global mais par sous-niveaux. Cette représentation permet une grande liberté dans l'établissement des priorités des secteurs clé à améliorer et également une grande finesse d'analyse car tous les secteurs clé peuvent être à tout moment (ré-)évalués. C'est une vision centrée sur le processus puisque sa cible est la capacité des processus.

Les avantages de la représentation continue sont la finesse d'évaluation (par processus) et la liberté laissée dans l'établissement du modèle. Cette plus grande liberté laisse place à plus de subjectivité et offre moins de guidance.

Dans cette représentation, les secteurs clé sont regroupés en quatre catégories : gestion de processus, gestion de projet, ingénierie et support, comprenant respectivement cinq, huit, six et six secteurs clé. À chaque secteur clé est associé un niveau de capacité, sur une échelle allant de 0 à 5 (voir 8.1).

0 - Incomplet	Les objectifs associés à ce secteur clé ne sont pas remplis.
1 - Réalisé	Les objectifs sont atteints, mais cette réussite repose essentiellement sur les individus.
2 - Géré	Les objectifs sont atteints en suivant des plans préétablis.
3 - Défini	Une politique de normalisation des processus est mise en place au niveau de l'organisation.
4 - Maîtrisé	Des mesures sont effectuées pour contrôler les processus et agir en cas de déviation par rapport aux objectifs de l'organisation.
5 - Optimisation	Les processus sont sans cesse remis en question afin d'être toujours en adéquation avec les objectifs de l'organisation.

TAB. 8.1 – Niveaux de capacité des secteurs clé dans la représentation continue

Il est ainsi possible de déterminer le profil d'une organisation en étudiant, pour chaque secteur clé, son niveau de capacité. Tous les secteurs clé n'atteignent pas forcément le même niveau de capacité, ce qui permet de déceler les points forts et les points faibles de l'organisation.

8.5 Équivalence continu - étagé

La représentation continue nous informe sur le niveau de capacité de l'organisation. Il existe néanmoins un modèle permettant de déterminer le profil de maturité de l'organisation.

La figure 8.4 montre les profils cibles à atteindre pour déterminer le niveau de maturité d'une organisation utilisant la représentation continue.

Les zones ombrées indiquent les profils qui sont équivalents au niveau de maturité. Par exemple, pour se situer au niveau 2 de maturité (NM2), l'organisation doit avoir atteint le niveau de capacité 2 (NC2) des secteurs clé du niveau 2 de maturité.

Nom	NM	NC1	NC2	NC3	NC4	NC5
Gestion des exigences	2	Profil Cible 2				
Mesure et Analyse	2					
Supervision et contrôle de projet	2					
Planification de projet	2					
Assurance qualité Processus et Produit	2					
Gestion des conventions fournisseurs	2					
Gestion de configuration	2					
Analyse de décision et résolution	3	Profil Cible 3				
Intégration de Produit	3					
Développement des Exigences	3					
Solution Technique	3					
Validation	3					
Vérification	3					
Définition du processus de l'organisation	3					
Focalisation Organisationnelle sur les Processus	3					
Gestion intégrée de projet	3					
Gestion des risques	3					
Gestion intégrée des fournisseurs	3					
Formation de l'organisation	3					
Constitution intégrée des équipes	3					
Environnement de l'organisation pour l'intégration	3					
Performance Organisationnel des Processus	4	Profil Cible 4				
Gestion Quantitative de Projet	4					
éploiement et Innovation de l'Organisation	5	Profil Cible 5				
Analyse des Causes et Résolution	5					

FIG. 8.4 – Maturité dans la représentation continue

8.6 Impacts et bénéfices

Une étude relative aux impacts de l'utilisation de CMMi auprès de douze organisations de taille et de localisation(USA, Europe, Australie) diverses [DD03] a tenté de démontrer les impacts de l'utilisation d'une démarche de qualité.

Les impacts de l'utilisation d'une démarche de qualité ont été étudiés selon 5 critères : coûts, planification, qualité, satisfaction du client et retour sur investissement.

Coûts

Dans cette catégorie, l'étude à voulu prendre en compte aussi bien le coût total ou intermédiaire du produit que les coûts de correction des défauts,... La moitié des organisations peut donner des exemples de réduction de coût en particulier dans la découverte et la correction de bogues ou défauts mais également dans les coûts globaux. Exemple[1] :

- 33% de réduction pour corriger une erreur (Boeing, Australia)
- 20% de réduction par unité logicielle(Lockheed Martin M&DS)

Planification

Cette catégorie couvre deux aspects de la planification : l'amélioration de prévision du temps nécessaire à l'accomplissement d'une tâche et la réduction du temps nécessaire pour accomplir cette tâche. 2/3 des organisations annoncent avoir pu améliorer positivement la planification de leur projet.

Exemples[1] :

- 50% de réduction sur les délais de livraison (Boeing, Australia)
- 30% d'augmentation de productivité logicielle (Lockheed Martin M&DS)

Qualité

L'augmentation de qualité est habituellement mesurée par la réduction du nombre de défauts à différents stades du processus. Cinq cas peuvent attester une augmentation mesurable de la qualité de leur produit, la plupart du temps due à une diminution de défauts.

Exemples[1] :

- But atteint de 20 + ou - 5 défauts pour 1000 lignes de code (Northrop-Grumman IT1)
- Réduction de 6,6 à 2,1 défauts pour 1000 lignes de code (Northrop-Grumman IT2)

[1] Seuls les exemples qui m'ont semblé les plus représentatifs sont repris ici. Il existe bien d'autres mesures dans l'étude

Satisfaction du client

Bien qu'étant l'un des point centraux dans la recherche d'amélioration de la qualité, peu de mesures objectives ont pu être établies pour ce point. 1/4 des cas montrent une amélioration nette de la satisfaction des clients.

Exemples[1] :

- Reception de 98% du maximum possible des primes sur les livraisons côté client. (NorthropGrumman IT2)

Retour sur investissement

Ce point fondamental doit permettre de déterminer l'impact réel au niveau des coûts de l'instauration d'une démarche qualité. 1/4 des cas attestent d'un retour positif de leur investissement.

Exemples :[1]

- Rapport de 5,1 (Accenture)
- Rapport de 13,1 calculés sur les défauts évités par heure passée en formation et en prévention des défauts (Northrop Grumman IT2)

8.7 Conclusion

CMMi est le modèle de référence relatif à l'évaluation et à l'amélioration de la maturité des processus. Il est destiné aux organisations qui veulent améliorer la maturité de leurs processus chaotiques vers des processus disciplinés et matures. C'est un cheminement qui demande l'engagement de tous les acteurs, qui peut s'avérer assez complexe à mettre en oeuvre (acteurs réfractaires à la modification de leurs habitudes, coûts élevés,...). Néanmoins, les études réalisées à ce jour montrent que la mise en place d'un tel processus de certification qualité est non seulement réalisable mais améliore effectivement la qualité et peut également s'avérer rentable.

Chapitre 9

OOS ET CMMi

9.1 Introduction

Le modèle CMMi consiste en un ensemble de 25 secteurs clé. C'est la réalisation (ou la non-réalisation) de ses processus qui permet de déterminer le niveau de capacité des processus et de maturité d'une organisation donnée. Pour chacun de ces secteurs clé existent un certain nombre d'objectifs. A leur tour, à chaque objectif correspondent un certain nombre de pratiques. Une pratique définit ce qui doit être fait et quels documents doivent être réalisés.

Bon nombre de ces pratiques telles qu'elles sont actuellement définies sont trop contraignantes pour pouvoir être intégrées au coeur d'un développement Open Source qui est et doit rester Agile. De plus CMMi a la réputation d'être ultra documenté. Certaines littératures parlent de 400 documents type différents ainsi que 1000 artefacts[1]. Ce qui va également à l'encontre du manifeste "Agile" qui recommande de se focaliser sur la déliverance d'un logiciel fonctionnel plutôt que sur une documentation pléthorique.

Une clé permettant de rendre CMMi plus Agile est de considérer que les pratiques ont été définies pour éclairer sur la façon de réaliser les objectifs. Pour obtenir la certification CMMi, une organisation doit démontrer que les objectifs des secteurs clé sont réalisés. Ceci peut être fait en mettant en évidence les pratiques associées, mais elles ne doivent pas obligatoirement être celles décrites dans la spécification CMMi. C'est le certificateur qui déterminera si les objectifs des secteurs clé sont bien réalisés grâce à l'application des pratiques de qualité.

Dans ce chapitre, la librairie de pratiques de qualité mises en évidence va être confrontée à CMMi afin de mettre en évidence ce que couvrent les pratiques de qualité au niveau de l'organisation.

Pour rappel, le modèle CMMi comporte 4 catégories comprenant un total

[1] Partie d'information utilisée ou produite lors du processus de développement d'un logiciel.

de 25 secteurs clé. Dans la section suivante, ces secteurs clé vont être associés à 0, 1, ... n pratiques déterminées au chapitre précédent.

9.2 Les secteurs clé

Les secteurs clé de la représentation continue peuvent être regroupés en 4 catégories :

9.2.1 Gestion des processus

Les secteurs clé de cette catégorie contiennent les activités servant à définir, planifier, gérer les différentes ressources, implémenter, contrôler, mesurer et améliorer les processus. On retrouve 5 secteurs clé :

Focalisation organisationnelle sur les processus

L'objectif est d'établir et maintenir une compréhension des différents processus de l'organisation et d'identifier, planifier et implémenter les activités de perfectionnement de ces processus en se basant sur les forces et faiblesses déjà connues. Les améliorations potentielles peuvent être obtenues de sources variées comme des métriques basées sur les processus de l'organisation ou d'organisations externes, des leçons venant du passé, du résultat d'une activité d'évaluation ou encore de recommandations diverses. Ces améliorations doivent tenir compte des besoins et objectifs de l'organisation.

- 3.1 Allégorie du centre commercial
- 3.6 Modèle de validation et d'évaluation de qualité

Définition des processus de l'organisation

L'objectif est d'établir et maintenir une librairie utilisable accessible aux membres de l'organisation. Cette librairie inclut la description des processus du modèle du cycle de vie du développement et des guides d'amélioration continue. Il s'agit du partage des bonnes pratiques et des leçons apprises par l'organisation.

- 3.1 Allégorie du centre commercial
- 3.3 Communication
- 3.4 Guidance

Formation de l'organisation

L'objectif est d'établir un programme de formation comprenant : l'identification des formations nécessaires à l'organisation, la mise en place effective

des formations et le maintien à long terme d'une capacité formative. L'objectif des formations est de développer les compétences et les connaissances des membre de l'organisation afin qu'ils puissent jouer leur rôle de façon effective et efficiente.

La méthodologie de développement Open Source ne prévoit pas de pratiques spécifiques à la formation. Les compétences et les connaissances des membres de la communauté sont assurées par la spécialisation personnelle de ses membres ainsi que par la guidance des membres expérimentés envers les nouveaux arrivants.

Performance organisationnelle des processus

Établir et maintenir une connaissance quantitative des performances des processus standards de l'organisation en fournissant des données et des modèles de compréhension qui seront une aide à la gestion des projets de l'organisation.

- 3.1 Allégorie du centre commercial
- 3.6 Modèle de validation et d'évaluation de qualité

Déploiement et innovation de l'organisation

Sélectionner et déployer des améliorations incrémentales et novatrices qui perfectionnent quantitativement les procédures et technologies de l'organisation dans le but de permettre à l'organisation de réaliser ses objectifs qualitatifs.

- 1.1 Code de la meilleure qualité possible
- 3.1 Allégorie du centre commercial
- 3.5 Outillage Open Source

9.2.2 Gestion de projets

Ces secteurs clé couvrent les activités de planning, de monitoring et de contrôle du projet.

Planification de projet

Établir et maintenir des plans qui définissent les activités du projet. Ces plans pourront être revus durant la progression du projet pour s'adapter aux changements.

- 1.3 Modélisation continue
- 1.4 Développement itératif et incrémental
- 3.1 Allégorie du centre commercial

Supervision et contrôle de projet

Fournir une compréhension de la progression du projet afin de pouvoir appliquer les actions correctives appropriées lorsque le projet dévie significativement par rapport au plan initial.

- 1.3 Modélisation continue
- 2.2 Révision par ses pairs
- 2.3 Parallélisation du développement
- 2.4 Parallélisation du débogage
- 3.6 Modèle de validation et d'évaluation de qualité

Gestion des conventions fournisseurs

Gérer les acquisitions des produits venant des fournisseurs pour lesquels il existe une convention formelle(détermination du type d'acquisition, sélection des fournisseurs, réception des produits, exécution des accords, transit des produits acquis dans le projet).

La méthodologie Open Source ne détermine pas de pratiques propres aux fournisseurs.

Gestion intégrée de projet

Établir et gérer le projet et les implications des différents intervenants dans les processus définis et standardisés de l'entreprise. Ce processus clé couvre également l'établissement d'une vision commune du projet partagée par les différentes équipes qui y sont impliquées.

- 2.1 Travail collaboratif
- 2.2 Révision par ses pairs
- 3.1 Allégorie du centre commercial
- 3.2 The wall
- 3.3 Communication
- 3.4 Guidance

Gestion des risques

Identifier les problèmes potentiels avant qu'ils ne surviennent afin que les activités de traitement des risques associés puissent être planifiées et déclenchées tout au long du cycle de vie du produit et du projet.

- 3.5 Outillage Open Source
- 3.6 Modèle de validation et d'évaluation de qualité

Constitution intégrée des équipes

Former et pérenniser une équipe qui possède les compétences et les connaissances requises à la bonne réalisation des tâches qui lui seront dévolues. Cette équipe devra travailler en collaboration avec les différents acteurs (internes et externes).

- 2.1 Travail collaboratif
- 3.2 The Wall
- 3.3 Communication
- 3.4 Guidance

Gestion intégrée des fournisseurs

Identifier proactivement[2] *les sources des produits qui seront utilisés pour satisfaire les spécifications et pour gérer les fournisseurs en maintenant une relation coopérative avec ceux-ci.*

La méthodologie Open Source ne détermine pas de pratiques propres aux fournisseurs.

Gestion quantitative de projet

Gérer quantitativement les processus définis du projet pour réaliser objectivement les objectifs de performance et de qualité.

- 3.6 Modèle de validation et d'évaluation de qualité

9.2.3 Ingénierie

Les secteurs clé de l'ingénierie couvrent les activité de développement et de maintenance. Au nombre de six, ces secteurs clé sont très fortement reliés les uns aux autres. Cette catégorie rassemble un ensemble de processus appartenant à des disciplines différentes (par exemple, l'ingénierie système et ingénierie logicielle) pour obtenir un processus de développement logiciel global. Les 6 processus clé sont les suivants :

Gestion des exigences

Gérer les spécifications du produit du projet ainsi que ses différents composants. Identifier les inconsistances entre les spécifications, la planification et le produit réalisé.

- 1.1 Code de la meilleure qualité possible
- 1.2 Documentation centralisée et à jour

[2] Avant que la situation ne devienne une source de conflit

- 1.3 Modélisation continue
- 1.4 Développement itératif et incrémental
- 2.2 Révision par ses pairs
- 3.6 Modèle de validation et d'évaluation de qualité

Développement des exigences

Produire et analyser les spécifications du client, du produit et de ses composants.

- 3.5 Outillage Open Source

Solution technique

Modéliser, développer et implémenter la solution technique devant répondre aux spécifications.

- 1.1 Code de la meilleure qualité possible
- 1.2 Documentation centralisée et à jour
- 1.3 Modélisation continue
- 1.4 Développement itératif et incrémental
- 2.1 Travail collaboratif
- 2.3 Parallélisation du développement
- 2.4 Parallélisation du débogage
- 3.5 Outillage Open Source

Intégration de produit

Assembler le produit à partir de ses sous-composants, s'assurer que le produit est bien intégré, fonctionne correctement et enfin, délivrer le produit.

- 2.3 Parallélisation du développement
- 2.4 Parallélisation du débogage
- 3.5 Outillage Open Source
- 3.6 Modèle de validation et d'évaluation de qualité

Vérification

Les pratiques associées à ce secteur clé doivent assurer que le produit correspond à ses spécifications.

- 2.2 Révision par ses pairs
- 2.4 Parallélisation du débogage
- 3.6 Modèle de validation et d'évaluation de qualité

Validation

Démontrer que le produit, ou que l'un de ses composants fonctionne comme prévu lorsqu'il se trouve dans l'environnement prévu.

- 2.4 Parallélisation du débogage
- 3.6 Modèle de validation et d'évaluation de qualité

9.2.4 Support

Les activités regroupées sous le terme de support correspondent aux secteurs clé suivants.

Gestion de configuration

Établir et maintenir l'intégrité du produit en réalisant des identifications de configuration, contrôle de configuration, audit de configuration.

- 1.2 Documentation centralisée et à jour
- 2.3 Parallélisation du développement
- 2.4 Parallélisation du débogage
- 3.6 Modèle de validation et d'évaluation de qualité

Assurance qualité processus et produit

Fournir à l'équipe et aux managers une vue d'ensemble des objectifs, des processus et du produit associé. Les pratiques de ce secteur clé doivent assurer que les processus planifiés sont bien implémentés.

- 1.2 Documentation centralisée et à jour
- 2.2 Révision par ses pairs
- 3.2 The Wall
- 3.5 Outillage Open Source
- 3.6 Modèle de validation et d'évaluation de qualité

Mesure et analyse

Développer et soutenir une capacité de mesure utilisée pour répondre au besoin des managers.

- 3.6 Modèle de validation et d'évaluation de qualité

Analyse de décision et résolution

Analyser différentes possibilités de décision en utilisant une évaluation formelle reposant sur des critères précis. Celle-ci examinera et évaluera toutes

les alternatives.

– 1.3 Modélisation continue

Environnement de l'organisation pour l'intégration

Fournir un produit et une infrastructure de développement intégrée et gérer les individus pour leur intégration.

– 1.2 Documentation centralisée et à jour
– 3.1 Allégorie du centre commercial
– 3.2 The Wall
– 3.4 Guidance
– 3.5 Outillage Open Source

Analyse des causes et résolution

Identifier les causes de défauts et autres problèmes et prendre des mesures afin qu'ils ne surviennent plus dans le futur.

– 2.3 Parallélisation du développement
– 2.4 Parallélisation du débogage
– 3.5 Outillage Open Source
– 3.6 Modèle de validation et d'évaluation de qualité

9.3 Récapitulatif des pratiques qualité par secteur clé

Catégorie	Secteurs clé	Pratique OS
Gestion des processus	Focalisation Organisationnelle sur les Processus	3.1 Allégorie du centre commercial 3.6 Modèle de validation et d'évaluation de qualité
	Définition du processus de l'organisation	3.1 Allégorie du centre commercial 3.3 Communication 3.4 Guidance
	Formation de l'organisation Performance organisationnelle des processus	3.1 Allégorie du centre commercial 3.6 Modèle de validation et d'évaluation de qualité
	Déploiement et innovation de l'organisation	1.1 Code de la meilleure qualité possible 3.1 Allégorie du centre commercial 3.5 Outillage Open Source

Catégorie	Secteurs clé	Pratique OS
Gestion de projets	Planification de projet	1.3 Modélisation continue 1.4 Développement itératif et incrémental 3.1 Allégorie du centre commercial
	Supervision et contrôle de projet	1.3 Modélisation continue 2.2 Révision par ses pairs 2.3 Parallélisation du développement 3.6 Modèle de validation et d'évaluation de qualité
	Gestion des conventions fournisseurs	
	Gestion intégrée de projet	2.1 Travail collaboratif 2.2 Révision par ses pairs 3.1 Allégorie du centre commercial 3.2 The wall 3.3 Communication 3.4 Guidance
	Gestion des risques	3.5 Outillage Open Source 3.6 Modèle de validation et d'évaluation de qualité
	Constitution intégrée des équipes	2.1 Travail collaboratif 3.2 The Wall 3.3 Communication 3.4 Guidance
	Gestion intégrée des fournisseurs	
	Gestion quantitative de projet	3.6 Modèle de validation et d'évaluation de qualité

Catégorie	Secteurs clé	Pratique OS
Ingénierie	Gestion des exigences	1.1 Code de la meilleure qualité possible 1.2 Documentation centralisée et à jour 1.3 Modélisation continue 1.4 Développement itératif et incrémental 2.2 Révision par ses pairs. 3.6 Modèle de validation et d'évaluation de qualité
	Développement des exigences	3.5 Outillage Open Source
	Solution technique	1.1 Code de la meilleure qualité possible 1.2 Documentation centralisée et à jour 1.3 Modélisation continue 1.4 Développement itératif et incrémental 2.1 Travail collaboratif 2.3 Parallélisation du développement 2.4 Parallélisation du débogage 3.5 Outillage Open Source
	Intégration de produit	2.3 Parallélisation du développement 2.4 Parallélisation du débogage 3.5 Outillage Open Source 3.6 Modèle de validation et d'évaluation de qualité
	Vérification	2.2 Révision par ses pairs 2.4 Parallélisation du débogage 3.6 Modèle de validation et d'évaluation de qualité
	Validation	2.4 Parallélisation du débogage 3.6 Modèle de validation et d'évaluation de qualité

Catégorie	Secteurs clé	Pratique OS
Support	Gestion de configuration	1.2 Documentation centralisée et à jour 2.3 Parallélisation du développement 2.4 Parallélisation du débogage 3.6 Modèle de validation et d'évaluation de qualité
	Assurance qualité processus et produit	1.2 Documentation centralisée et à jour 2.2 Révision par ses pairs 3.2 The Wall 3.5 Outillage Open Source 3.6 Modèle de validation et d'évaluation de qualité
	Mesure et analyse	3.6 Modèle de validation et d'évaluation de qualité
	Analyse de décision et résolution	1.3 Modélisation continue
	Environnement de l'organisation pour l'intégration	1.2 Documentation centralisée et à jour 3.1 Allégorie du centre commercial 3.2 The Wall 3.4 Guidance 3.5 Outillage Open Source
	Analyse des causes et résolution	2.3 Parallélisation du développement 2.4 Parallélisation du débogage 3.5 Outillage Open Source 3.6 Modèle de validation et d'évaluation de qualité

9.4 Conclusion

Excepté pour la prise en compte de la gestion des fournisseurs, chacun des secteurs clé est couvert par au moins une pratique de qualité. En mettant ce point de côté, la librairie de pratiques de qualité couvre donc tous les aspects du développement de logiciels Open Source. C'est l'étude de réalisation des secteurs clés, grâce aux pratiques de qualité associées, qui devront permettre au certificateur de déterminer le profil de capacité des processus ainsi que le niveau de maturité de l'organisation utilisant la méthodologie Open Source.

Conclusion

L'Open Source est à la base un type de logiciel dont la principale caractéristique est que le code source est disponible, duplicable et redistribuable. Le succès de projets comme Linux ou Apache a montré que l'implémentation de tels logiciels était réalisable.

Parallèlement au succès des logiciels Open Source, des méthodologies dites Agiles ont vu le jour. Ces méthodologies ont en commun des valeurs et sont appréciées pour leurs qualités d'adaptation tout au long de l'évolution du projet. Leurs principales caractéristiques sont un cycle de vie itératif et incrémental, un travail collaboratif, une méthode simple et directe et une grande capacité d'adaptation aux modifications qui leur sont demandées durant leur cycle de vie.

L'analyse de différents projets de développement de logiciels Open Source a mis en évidence une méthodologie de développement propre aux spécificités de l'Open Source. Celle-ci a tantôt été formalisée par des acteurs de l'Open Source, par exemple l'essai d'Eric S. Raymond sur le paradigme du bazar[Ray98], tantôt modelée tout au long de projets réalisés avec succès. Cette méthodologie de développement peut être qualifiée d'Agile et a été nommée "méthodologie Open Source" dans le cadre de cette étude.

L'analyse de projets Open Source et des théories agilistes a permis de mettre en valeur une librairie comprenant douze pratiques de qualité applicables à la méthodologie Open Source. Ces pratiques sont clairement orientées vers la qualité du produit, vers la qualité des processus du cycle de vie du développement et vers la qualité d'utilisation.

CMMi est un modèle d'amélioration et d'évaluation de la qualité qui a pour objectif la mise en place d'un processus d'amélioration et d'évaluation de la qualité. Ce modèle est généralement réservé aux méthodologies prédictives à cause de son extrême rigidité ainsi que la surabondance de documentation à fournir.

En prenant l'hypothèse que les pratiques définies dans le modèle CMMi ne sont que des outils qui permettent de réaliser les secteurs clé, et que ces outils peuvent être remplacés, il a été possible de transposer ce modèle rigide en un modèle plus agile basé sur les pratiques qualité de la méthodologie Open Source.

Le résultat est prometteur puisque si l'on ne tient pas compte de la gestion des fournisseurs, à chacun des vingt-cinq secteurs clé que compte CMMi correspond une ou plusieurs pratiques de qualité de la méthodologie Open Source.

Cela signifie que la librairie de pratiques de qualité de la méthodologie

Open Source couvre bien l'étendue de la discipline. C'est la mise en place effective de ces pratiques qui permettra de déterminer si l'application des pratiques permet de réaliser les objectifs de chacun des secteurs clés et également de déterminer quel niveau de capacité il est possible d'atteindre.

L'utilisation de la représentation continue de CMMi laisse la latitude que nécessite un projet composé de contributeurs volontaires. La grille de comparaison entre représentation continue et étagée permettra de déterminer le niveau de maturité.

Bibliographie

[Amb05] Scott Ambler. Quality in an agile world. 2005.

[And02] David J. Anderson. Agile development : Weed or wildflower. *CrossTalk*, Oct 2002.

[And05] David J. Anderson. Stretching agile to fit CMMi level 3. 2005.

[ASRW02] Pekka Abrahamsson, Outi Salo, Jussi Ronkainen, and Juhani Warsta. Agile software development methods : Review and analysis. *VTT Publications 478*, 2002.

[BBMW02] J-L Bénard, L. Bossavit, R. Médina, and D. Willimams. *L'Extreme Programming*. 2002.

[Bra] Jean-Yves Le Bras. Qualité logicielle.

[CDS97] Conte, Dunsmore, and Shen. *Software Engineering Metrics and Models*. benjamin/Cummings Publishing Company, Inc, 1997.

[Cet] Cetic. Une synthèse des approches qualité. *http ://www.cetic.be/article189.html*.

[CF78] John D. Cooper and Matthew J. Fisher. *Software Quality Management*. Petrocelli Book, 1978.

[CMP05] Comino, Manenti, and Parisi. From planning to mature : on the determinants of open source take off. juillet 2005.

[dAdMa] Groupe d'Utilisateurs Agile de Montréal. Manifeste agile (fr). *http ://www.pyxis-tech.com/agilemontreal/fr/agileprinciples_fr.html*.

[dAdMb] Groupe d'Utilisateurs Agile de Montréal. Manifeste agile (fr). *http ://www.pyxis-tech.com/agilemontreal/fr/agileprinciples_fr.html*.

[DAR01] Ahern D, Clouse Aaron, and Turner R. *CMMi Distilled*. Addison Wesley, 2001.

[DD03] Goldenson D and Gibson D. Demonstrating the impact and benefits of CMMi © : An update and preliminary results. 2003.

[dF02] Alain de Fooz. Le logiciel libre en quête d'un statut. *Athena*, 2002.

[Eyr99] Eyrolles, editor. *Open Sources : Voices from the Open Source Revolution*. 1999.

[Fai85] Richard Fairley. *Software Engineering Concepts*. 1985.

[FAQ] Faq latex. *http ://www.grappa.univ-lille3.fr/FAQ-LaTeX/*.

[Fen91] Norman E Fenton. *Software Metrics*. Chapman & Hall, 1991.

[FFHL03] Joseph Feller, Brian Fitzergerald, Scott Hissam, and Karim Lakhani, editors. *3rd Workshop on Open Source Software Engineering*. 2003.

[Gre03] Robert L. Greenberg. Open source software development. 2003.

[HS02] T.J. Halloran and William L. Scherlis. High quality and open source software practices. 2002.

[Hum95] Watt S. Humphrey. *A Discipline for Software Engineering*. Addison Wesley, 1995.

[IEE98] *Industry Implementation of International Standard ISO/IEC 12207 : 1995*. 1998.

[JC04] Goffinet J and Age C. CMMi ou comment maîtriser vos développements. *http ://www.zdnet.fr*, 2004.

[Jef00] Ron Jeffries. Extreme programming and the capability maturity model. 01/01/2000.

[JP] Vickoff J-P. Concepts du Capability Maturity Model. *http ://www.rad.fr/introcm2.htm*.

[LS05] Jean-Pierre Lorré and Etienne Savard. Évaluation d'aptitude industrielle pour les logiciels libres (business readiness rating). 2005.

[MRB04] Allan MacCormack, John Rusnack, and Carliss Baldwin. Exploring the structure of complex software designs : An empirical study of open source and proprietary code. 2004.

[Pau94] Mark C. Paulk. How iso 9001 compares with the CMM. *IEEE Software*, 1994.

[PH00] Gregor Polancic and Romana Vajde Horvat. A model for comparative assessment of open source products. 2000.

[Pre97] Roger S. Pressman. *Software Engineering : A Practitioner's Approach*. Mc Graw-Hill, 1997.

[Ray98] Eric Raymond. La cathédrale et le bazar. *Linuw France1*, mars 1998.

[Rob02] Jason E. Robbins. Adopting oos methods by adopting oos tools. 2002.

[Tea02] CMMi Product Team. Capability maturity model® integration (CMMISM), version 1.1. 2002.

[Tri03] Claire Tristram. L'industrie du logiciel prépare sa révolution. *Courrier Internationnal*, 2003.

[tSB99] Bruce Perens (traduction : Sébastien Blondeel). La définition de l'open source. *http ://www.linux-france.org/*, 1998-1999.

[Vab00] Petr N. Vabishchevich. Latex help e-book. *www.imamod.ru/˜ vab/*, 2000.

[Vis05] Robert Visseur. Comment assurer des développements logiciels libres de qualité ? 2005.

[Wes05] Linda Westfall. 12 steps to useful software metrics. 2005.

[Zah98] Sami Zahran. *Software Process Improvement*. 1998.