Extreme Programming
Méthodes Agiles

TOUR D'HORIZON

Business Interactif

Table des matières

1. INTRODUCTION

Nous sommes heureux de vous présenter notre nouveau white paper sur les méthodes agiles de gestion de projet et de développement, Extreme Programming étant probablement la plus connue de ces méthodes. Dans un contexte économique plus tendu, le succès des projets informatiques devient toujours plus critique. La démarche méthodologique adoptée joue pour beaucoup dans le succès de ces projets. Les méthodes agiles constituent elles le nouvel eldorado ? Business Interactif fait le point sur le sujet…

Business Interactif en quelques mots

Business Interactif conçoit et réalise de grands projets e-business à forte composante technologique, couvrant les aspects front, middle et back-office. Avec plus de 220 collaborateurs, dont 60% sur les aspects technologiques, Business Interactif accompagne ses clients sur des projets stratégiques en France et à l'étranger (Bureaux à Paris, Lyon, New-York, Tokyo et Rio).

Deux références symbolisent à elles seules l'étendue du savoir-faire de Business Interactif : OOSHOP.COM d'un côté (réalisation de l'ensemble du projet, tant sur les aspects Front que Middle et Back-Office), LANCOME.COM de l'autre. Que ce soit pour des projets à haute technicité ou nécessitant une valeur ajoutée marketing et créatrice hors-pair, Business Interactif s'est peu à peu imposé comme l'un des acteurs clé sur le marché de l'e-business. Pour plus de renseignements, nous vous invitons à consulter notre site : www.businessinteractif.fr

Pourquoi un white paper sur les méthodes agiles ?

Au cours des huit dernières années, nos équipes ont contribué à la réussite de grands projets E-business, dans une logique de maîtrise d'œuvre. L'une des étapes clé dans le développement de notre société a été la mise en œuvre d'un Plan Assurance Qualité permettant d'offrir à nos clients un haut niveau de service. La gestion d'un projet et les méthodes de développement optimales sont des préoccupations récurrentes dans notre métier. Nos grands projets sont réalisés en s'appuyant sur des méthodes éprouvées, tirées notamment de Merise pour la conception des modèles de données, de RAD pour la gestion de projet et le maquettage, d'UML pour la conception objets/composants. Nous avons au fil des années tiré le meilleur de ces différentes approches pour les adapter au contexte des projets e-business.

La montée en puissance de méthodes dites «nouvelles », les méthodes agiles, ne pouvait nous laisser insensibles. A ce titre, comme nous l'avions fait pour le Knowledge Management, il nous a paru intéressant de faire partager par le biais d'un White Paper notre vision de ce sujet : cartographie des méthodes, mais aussi retours d'expérience !

Jean-Louis Bénard, directeur technique de Business Interactif.

2. LES METHODOLOGIES AGILES : GENERALITES

2.1 CONTEXTE

Les méthodes de gestion de projet informatique connaissent, au même titre que les technologies mises en œuvre, une remise en cause permanente.

La proportion importante d'échec des projets vient souvent alimenter des réactions plus ou moins constructives aux méthodes mises en œuvre. Les évolutions des architectures technologiques et des outils de développement y contribuent également pour part importante.

Ainsi, UML et Unified Process sont venues en réaction aux méthodes dites « merisiennes » (cycle en V, etc.) à la fois poussées par les technologies objet et les insuffisances de méthodes préconisant des cycles très longs.

Les méthodes agiles n'échappent pas à la règle. Mais elles s'inscrivent dans une démarche plus radicale, que l'on pourrait résumer par « trop de méthode tue la méthode ».

De manière générale, leur but est d'augmenter le niveau de satisfaction des clients tout en rendant le travail de développement plus facile. Mais les véritables fondements des méthodes agiles résident plus précisément dans deux caractéristiques :

- Les méthodes agiles sont "adaptatives" plutôt que prédictives

Si les méthodes lourdes tentent d'établir dès le début d'un projet un planning à la fois très précis et très détaillé du développement sur une longue période de temps, cela suppose que les exigences et spécifications de base restent immuables. Or, dans tous les projets, les exigences changent au cours du temps et le contexte évolue aussi. Par conséquent, si elles fonctionnent sur des projets courts et de petite taille, ces méthodologies sont trop réfractaires au changement pour pourvoir être appliquées à des projets plus importants. A l'inverse, les méthodes agiles se proposent de réserver un accueil favorable au changement : ce sont des méthodes itératives à planification souple qui leur permettent de s'adapter à la fois aux changements de contexte et de spécifications du projet.

- Les méthodes agiles sont orientées vers les personnes plutôt que vers les processus

Les méthodes agiles s'efforcent de travailler avec les spécificités de chacun plutôt que contre la nature de chacun pour que le développement soit une activité plaisante où chacun se voit confier une part de responsabilité.

Nées en réaction aux méthodes traditionnelles, il existe une grande diversité de méthodes auxquelles ont peut donner le qualificatif d'agile. Avant de passer en revue les spécificités de chacune parmi les principales d'entre elles dans une seconde partie, essayons tout d'abord de dégager les principales caractéristiques communes à toutes ces méthodes, autrement dit : qu'est ce qui fait qu'une méthode de développement est une méthode agile ?

2.2 CARACTERISTIQUES COMMUNES DES METHODES AGILES

Très récemment, en février 2001, les instigateurs des principales méthodes agiles se sont réunis pour former l' "Agile Alliance" (http://agilealliance.org). Ensemble, ils ont pu dégager des principes fondamentaux sur lesquels

s'appuyer pour que le développement puisse se faire rapidement et s'adapter au changement.

Leur travail a abouti au "Manifeste pour le développement agile d'applications" :

Manifesto for Agile Software Development

We are uncovering better ways of developing
software by doing it and helping others do it.
Through this work we have come to value:

Individuals and interactions over processes and tools
Working software over comprehensive documentation
Customer collaboration over contract negotiation
Responding to change over following a plan

That is, while there is value in the items on
the right, we value the items on the left more.

Ce manifeste, comme on peut le voir, comporte quatre valeurs principales détaillées ci-dessous.

Les paragraphes suivants ont pour objet de présenter les méthodes agiles, telles qu'elles ont été conçues par les membres de l'Agile Alliance, ***ils ne traduisent pas spécifiquement la position de Business Interactif sur le sujet***.

2.2.1 Priorité aux personnes et aux interactions sur les procédures et les outils

La meilleure garantie de succès réside dans les personnes : même une bonne méthode, de bonnes procédures ne sauveront pas un projet de l'échec si l'équipe n'est pas constituée de personnes adéquates et ouvertes. Par contre, une mauvaise méthode peut conduire une équipe parfaite à l'échec. Pour constituer une équipe, pas besoin de génies du développement : un programmeur moyen, mais doué de capacités à travailler en groupe et à communiquer est un bien meilleur équipier.

La gestion de projet a longtemps établi la «méthode » comme le vecteur essentiel de succès d'un projet. Cette méthode a souvent pris la forme de procédures. L'importance accordée aux outils supports de la gestion de projets s'est également renforcée au détriment de l'individu.

Utiliser les outils appropriés (compilateurs, environnements de développement, outils de modélisation, etc.) peut être décisif. En revanche, l'utilisation d'outils surdimensionnés ou faisant de l'homme un simple instrument de la démarche est aussi néfaste que le manque d'outils appropriés.

Opposés à une approche trop outillée, les membres de l'AgileAlliance affirment que la meilleure garantie de succès réside dans les personnes : même une bonne méthode, de bonnes procédures ne sauveront pas un projet de

l'échec si l'équipe n'est pas constituée de personnes très impliquées dans le projet. Pour constituer une équipe, pas besoin de génies du développement : un programmeur moyen, mais doué de capacités à travailler en groupe et à communiquer est un bien meilleur équipier. Ce sont avant tout les interactions, les initiatives et la communication interpersonnelles qui feront le succès.

2.2.2 Priorité aux applications fonctionnelles sur une documentation pléthorique

Les partisans des méthodes agiles affirment la légitimité d'une certaine forme de documentation mais ils dénoncent les effets pervers d'une documentation pléthorique. En effet, l'écriture et le maintien à niveau de la documentation sont extrêmement consommateurs de ressources. Un document qui n'est pas mis à jour régulièrement devient très rapidement totalement inutile, voire même trompeur. Le manifeste est donc favorable aux documentations succinctes, ne décrivant que les grandes lignes de l'architecture du système mais régulièrement tenues à jour, et d'une documentation permanente du code lui-même. Le meilleur transfert des connaissances sur le système s'effectue de toute manière par la participation au travail de l'équipe.

2.2.3 Priorité de la collaboration avec le client sur la négociation de contrat

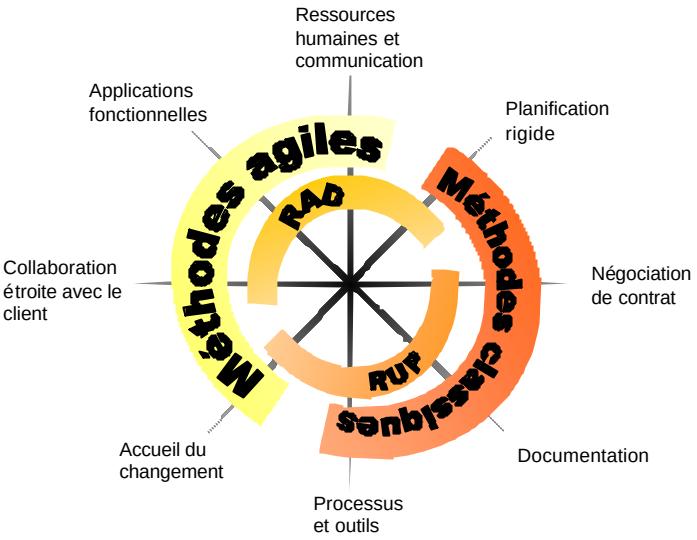
Le succès d'un projet requiert un feedback régulier et fréquent de la part du client. Un contrat qui spécifie les exigences, le planning et le coût d'un projet a priori relève d'une vision utopique d'une projet informatique. La meilleure manière de procéder pour le client est de travailler en étroite collaboration avec l'équipe de développement, pour lui fournir un feedback continu qui assure un meilleur contrôle du projet. Ainsi, des modifications de spécifications peuvent intervenir très tard dans le cycle de développement du projet. C'est en définitive une solution répondant réellement aux attentes du client qui est réalisée et non une solution répondant aux exigences d'un contrat établi a priori. Nous le verrons en synthèse, ce point ne reste pas simple à mettre en œuvre. Cela nécessite bien sûr une grande maturité du client et du prestataire de service afin d'établir une réelle relation de confiance, ainsi qu'une bonne compréhension de la réalité opérationnelle du projet par les juristes en charge du dossier.

2.2.4 Priorité de l'acceptation du changement sur la planification

C'est la capacité à accepter le changement qui fait bien souvent la réussite ou l'échec d'un projet. Lors de la planification, il est donc nécessaire de veiller à ce que le planning soit flexible et adaptable aux changements qui peuvent intervenir dans le contexte, les technologies et les spécifications.

En effet il est très difficile de penser dès le début à toutes les fonctionnalités dont on aimerait disposer et il est très probable que le client modifie ses exigences une fois qu'il aura vu fonctionner une première version du système.

Figure 1 - Positionnement des méthodes par rapport aux quatre critères agiles



Les quatre valeurs ci-dessus sont en fait déclinées sur douze principes plus généraux qui caractérisent en détail les méthodes agiles :

- "Notre priorité est de satisfaire le client en lui livrant très tôt et régulièrement des versions fonctionnelles de l'application source de valeur"

Les méthodes agiles recommandent de livrer très tôt, dans les premières semaines si possible une version rudimentaire de l'application puis de livrer souvent des versions auxquelles les fonctionnalités s'ajoutent progressivement. De cette manière, le client peut décider à tout moment la mise en production de l'application, dès qu'il la considère comme assez fonctionnelle. A chaque version (release), un feedback de la part du client est nécessaire pour permettre soit de continuer le développement comme prévu, soit d'opérer des changements. De cette manière, les changements dans les spécifications interviennent tôt dans le processus de développement et sont moins problématiques.

- "Accueillir le changement à bras ouverts, même tard dans le processus de développement. Les méthodologies agiles exploitent les changements pour apporter au client un avantage concurrentiel"

Ceci est un état d'esprit nécessaire : tout changement des exigences doit être perçu comme une bonne chose car cela signifie que l'équipe a compris et appris comment satisfaire encore mieux la demande.

Le but des méthodes agiles est de produire des systèmes très flexibles, de sorte que l'impact sur le système d'une évolution des spécification reste minimal.

- "Livrer le plus souvent possible des versions opérationnelles de l'application, avec une fréquence comprise entre deux semaines et deux mois"

Ne pas se contenter de livrer des liasses de documents décrivant l'application : il faut constamment garder pour objectif de livrer le plus rapidement possible au client une solution qui satisfasse ses besoins.

- "Clients et développeurs doivent coopérer quotidiennement tout au long du projet"

Pour qu'un projet puisse être considéré comme agile, il faut qu'il y ait une interaction permanente entre le client, les développeurs : c'est ce qui guide continuellement le projet.

- "Construire des projets autour d'individus motivés. Leur donner l'environnement et le support dont ils ont besoin et leur faire confiance pour remplir leur mission"

Dans un projet agile, les personnes sont considérées comme le facteur clé de succès. Tous les autres facteurs, processus, environnement, management sont susceptibles d'être changés s'ils s'avèrent être une entrave au bon fonctionnement de l'équipe.

- "La méthode la plus efficace de communiquer des informations à une équipe et à l'intérieur de celle-ci reste la conversation en face à face"

Le mode de communication par défaut au sein d'une équipe agile est la conversation et non l'écrit. Des documents peuvent bien sûr être rédigés mais ils n'ont en aucun cas pour but de consigner par écrit la totalité des informations relatives au projet. Même les spécifications peuvent très bien ne pas être écrites de manière formelle.

- "Le fonctionnement de l'application est le premier indicateur d'avancement du projet"

Contrairement à d'autres méthodes, l'avancement du projet ne se mesure pas à la quantité de documentation rédigée ni en terme de phase dans laquelle il se trouve mais bien en terme de pourcentage de fonctionnalités effectivement mises en place.

- "Les méthodes agiles recommandent que le projet avance à un rythme soutenable : développeurs et utilisateurs devraient pouvoir maintenir un rythme constant indéfiniment"

Il ne s'agit pas de sprinter sur 100 mètres, mais plutôt de tenir la distance sur un marathon : l'équipe se doit donc d'adapter son rythme pour préserver la qualité de son travail sur toute la durée du projet.

- "Porter une attention continue à l'excellence technique et à la conception améliore l'agilité"

La meilleure façon de développer rapidement est de maintenir le code source de l'application aussi propre et robuste que possible. Les membres de l'équipe doivent donc s'efforcer de produire le code le plus clair et le plus propre possible. Ils sont invités à nettoyer chaque jour le code qu'ils ont écrit.

- "La simplicité – art de maximiser la quantité de travail à ne pas faire – est essentielle"

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

- "Les meilleures architectures, spécifications et conceptions sont le fruit d'équipes qui s'auto organisent"

Les responsabilités ne sont pas confiées à quelqu'un en particulier mais à l'équipe dans son intégralité. Les membres de l'équipe prennent ensuite leurs responsabilités et se partagent les tâches sur le principe du volontariat.

- "A intervalles de temps réguliers, l'ensemble de l'équipe s'interroge sur la manière de devenir encore plus efficace, puis ajuste son comportement en conséquence"

Une équipe agile est consciente que son environnement est en perpétuelle évolution. C'est pourquoi elle ajuste continuellement son organisation, ses règles, son fonctionnement, etc. de manière à rester agile.

2.3 DES METHODES RAD (RAPID APPLICATION DEVELOPMENT) AUX METHODES AGILES, UNE FILIATION ?

2.3.1 Origines du RAD

Le RAD (Rapid Application Development) est né dans les années 80 à la suite d'un double constat. D'une part, le manque de concertation entre les informaticiens en charge de la réalisation d'un projet et les utilisateurs conduit souvent à la réalisation d'applications mal adaptées aux besoins. D'autre part les méthodes classiques de conduite de projet sont inadaptées à la vitesse des évolutions technologiques et la durée des projets est beaucoup trop longue.

Les objectifs premiers du RAD sont de conduire à l'amélioration de la qualité des développements tout en réduisant les délais et en facilitant la maîtrise des coûts. Pour cela, il est nécessaire d'associer méthodologie et relations humaines. Le RAD est fondamentalement une méthode basée sur la communication.

Les concepts du RAD ont été définis par James Martin avant d'être repris et adaptés au contexte français par Jean Pierre Vickoff dès 1989.

A ses début, le RAD a souvent été qualifié de "chaotique" et considéré comme une "absence de méthode" plutôt que comme une méthode. Si le RAD a souffert d'une si mauvaise image, c'est avant tout lié à la notion de rapidité qu'on a souvent tendance à considérer, à tort, comme incompatible avec celle de qualité. Pour éviter ce malentendu, Jean Pierre Vickoff propose de traduire RAD par "développement maîtrisé d'applications de qualité approuvée par les utilisateurs".

2.3.2 Les acteurs du RAD

Dans un projet RAD, la répartition des rôles est très structurée. Comme dans une approche classique, l'ensemble de l'organisation s'appuie sur un principe fondamental : la séparation des rôles opérationnels et des responsabilités entre d'une part la maîtrise d'ouvrage (MOA) qui représente l'utilisateur et à ce titre détermine les fonctionnalités à développer et leur degré de priorité et

d'autre part la maîtrise d'œuvre (MOE) qui apporte les solutions techniques aux problèmes posés par la maîtrise d'ouvrage. (Cette séparation des rôles est encore renforcée par la présence du Groupe d'animation et de Rapport RAD qui se charge d'organiser la communication du projet. Son rôle principal est de faciliter l'expression des exigences et de les formaliser en temps réel.

La maîtrise d'ouvrage

La maîtrise d'ouvrage a trois responsabilités principales :

- Définir les objectifs et les exigences du système : le maître d'ouvrage, qui préside le comité de pilotage est responsable de la rédaction de documents de cadrage dans lesquels sont explicités les objectifs d'ordre stratégique, fonctionnel, technologique, organisationnel, ainsi que les contraintes de projet. Ces objectifs, classés par ordre de priorité servent de base pour la prise de décisions de pilotage. Le maître d'ouvrage spécifie aussi les exigences en déléguant leur mise en forme à diverses personnes compétentes : par exemple, les exigences fonctionnelles sont exprimées par des représentants d'utilisateurs et des experts fonctionnels.
- Valider les solutions proposées et élaborées : en collaboration avec les personnes qui ont exprimé les exigences, le maître d'ouvrage vérifie que la ou les solutions proposées par la maîtrise d'œuvre permettent bien de satisfaire ces exigences.
- Préparer et piloter le changement induit : il s'agit là d'opérations de sensibilisation, formation, communication ainsi que d'organisation.

Au sein de la maîtrise d'ouvrage, on peut distinguer trois acteurs principaux :

- Maître d'ouvrage : prend des décisions sur les objectifs (produit, coût, délai) et les orientations du projet et décide *in fine* l'expression des exigences et la validation des solutions ainsi que les moyens à mettre en œuvre au sein de la maîtrise d'ouvrage.
- Coordinateur de Projet Utilisateurs ou Maître d'Ouvrage délégué : coordonne et valide les activités de la maîtrise d'ouvrage. Réalise le suivi des objectifs (produits, coûts / charges, délais) et des activités de la maîtrise d'ouvrage.
- Responsable de la cohérence et de la qualité fonctionnelle : supervise l'expression des exigences et de la validation des solutions, contrôle la cohérence des décisions prises dans les domaines fonctionnels. Supervise la vérification de la qualité fonctionnelle (point de vue utilisateurs) des solutions proposées et élaborées.

La maîtrise d'œuvre

Elle a également trois responsabilités principales :

- Proposer et réaliser la solution : cela passe notamment par la définition et la mise en œuvre du planning et des moyens humains et logistiques nécessaires
- Livrer des "fonctionnalités" : on entend par fonctionnalités les produits finis mais aussi des produits intermédiaires et diverses informations qui

permettront à la maîtrise d'ouvrage de préparer et de piloter le changement. Les dates de fourniture de ces différents livrables figurent au planning.

- Respecter les directives du Plan d'Assurance Qualité : en même temps que la réalisation de l'application, la maîtrise d'œuvre réalise son suivi sur des critères tels que les fonctionnalités, la qualité, le planning, les coûts, la visibilité.

Au sein de la maîtrise d'œuvre, on peut distinguer trois acteurs principaux :

- Maître d'œuvre : propose des objectifs (coût, délai en particulier) et des orientations pour le projet. Décide *in fine* les propositions de solution et les solutions elles-mêmes qui seront élaborées. Décide des moyens et méthodes à mettre en œuvre au sein de la MOE. Se coordonne avec les MOE des autres projets et les sociétés extérieures participant à la MOE du projet
- Pilote de Projet Informatique : coordonne les travaux de la MOE. Valide les propositions de solution et les solutions elles-mêmes élaborées par la MOE. Estime et met en œuvre les moyens MOE (planning de production, méthodes et outils)
- Responsable par domaine : pilote et suit l'équipe de production d'un domaine.

Le Groupe d'animation et de Rapport RAD

Le groupe d'animation et de rapport (GAR) a pour responsabilités la préparation des réunions (convocation, logistique et suivi), l'organisation des séances de travail, la formalisation des exigences exprimées, la gestion des comptes-rendus, la planification et l'animation des focus.

L'animateur doit adopter une attitude directive sur la forme et non directive sur le fond. Il se garde d'émettre un avis personnel, de porter un jugement et de favoriser des opinions. Il n'ajoute rien au discours des participants et se contente de maintenir les discussions bien centrées sur le thème voulu et de canaliser les entretiens. Pour cela, il peut procéder par exemple par synthèse (résumer l'essentiel en le reformulant) ou par élucidation (forcer les gens, par le biais de questions, à aller jusqu'au bout d'une idée).

Le rapporteur doit être capable de produire une documentation automatisée et de la maintenir à jour. C'est tout d'abord un analyste concepteur, spécialement formé aux AGL (Ateliers de génie Logiciel). Lors des réunions portant sur la conception du système, il doit être capable de le modéliser en direct. Il est aussi responsable de la synthèse des comptes-rendus de réunion.

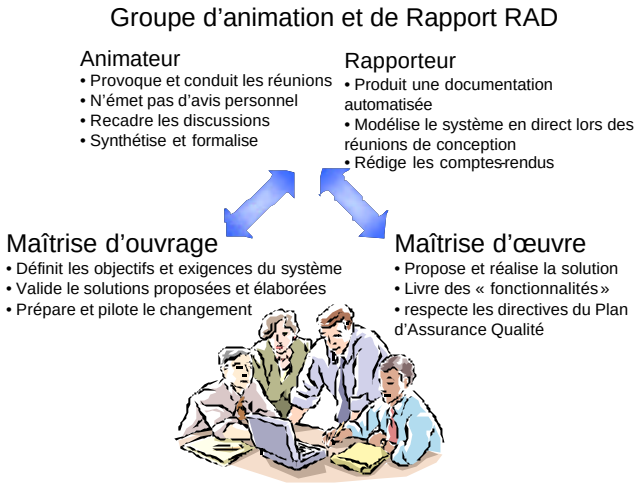


Figure 2 - Les acteurs du RAD

2.3.3 Les cinq phases d'un projet RAD

Initialisation (préparation de l'organisation et communication)

Cette phase, qui représente environ 5% du projet, définit le périmètre général du projet, établit la structure du travail par thèmes, recense les acteurs pertinents et amorce la dynamique du projet.

Cadrage (analyse et expression des exigences)

C'est aux utilisateurs de spécifier leurs exigences et d'exprimer leurs besoins lors d'entretiens de groupe. Il est généralement prévu de 2 à 5 jours de sessions par commission (thème). Cette phase représente environ 10% du projet.

Design (conception et modélisation)

Les utilisateurs participent à l'affinage et à la validation des modèles organisationnels : flux, traitements, données. Ils valident également un premier prototype ayant pour but de présenter l'ergonomie et la cinématique générale de l'application. 4 à 8 jours de sessions sont prévus par commission. Cette phase représente environ 25% du projet.

Construction (réalisation, prototypage)

Durant cette phase, l'équipe RAD (SWAT) construit au cours de plusieurs sessions itératives l'application module par module. L'utilisateur participe toujours activement aux spécifications détaillées et à la validation des prototypes. Cette phase représente environ 50% du projet.

Finalisation (recette et déploiement)

Cette phase, qui représente environ 10% du projet permet une livraison globale et le transfert du système en exploitation et maintenance.

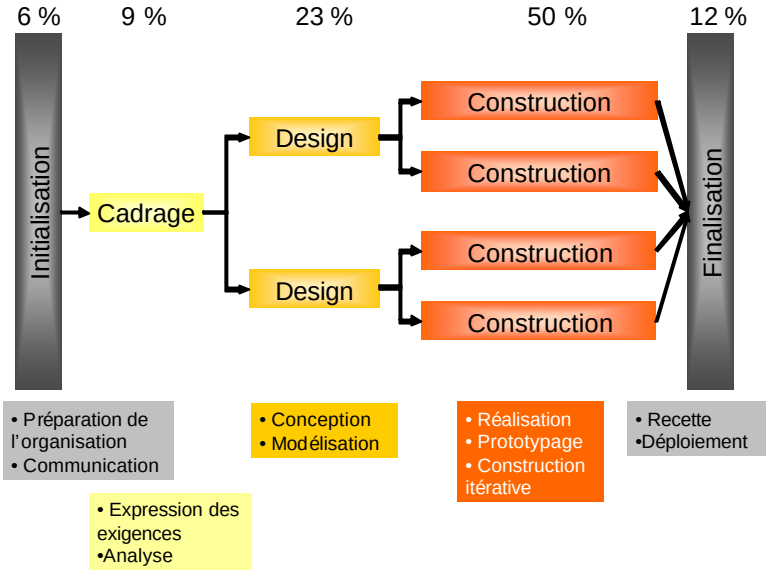


Figure 3 - Cycle de vie d'un projet RAD

2.3.4 RAD et modélisation

S'il se veut efficace, le RAD se doit d'être rapide et donc d'utiliser des techniques de modélisation rigoureuses mais simplifiées. Le RAD ne préconise pas de technique de modélisation particulière : il est tout à fait possible d'utiliser des modèles objets comme UML ou des modèles de Merise, à condition de les alléger pour n'en conserver que ce qui est strictement indispensable.

Dans le cadre d'une approche "classique" par le haut de type merisienne, tous les modèle d'abstraction merisiens ne sont pas utilisés. Dans le cas général, ne sont conservés que le modèle de Contexte (MC), le modèle Conceptuel de Communication (MCC), la hiérarchie de fonctions (ou MCT), le modèle Conceptuel de Données (MCD), et le modèle Organisationnel de Traitements (MOT).

Dans le cas d'un développement orienté objet, il est préconisé d'employer la notation ULM. Les principaux modèles sont les suivants : modèle de Classes, modèle d'Etat, modèle des Cas d'utilisation, modèle des Interactions, modèle de Réalisation, modèle de Déploiement.

2.3.5 Pourquoi le RAD n'est il pas à proprement parler une méthode "agile" ?

Essayons d'examiner la méthode RAD à la lumière des quatre critères qui font qu'une méthode est agile ou ne l'est pas.

Le RAD met bien en avant les ressources humaines et la communication par rapport au côté procédurier, notamment grâce à l'existence du groupe d'animation et de rapport qui a pour rôle de faciliter la communication entre la maîtrise d'œuvre et la maîtrise d'ouvrage.

Le RAD privilégie la production d'applications fonctionnelles par rapport à l'écriture massive de documentation.

Le RAD implique une collaboration étroite avec le client. Dans un processus RAD, le client est un acteur clé (maîtrise d'ouvrage).

Par contre, le dernier critère des méthodes agiles n'est pas véritablement rempli par le RAD. En effet, un projet RAD n'est pas conçu pour faire preuve d'une grande flexibilité par rapport au changement. Au contraire, le RAD se place plutôt du côté de la planification rigide.

Le RAD présente donc trois des quatre caractéristiques des méthodes agiles. C'est donc une méthode proche des méthodes agiles sans pour autant en faire véritablement partie.

2.4 DE UNIFIED PROCESS (UP & RUP) A L'AGILITE, UNE FILIATION ?

2.4.1 Les 5 principes d'UP

UP est une méthode générique de développement de logiciels. Cette méthode nécessite donc d'être adaptée à chacun des projets pour lesquels elle sera employée.

UP présente sept caractéristiques essentielles, les trois premières étant trois critères préconisés par UML (Unified Modeling Language) :

UP est pilotée par les cas d'utilisation

Les cas d'utilisation (« Use Cases ») sont les véritables pilotes du projet, quelle que soit l'activité et quelle que soit la phase. En effet, le système est tout d'abord analysé, conçu et développé pour des utilisateurs. Le système ne peut pas être développé par des informaticiens n'ayant aucune idée de ce que sont le métier et l'environnement des utilisateurs : tout doit donc être fait en adoptant le point de vue utilisateur. Les cas d'utilisation sont l'outil de modélisation des besoins en termes de fonctionnalités : c'est sous cette forme que sont exprimées les exigences. Les cas d'utilisation servent aussi de base pour l'analyse, le modèle logique, ainsi que de base pour les tests fonctionnels.

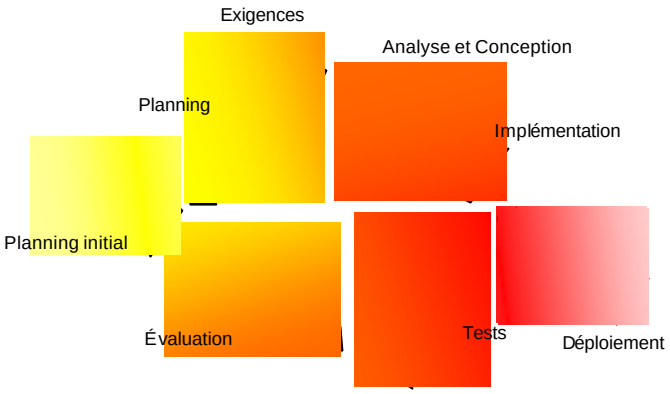
UP est centrée sur l'architecture

UP se soucie très tôt de l'architecture dans le cycle de vie d'une application. L'architecture du système est décrite à l'aide de différentes vues. Alors que les modèles cas d'utilisation, analyse, conception, implémentation, déploiement, sont complets et exhaustifs, les vues définies par l'architecte ne reprennent que les éléments significatifs. L'architecte procède de manière incrémentale : il commence par définir une architecture simplifiée qui répond aux besoins classés comme prioritaires avant de définir à partir de là les sous-systèmes de manière beaucoup plus précise.

UP est itérative et incrémentale

UP procède par itérations par opposition aux méthodes en cascade qui sont plus coûteuses et ne permettent pas une bonne gestion du risque. De plus, il est toujours difficile de corriger à la fin une erreur commise au début d'un projet. En procédant de manière itérative, il est possible de découvrir les erreurs et les incompréhensions plus tôt. Le feedback de l'utilisateur est aussi encouragé et les tests effectués à chaque utilisation permettent d'avoir une vision plus objective de l'avancement du projet. Enfin, le travail itératif permet à l'équipe de capitaliser à chaque cycle les enseignements du cycle précédent.

Figure 4 - UP est une méthode itérative et incrémentale



UP gère les besoins et les exigences

Les exigences du client changent au cours du projet. UP recommande de découvrir les besoins, de les organiser et de les documenter de manière à avoir une approche disciplinée de l'étude des exigences et de pouvoir les filtrer, les trier par propriété et réaliser leur suivi pour pouvoir estimer de manière objective les fonctionnalités et les performances du système à développer.

UP est fondée sur la production de composants

UP recommande de structurer l'architecture à base de composants (COM, CORBA, EJB, etc.). Les architectures ainsi construites sont de fait plus robustes et modulaires. De plus, cette approche permet la réutilisation de composants existants si ceux-ci ont été développés de manière assez générique. Enfin, la visualisation à l'aide des outils de modélisation est facile et automatique.

UP pratique la modélisation visuelle

UP préconise la modélisation visuelle du système. Un modèle est une simplification de la réalité qui décrit le système selon un certain point de vue. La modélisation sous différents points de vue permet une meilleure compréhension et une compréhension globale du système à concevoir. L'utilisation d'ou-

tils visuels augmente encore les capacités de l'équipe à gérer la complexité des systèmes.

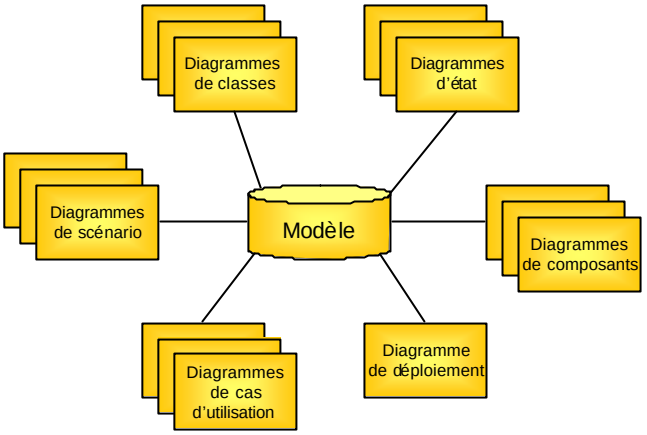
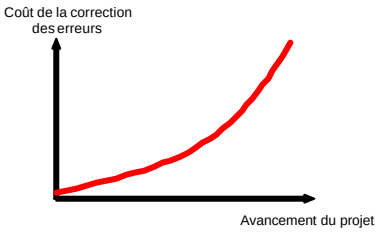


Figure 5 - Modélisation visuelle selon différentes vues

UP se soucie en permanence de la qualité



Partant du constat que le coût de la correction des erreurs augmente exponentiellement au fur et à mesure de l'avancement du projet, il est absolument nécessaire d'avoir un contrôle permanent et rigoureux des fonctionnalités, de la fiabilité, des performances de l'application et des performances du système. Pour cela, une automatisation des tests tout au long du cycle de vie s'impose. Ainsi, les défauts sont détectés très rapidement et le coût de leur correction est minimisé, l'estimation de l'état d'avancement du projet est plus objective et la qualité et les performances des parties "à risque" est améliorée.

UP gère les risques de façon systématique et permanente

La phase de pré-étude a pour but d'identifier les risques qui pourraient mettre le projet en péril. Tout au long du projet, il est recommandé de maintenir une liste de risques, issus de la phase de pré-étude, classés selon leur danger pour l'équipe afin d'avoir une vue plus explicite des choses.

2.4.2 Les activités dans la méthode UP

UP définit les activités essentielles et propose, pour chacune d'entre elles, une liste d'intervenants (architecte, analyste...) et une liste de travaux associés.

Expression des besoins

UP distingue deux types de modèles : "Domain Model" et "Business Model". Le modèle de domaine ne dépend pas de l'application : il a pour but d'apprendre un domaine, un métier jusque là méconnu. Le modèle "business" est plutôt une modélisation des procédures en vigueur dans l'entreprise et s'intéresse directement aux collaborateur et à leurs divers travaux. C'est à partir de ce modèle que peut alors démarrer l'analyse des cas d'utilisation du système. Il s'agit de mieux connaître le domaine et ses activités avant de s'intéresser aux fonctionnalités du système par une étude par les cas d'utilisation.

Analyse

Comprendre et de structurer le logiciel à développer sont les objectifs de l'analyse. On construit dans cette activité une représentation interne quasi-idéale du système, sans trop tenir comte des exigences et contraintes de conception. L'analyse utilise le langage du développeur alors que l'expression des besoins se fait du point de vue utilisateur.

Conception

La conception consiste à définir l'architecture du système. La conception n'est non pas une phase du process UP mais une activité qui trouve sa place dans toutes les phases. La conception est ainsi réalisée de manière incrémentale. Dans une phase de pré-étude elle consiste par exemple à maquetter l'architecture, tandis qu'en phase de construction la conception de grossière devient beaucoup plus détaillée. Si l'analyse est une vue purement logique, la conception est plus physique et se soucie des contraintes que l'analyse avait laissées de côté.

Implémentation et Test

Cette activité consiste à créer à proprement parler les divers composants : sources, scripts, puis exécutables... Les tests se basent sur les cas d'utilisation.

2.4.3 Les phases du cycle de vie

Etude d'opportunité

C'est durant cette phase qu'il faut se poser la question de la faisabilité du projet, des frontières du système, des risques majeurs qui pourraient mettre en péril le projet. A la fin de cette phase, est établi un document donnant une vision globale des principales exigences, des fonctionnalités clés et des contraintes majeures. Environ 10 % des cas d'utilisation sont connus à l'issue de cette phase. Il convient aussi d'établir une estimation initiale des risques, un "Project Plan", un "Business Model".

Elaboration

Cette phase a pour but de préciser l'architecture. A ce stade, 80 % des cas d'utilisation sont obtenus. Cette phase permet aussi de connaître des exi-

gences supplémentaires, en particulier des exigences non fonctionnelles, de décrire l'architecture du système, de le prototyper, de réviser la liste des risques et de mettre en place un plan de développement. En théorie, c'est à ce stade qu'on est à même de faire une offre de réalisation du système.

Construction

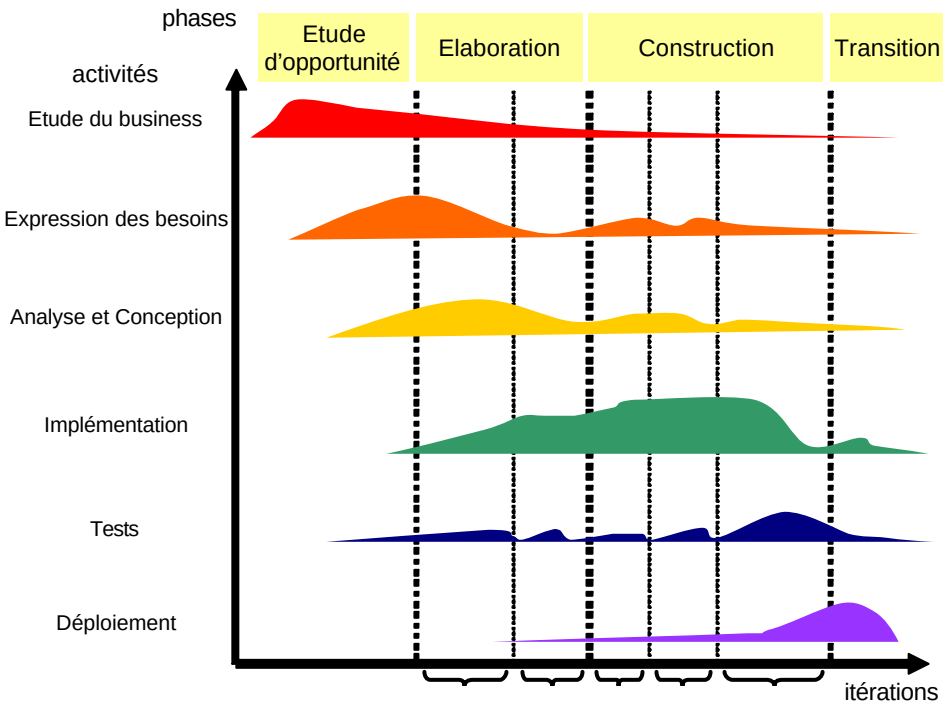
L'objectif est de fournir une version beta de la release en cours de développement ainsi qu'une version du manuel utilisateur.

Transition

Cette phase a pour objectif de peaufiner le système et de corriger les derniers bugs pour pouvoir passer de la version beta au déploiement sur l'ensemble des sites. Comme le nom de la phase l'indique, il s'agit aussi de préparer la release suivante et de boucler le cycle soit sur une nouvelle étude d'opportunité soit une élaboration ou construction.

Figure 6 - Importance des activités au cours des différentes phases

D'après Rational Software Corporation



2.4.4 Pourquoi UP n'est pas à proprement parler une méthode agile ?

Comme nous l'avons fait pour le RAD, positionnons UP par rapport aux quatre critères d'agilité.

UP s'efforce de produire des applications fonctionnelles et surtout en coïncidence avec les besoins exprimés par le client. Cette caractéristique vient du fait qu' UP est pilotée par les cas d'utilisation.

UP insiste sur la nécessité d'une collaboration étroite avec le client. L'expression des besoins se fait en début de projet, mais à chaque itération, la phase de transition permet un retour du client sur ce qui est fait.

UP fait aussi un premier pas vers des méthodes souples par rapport au changement.

En revanche, la méthode UP reste très procédurière. De plus UP s'ancre très fortement sur la modélisation UML et ses outils. Cette modélisation parfois qualifiée de "modélisation à outrance" a tendance à alourdir quelque peu la méthode.

En conséquence, UP ne remplit pas pleinement les critères d'agilité tout en restant très proche d'une méthode agile.

2.5 LA SITUATION EN FRANCE : UN FORT ANCRAGE DE LA METHODE MERISE ET DE LA MODELISATION ENTITE-RELATION

La conception d'un système passe par une phase d'analyse qui nécessite des méthodes permettant de mettre en place un modèle du système à développer. Parmi les méthodes d'analyse, Merise, qui date de la fin des années soixante-dix, reste très fortement implantée en France.

La méthode Merise est fondée sur la séparation des données et des traitements à effectuer en plusieurs modèles conceptuels, logiques et physiques. La séparation des données et des traitements a pour but de donner une certaine longévité au modèle. En effet, la structure des données n'a pas à être souvent modifiée dans le temps, tandis que les traitements le sont plus fréquemment. La méthode Merise est apparue dans les années 1978-1979, à la suite d'une consultation nationale lancée en 1977 par le ministère de l'Industrie. Cette consultation avait pour but de choisir des sociétés de conseil en informatique chargées de définir une méthode de conception de systèmes d'information. Les deux principales sociétés ayant mis au point cette méthode sont le CTI (Centre Technique d'Informatique), et le CETE (Centre d'Etudes Techniques de l'Equipement).

La démarche préconisée par Merise est la suivante :

- Schéma directeur : permet de définir les objectifs du projet
- Étude préalable : il s'agit de référencer les moyens existants, de déterminer les limites du système existant. Sur la base des besoins futurs, plusieurs scénarios sont proposés. A la fin de cette phase d'étude, un seul scénario est retenu sur des critères de coûts, limites, impacts, délais et faisabilité)
- Analyse détaillée : le premier modèle à établir est le Modèle Conceptuel des Données (MCD) basé sur une modélisation entités / relations. Au cours de cette phase, on établit aussi le Modèle Organisationnel des Traitements (MOT) puis le Modèle Logique des Données (MLD).
- Analyse technique : il s'agit de définir les solutions techniques retenues en établissant le Modèle Physique des Données (MPD) et le Modèle Opérationnel des Traitements (MOPT)

Merise permet de modéliser un système selon différents degrés d'abstraction : le modèle conceptuel est théorique et ne se soucie pas ou peu de l'implémentation, le modèle logique représente un choix logiciel et le modèle physique représente un choix d'architecture.

Merise, méthode vieille d'une vingtaine d'années reste particulièrement bien ancrée en France, notamment pour la modélisation des données. On a donc une situation un peu particulière dans l'hexagone : les méthodologies plus récentes sont en général modifiées pour pouvoir s'adapter à l'existant Merise encore très utilisé pour ses performances en matière de modélisation des données.

3. PRINCIPALES METHODOLOGIES AGILES, ETAT DES LIEUX, COMPARAISON

L'agilité comprend plusieurs courants de pensée qui ont conduit à des méthodologies reposant sur les mêmes concepts mais présentant chacune des singularités. Présentant successivement Xtreme Programming, DSDM, ASD, CRYSTAL, SCRUM, FDD, cette partie se conclut par une vision de synthèse sous la forme d'une comparaison.

3.1 EXTREME PROGRAMMING (XP)

3.1.1 Aux origines d'eXtreme Programming

La suppression des risques existants sur un projet représente la philosophie d'XP. . En introduisant cette méthode, ses créateurs veulent en finir avec la dérive de délais, l'annulation des projets, la non-qualité, la mauvaise compréhension du métier du client et l'impossibilité de suivre les changements.

XP a été mis en œuvre pour la première fois en 1996 sur le projet C3, Chrysler Comprehensive Compensation System, qui consistait à mettre en place un nouveau système de gestion de la paie des dix mille salariés du fabricant automobile. Sur ce projet, la méthode XP a été adoptée pour tenter de sauver le projet d'une dérive annoncée.

Les pères de la méthode, Ward Cunningham et Kent Beck définissent eXtreme Programming comme "une méthode basée sur des pratiques qui sont autant de boutons de contrôle poussés au maximum".

3.1.2 Les 4 valeurs d'eXtreme Programming

XP met en avant quatre valeurs prenant en considération à la fois les enjeux commerciaux et les aspects humains des projets de développement d'applications.

Communication

L'absence de communication est certainement l'un des défauts les plus graves qui mettent en péril un projet. Diverses pratiques XP tendent à rendre la communication omniprésente entre tous les intervenants : entre développeurs (programmation en binôme), entre développeurs et managers (tests, estimations), entre développeurs et clients (tests, spécifications).

Toutes ces pratiques qui forcent à communiquer ont pour but de permettre à chacun de se poser les bonnes questions et de partager l'information.

Simplicité

Cette valeur de simplicité repose sur le pari qu'il coûte moins cher de développer un système simple aujourd'hui quitte à devoir engager de nouveaux frais plus tard pour rajouter des fonctionnalités supplémentaires plutôt que de concevoir dès le départ un système très compliqué dont on risque de n'avoir plus besoin dans un avenir proche. XP encourage donc à toujours s'orienter vers la solution la plus simple qui puisse satisfaire les besoins du client.

Feedback

Le retour est immédiat pour les développeurs grâce aux tests unitaires. Pour les clients, le retour se fait à l'échelle de quelques jours grâce aux tests fonctionnels qui leur permettent d'avoir une vision permanente de l'état du système.

Un feedback permanent est positif pour le client qui a une bonne vision du projet, peut détecter tout écart par rapport au planning et à ses attentes de manière à les corriger rapidement. Pour les développeurs, un feedback permanent permet de repérer et de corriger les erreurs beaucoup plus facilement.

Cette notion de feedback est indispensable pour que le projet puisse accueillir le changement.

Courage

Du courage est nécessaire aussi bien chez le client que chez les développeurs. Pour mener à bien un projet XP, le client doit avoir le courage de donner un ordre de priorité à ses exigences, de reconnaître que certains de ses besoins ne sont pas toujours très clairs. De son côté, le développeur doit avoir le courage de modifier l'architecture même si le développement est déjà bien avancé, de jeter du code existant et d'accepter qu'il est parfois plus rapide et efficace de réécrire une portion de code à partir de zéro plutôt que de bricoler du code existant.

3.1.3 Principes de base

Feedback rapide

L'idée de base est issue de la psychologie de l'apprentissage : plus le retour suit de près une action, plus l'apprentissage qui en résulte est important. C'est pour utiliser au mieux cette caractéristique qu'eXtreme Programming préconise des itérations de courte durée et l'implication forte du client. Un retour rapide pour le développeur permet une correction ou un changement de direction plus rapide et un retour rapide vers le client lui permet de tester en permanence l'adéquation du système à ses besoins et au marché.

Assumer la simplicité

XP recommande de traiter tous les problèmes par la solution la plus simple possible, partant du principe qu'un travail propre, simple et minimal aujourd'hui est facile à améliorer par la suite.

Changements incrémentaux

Une série de petits changements progressifs est toujours bien plus sûre et bien plus efficace qu'un grand chambardement. Ceci est valable à plusieurs niveaux : dans un projet XP, l'architecture et la conception changent petit à petit, de même que le planning et la composition de l'équipe.

Accueillir le changement à bras ouverts

La meilleure stratégie est celle qui préserve le maximum d'options tout en apportant une solution aux problèmes les plus urgents.

Un travail de qualité

Parmi les quatre variables qui définissent un projet : taille, coûts, délais et qualité, la qualité n'est pas vraiment indépendante. Pour la réussite d'un projet, la qualité ne peut qu'être "excellente" si chacun aime le travail qu'il fait.

Apprendre à apprendre

Plutôt que de s'en remettre à des théories ou des idées reçues pour répondre aux questions comme : quelle quantité de tests est nécessaire ? combien de temps dois-je passer à retravailler et améliorer le code que j'ai écrit ? etc., mieux vaut apprendre à chacun à apprendre par soi même en fonction de chaque projet.

Faible investissement au départ

Allouer peu de ressources à un projet au départ force les clients et les développeurs à aller à l'essentiel et à dégager ce qui est réellement prioritaire et porteur de valeur.

Jouer pour gagner

L'état d'esprit dans lequel doit se placer une équipe XP peut être comparée à celui d'une équipe de football ou de basket : jouer pour gagner et non pour éviter de perdre.

Des expériences concrètes

Toute décision abstraite doit être testée. Dans cet ordre d'idée, le résultat d'une séance de conception ne doit pas être un modèle finalisé un peu parachuté, mais une série de solutions évoquées au cours de la séance.

Communication ouverte et honnête

Il est essentiel que chacun puisse dire sans peur qu'une partie de code n'est pas optimale, qu'il a besoin d'aide, etc.

Travailler avec et non contre les instincts de chacun

Les gens aiment réussir, apprendre, interagir avec d'autres, faire partie d'une équipe, avoir le contrôle des choses. Les gens aiment qu'on leur fasse confiance. Les gens aiment faire un travail de qualité et voir leurs applications fonctionner. XP cherche à offrir à chacun des moyens d'exprimer ces qualités au cours du projet

Responsabilités acceptées

Il est nécessaire de donner à chacun la possibilité de prendre des responsabilités (par exemple, les développeurs se répartissent eux mêmes les tâches sur la base du volontariat). Cela permet d'éviter les frustrations dans le cas où les responsabilités sont imposées et non délibérément choisies.

Adaptation aux conditions locales

Adopter XP ne signifie pas prendre à lettre la méthode mais intégrer ses principes pour l'adapter aux conditions particulières de chaque projet.

Voyager léger

Un développeur a tendance à transporter beaucoup de choses inutiles. Un débat est soulevé à ce sujet à propos des règles de nommage. Si certaines sont utiles et même indispensables, d'autres sont probablement des contraintes inutiles ou non respectées. Les simplifier pour ne garder que celles qui sont essentielles et utilisées par tous, c'est voyager plus léger.

Mesures honnêtes

La volonté de contrôler le déroulement d'un projet conduit a évaluer, mesurer certains paramètres. Il faut toutefois savoir rester à un niveau de détail pertinent. Par exemple, il vaut mieux dire "cela prendra plus ou moins deux semaines" que de dire "cela prendra 14 176 heures".

3.1.4 Les 12 pratiques XP

Planning game

Cette pratique a pour but de planifier uniquement les releases. La planification se fait sous forme de jeu auquel participent les développeurs et le client.

Pendant une première phase dite d'exploration, le client exprime ses besoins en termes de fonctionnalités. Pour cela, il les écrit sous forme de "user stories", c'est à dire sous forme de courtes descriptions du comportement qu'il attend du système, exprimées avec un point de vue utilisateur. Au cours de cette même phase, les développeurs attribuent à chaque user story un nombre de points, fonction du temps qu'ils estiment nécessaire au développement des fonctionnalités contenues dans chaque user story. S'il s'avère que les user stories nécessitent un temps de développement trop long, elles sont découpées en scénarios élémentaires.

Dans une deuxième phase dite d'engagement, les user stories sont triées en fonction de la valeur qu'elles apportent au client et des risques encourus lors de leur développement. Ceci permet d'aboutir à un classement des user stories par ordre de priorité. En fonction de la vélocité de l'équipe, les développeurs et le client s'entendent sur la quantités de user stories qui seront développées au cours de l'itération à venir et qui constitueront la prochaine release. La vélocité évoquée ci-dessus est un indicateur du nombre de points qui peuvent être développées au cours d'une itération (elle se calcule en faisant le rapport du nombre de points développés au cours de l'itération

précédente et du produit du nombre de développeurs par la durée de l'itération).

Enfin la phase de direction permet de mettre à jour le planning de la prochaine release.

Cette pratique permet de combiner les priorités du client avec les estimations des développeurs afin de convenir du contenu de la prochaine release et de la date à laquelle elle devra être prête.

Petites releases

Pour une bonne gestion des risques, la sortie des releases doit intervenir le plus souvent possible. En conséquence, d'une version à l'autre, l'évolution doit être la plus petite possible, tout en s'efforçant d'apporter le plus de valeur ajoutée et des fonctionnalités dans leur intégralité.

Utilisation de métaphores

XP recommande d'utiliser des métaphores pour décrire l'architecture du système. De telles images permettent à tout le monde (y compris les commerciaux qui n'ont pas forcément de grandes compétences techniques) d'avoir une vision globale du système et d'en comprendre les éléments principaux ainsi que leurs interactions.

Conception simple

La simplicité est une des valeurs fondamentales d'XP. Il faut toujours développer la solution la plus simple possible et éviter de développer plus que ce dont on a besoin. Ceux qui pratiquent XP résument cela sous la phrase YAGNI ("You ain't gonna need it", « vous n'en aurez pas besoin »). Les seules exigences sont de satisfaire tous les tests, de ne jamais dupliquer une logique et d'utiliser le moins possible de classes et de méthodes.

Dans ce même ordre d'idées, la documentation produite lors d'un projet XP se réduit au minimum vital, c'est à dire la documentation demandée par le client.

Tests (unitaires et fonctionnels)

Les tests unitaires sont écrits et effectués par les développeurs pour vérifier le bon fonctionnement et la non régression des méthodes ou des constructeurs. Pour une qualité de code encore meilleure, il est recommandé d'écrire les tests avant le code de l'application.

Les tests fonctionnels sont conçus par le client et lui permettent d'une part de vérifier le fonctionnement global du système, de contrôler l'évolution du projet, et d' d'affiner l'expression de ses besoins.

Refactoring du code

Les développeurs d'un projet XP doivent s'habituer à retravailler un peu chaque jour du code existant et fonctionnant parfaitement pour le maintenir propre, le rendre plus lisible et plus robuste.

Le but de cette pratique est de simplifier le code, tout en faisant en sorte que tous les tests soient satisfaits. D'un point de vue purement fonctionnel, cette simplification n'est pas nécessaire puisqu'elle intervient sur du code qui fonctionne parfaitement. En revanche, le refactoring du code assure que l'ajout de fonctionnalités supplémentaires sera facilité. Le refactoring tend à produire un code mieux pensé, plus modulaire, sans duplications de code et donc plus facile à maintenir.

Programmation en binôme

Toute l'écriture du code se fait à deux personnes sur une même machine, avec une seule souris et un seul clavier. On distingue deux rôles : le pilote ("driver"), celui qui a le clavier, cherche la meilleure approche sur une portion de code bien précise tandis que l'autre développeur, le "partner" peut observer avec beaucoup plus de recul et ainsi suggérer d'autres solutions ou soulever des problèmes d'ordre plus général.

Au premier abord, cette pratique peut sembler être une perte de temps, mais il s'avère que le travail se fait plus vite, que le code est de meilleure qualité (moins d'erreurs de syntaxe, meilleur découpage, etc.), avec une meilleure compréhension du problème. Pour les développeurs, le travail est moins fatiguant.

Les binômes ne doivent pas être statiques : chacun change de partenaire relativement souvent. Ceci pour un meilleur partage des connaissances et pour permettre à chacun d'avoir une relativement bonne vision sur l'ensemble du code.

Appropriation collective du code

Toute l'équipe est sensée connaître la totalité du code (plus ou moins dans le détail selon les parties, évidemment). Cela implique que tout le monde peut intervenir pour faire des ajouts ou des modifications sur une portion de code qu'il n'a pas écrit lui même si cela s'avère nécessaire.

Intégration continue

Après chaque fin de tâche, c'est à dire plusieurs fois par jour, le code nouvellement écrit doit être intégré à l'existant de manière à avoir à tout moment un existant fonctionnel qui passe avec succès tous les tests. Ainsi, quand une tâche est démarrée, elle peut se fonder sur la version la plus à jour de ce qui a déjà été fait.

Pas de surcharge de travail

L'objectif est de ne pas dépasser 40 heures de travail par semaine pour les développeurs. Il ne s'agit pas de chercher à travailler peu, mais les spécialistes d' eXtreme Programming ont pris conscience du fait que la surcharge de travail, en plus d'être désagréable, est néfaste. En effet, le code devient moins bien pensé, le refactoring est laissé de côté, ce qui conduit à une baisse de la qualité et à une recrudescence des bugs. En vertu de ce principe, une équipe ne doit jamais être surchargée de travail plus de deux se-

maines consécutives. Si cela devait toutefois se produire, l'équipe se doit de réagir en redéfinissant la quantité de user stories à implémenter au cours de l'itération ou en modifiant sa manière de procéder.

Client sur site

L'implication forte du client passe par la présence sur site d'une personne minimum à temps plein pendant toute la durée du projet. Cette personne doit avoir à la fois le profil type de l'utilisateur final et une vision plus globale du contexte pour pouvoir préciser les besoins, leur donner une ordre de priorité, les transcrire sous forme de user stories, et établir les tests fonctionnels.

La présence du client sur site est dictée par des impératifs en matière de réactivité. En effet, au fil de la programmation, les développeurs soulèvent fréquemment des questions sur des points non abordés ou restés obscurs. En étant sur place, le client peut ainsi apporter immédiatement des réponses à ces questions, évitant ainsi que les programmeurs commencent à développer certaines fonctionnalités sur la base de ce qu'ils supposent être les désirs du client.

La présence à temps plein du client sur site n'implique pas forcément que cette activité occupe la totalité de son temps : il est tout à fait envisageable pour le client de continuer une partie de son activité tout en étant délocalisé auprès de l'équipe de développeurs.

Standards de code

Il est nécessaire de disposer de normes de nommage et de programmation pour que chacun puisse lire et comprendre facilement le code produit par les autres. Ceci est d'autant plus essentiel que la propriété du code est collective et que les programmeurs sont amenés à changer de binôme régulièrement. En général, les conventions adoptées lors des projets XP sont assez intuitives et résultent de pratiques plutôt naturelles chez les développeurs.

3.1.5 Cycle de vie

Ce paragraphe donne une vision du cycle de vie idéal d'un projet XP. Plusieurs phases composent le cycle de vie d'une application.

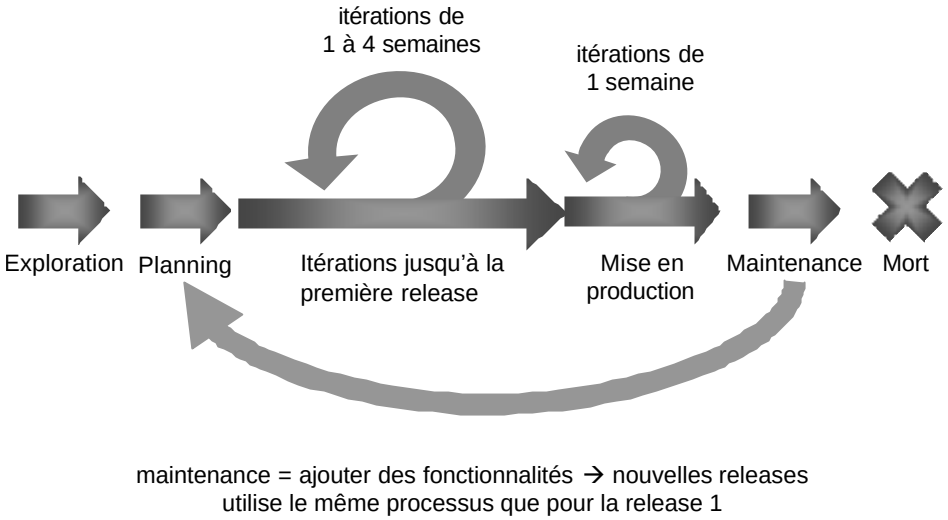


Figure 7 - Les grandes lignes du cycle de vie d'un projet XP

Exploration

Au cours de cette phase, les développeurs se penchent sur des questions d'ordre technique destinées à explorer les différentes possibilités d'architecture pour le système et à étudier par exemple les limites au niveau des performances présentées par chacune des solutions possibles.

Le client de son côté s'habitue à exprimer ses besoins sous forme de user stories que les développeurs devront estimer en terme de temps de développement.

Planning

Nous ne détaillerons pas ici les procédures entrant dans la phase de planning, ceci ayant déjà été fait plus haut, dans le paragraphe intitulé "planning game".

Le planning de la première release est fait de telle sorte qu'un système pourvu uniquement des fonctionnalités essentielles soit mis en production dans un temps minimum et soit enrichi par la suite.

Le planning game dure un ou deux jours et la première release est en général programmée pour deux à six mois plus tard.

Itérations jusqu'à la première release

C'est la phase de développement à proprement parler de la première version de l'application. Celle ci se fait sous forme d'itérations de une à quatre semaines.

Chaque itération produit un ensemble de fonctionnalités passant avec succès les tests fonctionnels associés. La première itération est dédiée à la mise en place de l'architecture du système.

Ces itérations courtes permettent de détecter rapidement toute déviation par rapport au planning qui a été fait jusqu'à la sortie de la release. En cas de déviation, quelque chose est à revoir, soit au niveau de la méthode, soit au niveau des spécifications, soit au niveau du planning.

Durant les itérations, de brèves réunions réunissent toute l'équipe quotidiennement pour mettre chacun au courant de l'avancement du projet.

Mise en production

Les itérations de cette phase sont encore plus courtes, ceci pour renforcer le feedback. En général, des tests parallèles sont conduits au cours de cette phase et les développeurs procèdent à des réglages affinés pour améliorer les performances (performance tuning).

A la fin de cette phase, le système offre toutes les fonctionnalités indispensables et est parfaitement fonctionnel et peut être mis à disposition des utilisateurs.

Maintenance

Il s'agit dans cette phase de continuer à faire fonctionner le système désormais existant et de lui adjoindre les fonctionnalités secondaires qui avaient volontairement été laissées de côté jusque là.

Le développement de nouvelles fonctionnalités sur un système déjà mis en production requiert une prudence accrue de la part des développeurs et cette phase est donc cruciale.

A chaque nouvelle release, une nouvelle phase d'exploration rapide doit avoir lieu.

Mort

La fin d'un projet intervient quand le client n'arrive plus à écrire de user stories supplémentaires ce qui signifie que pour lui, tous ses besoins ont été satisfaits ou quand le système n'est plus capable de recevoir de modifications pour satisfaire de nouveaux besoins tout en restant rentable.

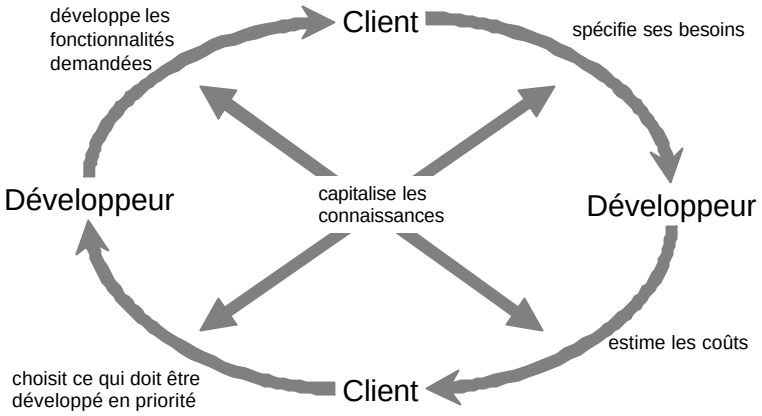


Figure 8 - Interactions entre les acteurs d'un projet XP

3.1.6 Rôles

Développeur

Il est l'élément principal d'un projet XP. En apparence, le développeur passe simplement son temps à écrire des lignes de code, rajouter des fonctionnalités, simplifier, optimiser son code. Mais son rôle ne se limite pas à cela. La principale qualité d'un développeur est sa capacité à communiquer. XP requiert aussi la capacité à travailler en binôme, aptitude qui n'est pas forcément requise pour d'autres types de projets. Enfin, un développeur doit s'habituer à la simplicité : construire la solution la plus simple et ne construire que ce qui est absolument nécessaire doit devenir un réflexe.

Un développeur doit avoir d'autres compétences plus techniques : être capable d'écrire du code propre, de pratiquer le refactoring, de recourir aux tests unitaires, etc.

Enfin, XP requiert du courage de la part des développeurs.

Client

Le client est l'autre moitié du duo essentiel dans l'approche eXtreme Programming. Le développeur sait comment programmer et le client sait quoi programmer.

Pour un projet XP, le client doit apprendre à exprimer ses besoins sous forme de user stories, à leur donner un ordre de priorité et à dégager ce qui est essentiel et valorisant pour lui.

Dans l'idéal, le client a à la fois le profil de l'utilisateur et une vision plus élevée sur le problème et l'environnement du business dans lequel le projet s'inclut.

Le client doit aussi apprendre à écrire les cas de tests fonctionnels et faire preuve, lui aussi, de courage.

Testeur

Étant donné que les tests unitaires sont à la charge des développeurs, le testeur a pour rôle d'aider le client à choisir et à écrire ses tests fonctionnels. Le testeur n'est pas une personne isolée, chargée de mettre le système en défaut et d'humilier les développeurs : il s'agit juste d'une personne chargée de faire passer régulièrement la batterie de tests.

Tracker

C'est un peu la conscience de l'équipe. Son rôle est d'aider l'équipe à mieux estimer le temps nécessaire à l'implémentation de chaque user story et de garder un œil sur le planning en relation avec l'avancement réel du projet.

C'est en quelque sorte l'historien et le rapporteur de l'équipe, chargé de collecter toutes les informations qui peuvent s'avérer utiles.

Coach

Le coach a la responsabilité globale de tout le processus. Son rôle est de recadrer le projet, d'ajuster les procédures. Toute la difficulté de sa tâche réside dans le fait qu'il se doit d'intervenir de la manière la moins intrusive possible. Au fur et à mesure de la maturation de l'équipe, sont rôle diminue et l'équipe devient plus autonome.

Consultant

La programmation en binôme rend assez peu probable l'existence de domaines de compétences dans lesquels seuls un ou deux membres de l'équipe ont des connaissances suffisantes. C'est une force car cela rend l'équipe très flexible mais c'est aussi une faiblesse car la volonté de simplicité se fait parfois au détriment de connaissances techniques très poussées. Quand le problème se présente, l'équipe a recours aux services d'un consultant. Le rôle d'un consultant est d'apporter à l'équipe les connaissances nécessaires pour qu'ils résolvent eux mêmes leur problème et non de leur apporter une solution toute faite.

Big Boss

Le Big Boss apporte à l'équipe courage et confiance.

3.1.7 Forces et faiblesses

Extreme Programming apparaît comme la plus radicale des méthodes agiles.

Cette méthode se révèle particulièrement efficace dans le cadre de petits projets. XP réalise des applications de qualité grâce à la rigueur imposée sur les tests, qui plus est collent au désirs du client puisque celui-ci est intégré au projet de A à Z.

Aussi efficace qu'elle soit, la méthode XP n'est pas applicable dans tous les cas. Dans le cadre d'un projet de type forfaitaire où le prix, les délais et les besoins sont fixés, XP ne peut pas réellement être mise en œuvre. Le cas d'une équipe supérieure à une douzaine de personnes est un autre exemple

où XP est difficilement adaptable car le nombre risque de ralentir, d'alourdir les procédures et de rendre la communication plus difficile et moins efficace.

Enfin, XP est sans doute une des plus contraignante des méthodes agiles, aussi bien pour le client que pour les développeurs. En effet, l'investissement demandé au client est très important puisqu'il doit déléguer une personne à temps plein sur le lieu où est développée l'application, avec toutes les conséquences que cela induit. En ce qui concerne les développeurs, la pratique de la programmation en binôme n'est pas forcément très bien ressentie par tous. Or un projet XP ne peut pas être un franc succès si tous ses participants n'adhèrent pas pleinement à la méthode. Avant de se lancer dans un tel projet, il faudra tout d'abord convaincre le client et motiver l'équipe ce qui n'est pas toujours aisé.

3.2 DYNAMIC SOFTWARE DEVELOPMENT METHOD (DSDM)

3.2.1 Les origines de DSDM

La méthode DSDM est née du même constat que toutes les méthodes agiles, à savoir que les temps de développement étaient jusqu'à présent trop longs, que les applications livrées ne correspondaient pas toujours exactement aux besoins, que les développeurs étaient peu impliqués dans la conception et que personne n'avait de vue complète des systèmes développés.

DSDM s'appuie aujourd'hui sur une association loi 1901, indépendante et à but non lucratif, dont le but est de développer et de promouvoir une méthode pour le développement rapide d'applications. Il s'agit d'une approche dynamique qui requiert l'implication des utilisateurs.

C'est à la suite d'une initiative britannique en 1994 qu'est née DSDM. L'approche ayant été adoptée par des entreprises prestigieuses comme British Airways, Barclays Bank, Marks and Spencer, etc., l'association a connu une croissance exponentielle et fait preuve aujourd'hui d'une volonté de diffusion internationale.

3.2.2 9 principes fondamentaux

1. L'implication active des utilisateurs est impérative

DSDM part du principe qu'une implication forte des utilisateurs est nécessaire pendant tout le cycle de vie pour éviter des retards dans la prise de décision. Les utilisateurs ne sont pas considérés comme de simples fournisseurs d'information mais bien comme des acteurs à part entière de l'équipe.

2. Les équipes DSDM doivent être autorisées à prendre des décisions

Dans le cadre de la méthode DSDM, une équipe regroupe des développeurs et des utilisateurs. Cette équipe doit être capable de prendre des décisions sur l'évolution et la modification des besoins et de déterminer le niveau de fonctionnalité sans demander l'aval de la direction.

3. Le produit est rendu tangible aussi souvent que possible

Pour des raisons de flexibilité, DSDM se fonde sur la livraison fréquente de fournitures. Les délais sont volontairement pris très courts ce qui force à définir quelles sont les activités qui permettront d'atteindre les résultats prioritaires dans le temps imparti.

4. L'adéquation au besoin métier est le critère essentiel pour l'acceptation des fournitures

L'objectif de DSDM est de livrer à temps des fonctions métiers qui permettent de satisfaire les besoins de l'entreprise. L'élaboration d'un système plus global n'est pas une priorité : elle peut être réalisée après coup.

5. Un développement itératif et incrémental permet de converger vers une solution appropriée

Le principe d'évolution incrémentale des systèmes permet de tirer un meilleur parti du contact entre développeurs et utilisateurs. Cette manière de procéder permet un retour permanent des utilisateurs vers les développeurs et une meilleur adéquation des systèmes construits avec les besoins.

De plus, le caractère itératif et incrémental de la méthode permet la livraison de solutions partielles pour répondre à des besoins immédiats.

6. Toute modification pendant la réalisation est réversible

Il est nécessaire de maîtriser la gestion de configuration du logiciel en développement de manière à ce que tout changement effectué soit réversible. Parfois, pour annuler certains changements, il est plus simple de reconstruire que de revenir en arrière. Il importe de s'efforcer de rendre toute transformation réversible.

7. Les besoins sont définis à un niveau de synthèse

Le schéma directeur fixe les exigences du système à un niveau suffisamment élevé pour permettre une évolution itérative des niveaux plus détaillés.

8. Les tests sont intégrés pendant tout le cycle de vie

Les tests font partie intégrante de l'activité de développement. Ils servent à valider au fur et à mesure du développement du système que celui-ci est opérationnel tant sur le plan technique que fonctionnel. En fin de cycle, les tests sont les garants du bon fonctionnement du système.

9. Un esprit de coopération entre tous les acteurs est primordial

Dans la démarche DSDM les fonctions détaillées ne sont pas figées dès le départ. Le demandeur et le réalisateur doivent faire preuve de souplesse tout au long du cycle de vie, ce qui implique la nécessité d'avoir une procédure de gestion des modifications peu contraignante.

3.2.3 5 phases

DSDM se définit plus comme un "canevas" que comme une méthode, c'est à dire qu'elle ne fournit que "l'ossature" du processus global et une description des processus et produits qui devront être adaptés à un projet ou à une entreprise particulière.

Le processus de développement est constitué de cinq phases :

- L'Etude de Faisabilité
- L'Etude du " Business "
- Le Modèle Fonctionnel itératif
- La conception et le développement itératifs
- La Mise en œuvre dans l'environnement de travail.

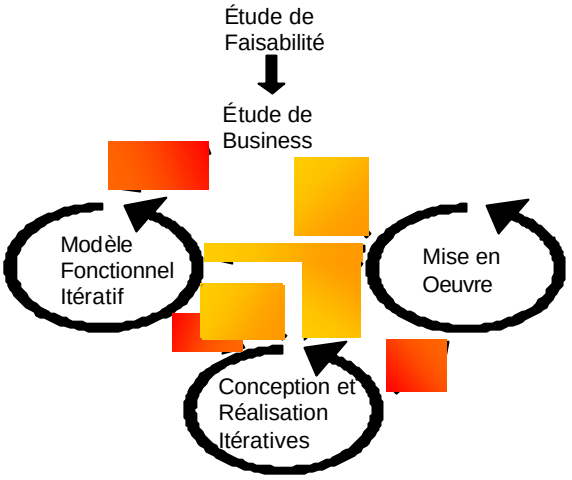


Figure 9 - Cycle de vie d'un projet DSDM

L'Etude de Faisabilité et l'Etude du "Business" sont réalisées de manière séquentielle. Comme elles établissent les règles de base pour le reste du développement elles doivent être terminées avant que les trois autres phases itératives ne démarrent. Les trois phases suivantes peuvent converger ou se chevaucher mais cela dépend essentiellement de la nature du système développé et les outils utilisés. Chaque phase génère un minimum de livrables répondant à un besoin bien précis.

L'Etude de Faisabilité

Première tâche à accomplir : déterminer si DSDM est l'approche qui convient pour le projet considéré. C'est dans cette phase que le problème à résoudre est posé. Il faut ensuite évaluer les coûts et d'un point de vue technique, l'aptitude du système envisagé à résoudre le problème métier. dans la mesure où les besoins sont généralement urgents, cette phase est nécessairement courte et ne dépasse pas quelques semaines. Cette phase ne laisse pas suffisamment de temps pour produire une documentation volumineuse. Le Rapport de Faisabilité couvre donc les thèmes habituels mais avec peu de détails. L'Etude de Faisabilité produit aussi un Plan Global de développement, qui vient renforcer l'hypothèse que l'objectif fixé est réalisable et, de

manière facultative, un Prototype de faisabilité pour montrer que la solution est réalisable.

L'Etude du Business

L'étude du Business ou étude des besoins industriels doit elle aussi être aussi courte que possible. Elle a pour but de permettre la compréhension suffisamment approfondie des besoins et des contraintes techniques pour ne pas mettre en péril la suite du déroulement du projet.

Cette phase permet l'étude des processus à automatiser et des besoins en informations. Cette étude passe par l'organisation d'Ateliers Facilités (travail collectif animé par un médiateur) dont le rôle est de dégager la Définition du Domaine Industriel qui répertorie non seulement les processus de l'entreprise et informations associées mais aussi les classes (ou types) d'utilisateur qui seront concernées d'une manière ou d'une autre par la mise en place du système. Ces classes permettent d'identifier les utilisateurs qui participeront au développement. Ces utilisateurs auront pour responsabilité d'une part de fournir les informations nécessaires au projet et d'autre part de tenir l'ensemble de la communauté informatique au courant de son évolution. Les Ateliers Facilités permettent aussi de classer par ordre de priorité toutes les fonctionnalités à développer. L'ordre de priorité repose principalement sur le degré d'urgence des besoins métiers, mais il peut aussi intégrer des besoins non fonctionnels tels que la sécurité par exemple.

La Définition de l'Architecture Système, qui décrit aussi bien les plates-formes opérationnelles et de développement que l'architecture du logiciel développé (c'est à dire les principaux composants et interfaces associées), est un autre élément essentiel de L'étude du Business. La définition de l'architecture à ce niveau est indispensable car certains modules commenceront à être développés dès l'étape suivante. Bien entendu, la Définition de l'Architecture Système est susceptible d'être ajustée au cours des phases ultérieures.

Enfin, le Plan Global issu de l'Etude de Faisabilité est détaillé pour créer le plan Global de Prototypage qui décrit le mode de développement à utiliser au cours des deux phases suivantes ainsi que le mode de contrôle et de tests à mettre en oeuvre.

Modèle Fonctionnel Itératif

Le Modèle Fonctionnel Itératif a pour but de préciser la définition de tous les aspects du système qui sont liés au business. Il consiste en une description plus précise des besoins de haut niveau, à la fois en termes de traitement et d'information. Cette phase produit aussi bien des modèles d'analyse standard que des modules logiciels.

Les phases "Modèle Fonctionnel Itératif" et "Conception et Réalisation Itératives" reposent toutes deux sur un cycle de quatre activités :

1. Identifier ce qui doit être produit.
2. Décider comment et à quel moment le faire.
3. Créer le produit.

4. Vérifier qu'il a été développé de manière appropriée (par le biais de tests et à la lumière des documents précédemment produits)

Les modules logiciels du Modèle Fonctionnel répondent aux principaux besoins, y compris les besoins non fonctionnels comme la sécurité ou la facilité d'utilisation. Ils subissent des tests au fur et à mesure de leur production : ceci sous-entend les tests techniques unitaires ainsi que des mini-tests de recettes effectués directement par les utilisateurs. Les tests permettent de déterminer si les composants produits peuvent ou non être réutilisables et servir de base à un ensemble de fonctionnalités. Les aspects non fonctionnels sont testés au cours de la phase suivante.

En pratique, il est souvent plus simple et plus intuitif de traiter intégralement un domaine fonctionnel et ses aspects non fonctionnels avant de passer à un autre domaine. Selon la manière dont l'application a été découpée en divers modules plus ou moins indépendants, les phases "Modèle Fonctionnel Itératif" et "Conception et Réalisation Itératives" seront plus ou moins imbriquées.

Conception et Réalisation Itératives

Le passage du Modèle Fonctionnel à La Conception et au Développement est subordonné à l'accord sur un Prototype Fonctionnel présentant tout ou partie du Modèle Fonctionnel. Si ce prototype ne présente qu'une partie du modèle fonctionnel, les activités de prototypage et de développement peuvent être menées en parallèle.

La phase Conception et Réalisation Itératives est l'étape qui a pour but de rendre le système conforme à des standards assurant qu'il peut être placé entre les mains des utilisateurs. Le produit majeur de cette phase est le Système Testé. Dans des cas particuliers, le client peut exiger l'ajout d'une phase de test à la fin de chaque itération, mais DSDM préconise l'étalement des tests au fur et à mesure des phases "Modèle Fonctionnel Itératif" et "Conception et Réalisation Itératives".

Le Système testé n'est pas forcément complet sur le plan fonctionnel mais il se doit de satisfaire les besoins qui ont été définis comme prioritaires pour l'étape en cours.

Mise en Œuvre

C'est au cours de la phase de Mise en Oeuvre qu'est effectuée la migration du système de l'environnement de développement à l'environnement opérationnel. Cette phase inclut également la formation des utilisateurs n'ayant pas pris part au projet.

Le Système Livré et la documentation (y compris la documentation utilisateur) font partie des livrables de cette phase. La documentation utilisateur est finalisée durant cette phase mais elle doit avoir été commencée durant la phase Conception et Réalisation Itératives. C'est l'une des responsabilités des utilisateurs ambassadeurs que de s'assurer de la qualité de la documentation et de la formation utilisateur.

Le Document de Capitalisation de Projet est l'autre produit de cette phase. Il permet de faire le point sur les besoins exprimés et la manière dont le système y répond. Quatre résultats sont possibles :

1. Tous les besoins ont été satisfaits. Aucune autre tâche supplémentaire n'est donc nécessaire.
2. Un aspect fonctionnel important a été provisoirement ignoré. Il faut dans ce cas revenir à la phase Etude du business et re-dérouler le processus à partir de là.
3. Une fonctionnalité à faible priorité a été délibérément ignorée. Pour l'ajouter, il suffit de revenir à la phase Modèle fonctionnel itératif.
4. Un aspect technique mineur a également été délaissé : il peut à présent être traité en revenant à la phase Conception et Réalisation Itératives.

3.2.4 Les rôles

DSDM définit de manière très précise des rôles à attribuer aux différentes personnes qui vont prendre part au projet. Il est important de noter qu'un rôle ne correspond pas nécessairement à une et une seule personne physique. En fonction de la taille du projet, une personne peut se voir attribuer plusieurs rôles et un même rôle peut être tenu par plusieurs personnes.

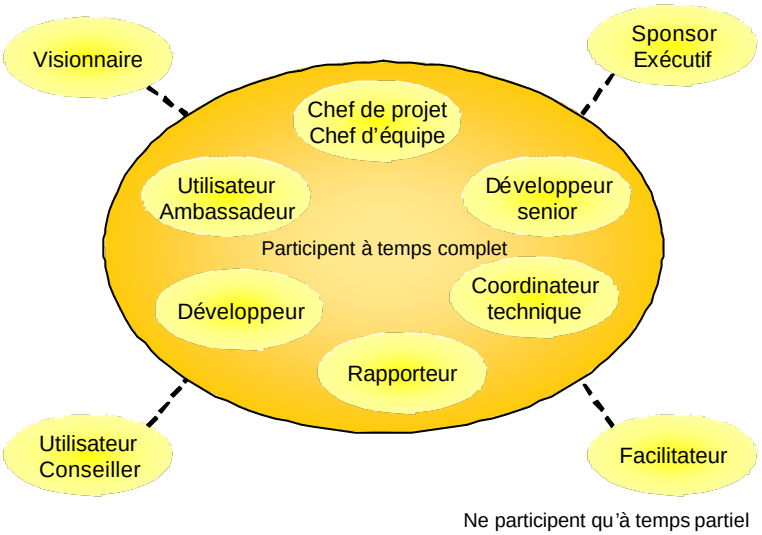


Figure 10 - Les rôles dans une équipe DSDM

Rôles	Responsabilité
Sponsor exécutif	Assurer l'efficacité et la rapidité du processus décisionnel Réagir aux questions de progression Assurer la disponibilité des fonds et autres ressources Contrôler le maintien de l'adéquation du projet au business Engagement et disponibilité pendant tout le cycle de développement
Visionnaire	Promouvoir la traduction de la vision en pratique de travail Avoir une vision plus large de bout en bout du processus du

	business
	Contribuer aux sessions relatives aux besoins clefs
	Contribuer aux sessions importantes de conception
	Contribuer aux sessions importantes de révision
	Résoudre les conflits dans l'ensemble du domaine du business
	Assurer la disponibilité des ressources utilisateurs
	Contrôler les progrès
	Engagement et disponibilité pendant toute la durée du cycle de développement
Utilisateur ambassadeur	Fournir les données essentielles aux sessions concernant les besoins du business et la conception
	Fournir le détail des scénarios du business
	Communiquer avec les autres utilisateurs et obtenir leur aval
	Fournir des données aux sessions de prototypage
	Réviser la documentation
	Evaluer et accepter le logiciel livré
	Fournir la documentation pour les utilisateurs
	Assurer que la formation se fait de façon adéquate
	Organiser et contrôler les tests utilisateurs
Utilisateur conseiller	Fournir l'information à la demande
	Participer au processus de prototypage et de révision par ses conseils et son assistance sur des questions pratiques importantes
	Approuver les conceptions et prototypes dont l'utilisation est acceptable
	Contribuer aux tests business et d'ergonomie
Chef de projet	Rendre compte aux cadres supérieurs et au Comité directeur
	Planifier le projet
	Contrôler les progrès
	Gérer le risque
	Orienter et motiver les équipes
	Définir les objectifs des équipes
	Présider les réunions du projet
	Gérer la participation des utilisateurs
	Traiter les exceptions
	Identifier et faire appel aux rôles spécialistes lorsque requis
	Traiter les problèmes de progression des équipes du projet
Coordinateur technique	Définir l'environnement technique
	Contrôler les procédures de gestion de configuration
	Assurer l'adhésion aux standards de bonnes pratiques
	Conseiller et coordonner les activités techniques de chaque

	équipe
	Assister aux sessions de prototypage pour conseiller sur l'application des standards
	Assurer que les objectifs de maintenabilité sont satisfaits
	Convenir et contrôler l'architecture logiciel
	Identifier les possibilités de réutilisation
	Gérer le contrôle de mise en pratique
Chef d'équipe	Développer et maintenir la conformité à la définition du domaine d'activité du business
	Assurer le traitement des besoins du business formulés par les utilisateurs
	Organiser les sessions de prototypage et de révision entre les utilisateurs et les développeurs
	Encourager la participation complète des membres de l'équipe dans le cadre des rôles et des responsabilités définis
	Conduire les sessions de prototypage à atteindre les objectifs fixés dans les délais et dans le cadre des contraintes de l'ordonnancement
	Assurer la documentation et le contrôle des changements
	Promouvoir le bien-être et la motivation de l'équipe
Développeur et développeur confirmé	Création d'une documentation détaillée lorsque nécessaire
	Travailler avec les utilisateurs pour définir les besoins du business, créer les prototypes et les programmes finalisés
	Créer d'autres composants comme, par exemple, un modèle logique de données
	Créer la fiche technique des tests
	Réviser et tester son propre travail et celui des autres
Facilitateur	Convenir de la portée de l'atelier avec le chef de projet
	Planifier l'atelier
	Etre familiarisé avec le domaine du business
	Interviewer les participants pour s'assurer qu'ils conviennent ainsi que de la réalisation de tout travail préparatoire
	Faciliter la tâche de l'atelier pour qu'il satisfasse à ses objectifs
	Faire une revue de l'atelier par rapport à ses objectifs
Rapporteur	Enregistrer tous les points formulés qui sont pertinents pour le système
	Aider à l'interprétation subséquente
	Gérer la distribution de la documentation du projet

Selon la taille et la nature du projet, le chef de projet peut juger nécessaire de faire appel à des rôles d'experts pour remplir des fonctions particulières. Citons pour exemple quelques-uns de ces rôles : Consultant Business, Consultant Technique, Spécialiste d'adéquation du système à l'homme,

Chargé de planification capacité/ performance, Expert en matière de sécurité, Architecte données, Responsable de la qualité, Equipiers pour la gestion du support et de la maintenance, etc.

3.2.5 Forces et faiblesses de la méthode

DSDM présente de nombreux avantages : mise en œuvre rapide de solutions prioritaires, adéquation du système aux besoins, respect des délais et des coûts, meilleure formation des utilisateurs, implication des utilisateurs dans le développement, tests intégrés, moins de bureaucratie, souplesse face au changement, conformité avec la norme ISO 9001.

Il est pourtant possible de dégager quelques faiblesses ou quelques points qui se présentent comme un frein à la méthode DSDM.

L'implication des utilisateurs tout d'abord soulève un certain nombre de questions. Comment concrètement impliquer l'utilisateur si l'utilisateur n'est pas le client ? Dans ce cas, comment le client percevra-t-il le pouvoir donné à l'utilisateur de faire évoluer les spécifications ?

De plus, l'équipe, constituée de développeurs et d'utilisateurs, est amenée à prendre des décisions concernant le niveau de fonctionnalités sans l'aval de la direction. La question qui se pose alors est de savoir si l'équipe dispose bien de toutes les informations nécessaires, notamment concernant le Business, pour prendre de telles décisions.

Enfin, DSDM préconise de toujours mettre en place la solution la plus simple qui permet de résoudre le problème posé, partant du principe qu'il est plus simple de rajouter des fonctionnalités par la suite, un peu à la manière d'XP. Il convient peut être de prendre ce point avec modération : s'il ne faut clairement pas tomber dans les pièges du "trop générique", dans certains cas particuliers, il est parfois plus simple de penser dès le début à d'éventuelles fonctionnalités futures (par exemple, si on n'a pas prévu au départ la gestion multilingues d'un système, il est assez difficile d'ajouter cette fonctionnalité après coup).

3.3 ADAPTIVE SOFTWARE DEVELOPMENT

3.3.1 Contexte

Adaptive Software Development est une méthode agile développée par Jim Highsmith, président d'Information Architects, Inc. Elle s'adresse tout particulièrement aux projets e-business. Les plannings traditionnels, associés à un certain degré de certitude et de stabilité du business, ne conviennent plus pour diriger les projets actuels qui doivent être réalisés en des temps très courts, supporter de nombreux changements et incertitudes. Jim Highsmith propose donc avec Adaptive Software Development un cycle de vie plus souple face aux changements qui surviennent constamment. La capitalisation des connaissances et la collaboration entre développeurs, testeurs et client sont deux valeurs essentielles de cette méthode.

3.3.2 Les 6 caractéristiques principales d'ASD

Le cycle de vie préconisé par ASD s'appuie sur six valeurs fondamentales :

Focaliser sur une mission ("mission focused")

En général, les spécifications ne sont pas définies de manière précise dès le début mais le projet est guidé par une mission. La mission est comme une sorte de guide pour le projet : au début, elle encourage l'exploration puis au fur et à mesure que le projet se précise et avance, elle le recadre dans le droit chemin. En fait, on peut voir la mission plutôt comme une frontière que comme un point précis à atteindre. C'est d'ailleurs la mission, si elle se précise au cours de l'avancement du projet, qui empêche que les itérations ne soient que de simples oscillations sans progrès.

Se baser sur des composants ("component-based")

Il est nécessaire de se focaliser non sur les tâches mais sur les résultats, c'est à dire sur les composants d'une application. Dans le cadre d'ASD, on entend par composant un ensemble de fonctionnalités qui doivent être développées durant une itération. A ce titre, la documentation peut être considérée comme un composant livrable. Si elle a été demandée par le client pour la fin d'une itération, elle est considérée comme un composant mais revêt une priorité secondaire par rapport aux fonctionnalités à proprement parler qui procurent des résultats directs au client.

Itérer

A la différence du domaine industriel où la production se fait en grandes séries devoir refaire ou retravailler quelque chose est considéré comme un échec. Dans le cas du développement d'applications, au contraire, la conception se fait de manière itérative et incrémentale : les composants évoluent en fonction du feedback des utilisateurs.

Découper le temps et fixer des deadlines ("timeboxing")

Un des principe d'ASD est la nécessité de fixer des dates butoirs de livraison qui fixent en même temps la fin d'une itération ou d'un projet. Ces deadlines ne doivent pas pour autant servir de prétexte pour surcharger les développeurs lorsque la date approche ou pour négliger la qualité ce qui détruit tout le travail collaboratif qui a été fait auparavant. Le timeboxing est en fait un moyen de forcer la prise de décisions difficiles et de forcer à réévaluer constamment la validité de la mission (ambitions, planning, ressources, etc.)

Gérer le risque ("risk-driven")

Etant donné que les projets auxquels s'adresse ASD sont des projets qu'on pourrait qualifier d' "extrêmes" par leurs délais serrés, leur absence totale de stabilité dans les besoins et les exigences de qualité, une bonne gestion des risques est cruciale. Il est donc nécessaire d'analyser précisément tous les risques critiques qui pourraient mettre en péril le projet avant de le démarrer.

Tolérer le changement

La capacité à supporter des changements de spécifications, de techniques en cours de développement est vue comme un avantage concurrentiel par la méthode ASD.

3.3.3 Le cycle de vie selon Adaptive Software Development

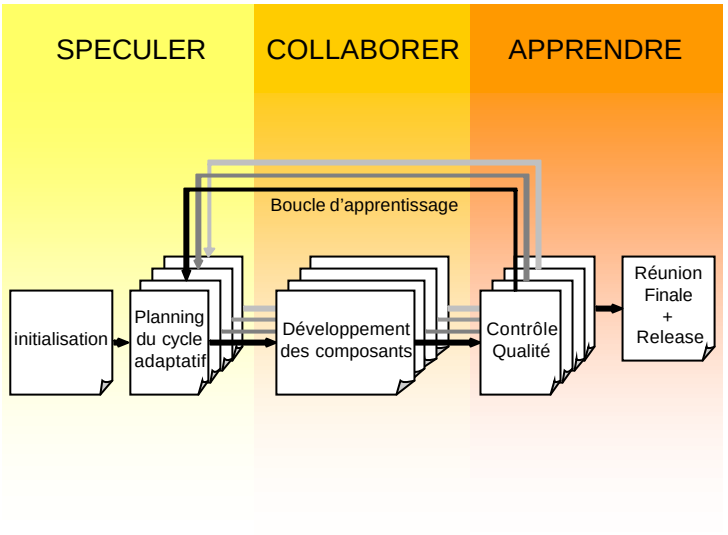


Figure 11 - Le cycle de vie selon Adaptive Software Development

Spéculer

La phase de spéculation comprend les tâches d'initiation et de planification du cycle. Cette phase comprend sept étapes :

- Conduire la phase d'initiation du projet
- Déterminer la date butoir de fin de projet
- Déterminer le nombre optimal d'itérations et la date de fin de chacune d'entre elles
- Donner une mission objective à chaque itération
- Affecter les composants de base aux itérations
- Affecter les technologies et les composants support aux itérations
- Développer une liste de tâches à réaliser

L'initialisation d'un projet est assez similaire à celle préconisée par les approches de management courantes. Elle prend en compte l'établissement de la mission et des objectifs du projet, l'étude des contraintes, l'établissement de l'organisation du projet, l'identification des collaborateurs clés, l'expression des exigences, une première estimation de la taille et de l'étendue du projet et l'identification des risques critiques. Pour gagner du temps, les données de l'initialisation sont collectées au cours de réunions JAD (Joint Application Development). Pour un projet de petite taille, cette phase peut être accomplie en environ une semaine.

La deuxième étape est de déterminer le temps imparti au projet. La date butoir est établie sur la base des résultats de la phase d'initialisation. Le

verbe "spéculer" donné à la phase ne signifie pas pour autant que l'établissement des délais abandonne toute estimation : il signifie juste une prise de conscience de la fragilité des estimations.

En troisième lieu, l'équipe doit décider le nombre idéal d'itérations et affecter à chacune une certaine durée. Pour des projets de petite à moyenne taille, celles-ci durent en général entre 4 et 8 semaines mais ces chiffres ne sont donnés qu'à titre indicatifs et doivent être adaptés à chaque projet.

L'équipe doit ensuite associer à chaque itération une thématique ou un objectif. C'est ce qui assure la visibilité du projet. Chaque cycle produit un ensemble de composants livrables qui rendent le produit visible aux yeux du client. A l'intérieur de chaque itération, l'application est construite, compilée ce qui la rend visible aux yeux de l'équipe de développement.

Enfin, il faut affecter les différents composants à développer aux différentes itérations. La décision de cette affectation se fait en s'assurant que chaque itération produit quelque chose d'utile au client, en identifiant et en gérant les risques, en prenant en compte les interdépendances naturelles entre les composants, et en équilibrant l'utilisation des ressources. Le meilleur outil pour réaliser cette répartition reste encore un tableau dont la première colonne contient tous les composants identifiés classés par catégories (fonctionnalités de base, composants technologiques, support, etc.) et comprenant une colonne par cycle. L'expérience prouve que lorsque ce type de planning est réalisé par l'équipe et non uniquement par le chef de projet, la compréhension du projet est améliorée.

Collaborer

C'est cette phase qui délivre réellement les composants en état de fonctionnement. Durant cette phase, le chef de projet s'occupe plus de veiller à ce que les développeurs collaborent efficacement entre eux plutôt que de concevoir, tester et coder. Comment les personnes interagissent et comment elles gèrent leurs interdépendances sont toujours des problèmes critiques. Dans le cas de petits projets, particulièrement si les membres de l'équipe travaillent à proximité les uns des autres, ces relations peuvent être gérées de manière informelle. La communication se fait bien souvent par des discussions de couloir ou par des bribes de schémas dessinés sur un tableau blanc. Dans le cas d'un projet plus important impliquant une équipe géographiquement disséminée, le problème est plus épineux. ASD n'est pas une méthode fermée sur elle même. Jim Highsmith lui même reconnaît que dans le cadre de projets de petite taille, le développement collaboratif peut être amélioré par l'utilisation de certaines des pratiques issues d'eXtreme Programming, comme par exemple la programmation en binôme ou la propriété collective du code.

Apprendre

Une pratique clé de la méthode ASD est un examen minutieux de la qualité, ce qui est un élément clé de la capitalisation des connaissances.

A la fin de chaque cycle de développement, quatre catégories de connaissance peuvent être complétées :

- Qualité du point de vue de l'utilisateur / client : fournir une certaine visibilité au client et obtenir un feedback de sa part est un facteur clé de capitalisation des connaissances et des expériences. Pour ce faire, ASD recommande de procéder à des réunions appelées "customer focus group" à la fin de chaque itération. Le but de ces séances est d'explorer une version fonctionnelle de l'application et de lister les demandes de modifications émanant du client.
- Qualité du point de vue technique : une méthode standard pour évaluer la qualité d'un point de vue technique est la revue technique ("technical review"). Une revue de la conception doit être faite à la fin de chaque itération alors que des révisions du code et des plans de tests doivent avoir lieu tout au long du cycle.
- Le fonctionnement de l'équipe et les pratiques utilisées. Il s'agit de contrôler les performances de l'équipe. Ceci est fait au cours de réunions qui portent le nom de "postmortems" à la fin de chaque itération et à la fin du projet. Ces réunions sont de véritables autopsies du projet au cours desquelles se posent quatre questions : qu'est ce qui fonctionne bien ? qu'est ce qui ne fonctionne pas encore ? sur quelle pratique devons nous insister ? que devons nous réduire ? Ce type de réunions forcent les collaborateurs à apprendre sur eux mêmes et sur la manière dont ils travaillent.
- L'état d'avancement du projet. Ceci n'a pas strictement de lien avec la qualité : il s'agit simplement de revoir le planning au début de chaque itération. Le questionnement basique porte sur : où en est le projet ? où en est le projet par rapport à ce qui était prévu ? où devrait-on en être ? Dans une approche à base de composants, l'état d'avancement d'un projet ne peut pas se résumer en un seul document terminé ou un seul livrable, mais il consiste en un ensemble de composants dans divers états d'avancement.

3.3.4 Forces et faiblesses

Cette méthode fait preuve des qualités de méthodes agiles, à savoir souplesse face au changement, rapidité, respect des coûts et de délais, implication du client pour une application qui correspond mieux aux besoins, etc.

Il n'en reste pas moins qu'ASD reste un cadre très général qui demande à être adapté à chaque projet, et éventuellement à être enrichi d'éléments et de pratiques venus d'autres méthodes agiles.

3.4 CRYSTAL METHODOLOGIES

3.4.1 Origine

Plus qu'une méthodologie, Crystal est un cadre méthodologique très fortement adaptable aux spécificités de chaque projet. Ces méthodes ont été développées par Alistair Cockburn. Nous nous intéresserons ici à une des déclinaisons de Crystal, Crystal Clear, qui rentre parfaitement dans le cadre

de notre étude et se prête à la comparaison avec les autres méthodes mentionnées dans cette partie.

3.4.2 Les valeurs partagées par l'équipe

Communication

La communication est omniprésente dans une équipe qui travaille avec la méthode Crystal. Ceci est une condition sine qua non pour réussir le "jeu coopératif" qu'est le projet. En effet, Alistair Cockburn écrit :

"Software development is a cooperative game, in which the participants help each other in reaching the end of the game – the delivery of the software"

La communication est tellement cruciale que le nombre de membres d'une équipe est limité à 6 personnes pour que l'équipe puisse fonctionner. Il est recommandé que tous les membres de l'équipe soient placés dans la même pièce ou, au pire, dans deux pièces voisines afin de faciliter la communication par la proximité.

Toujours dans l'idée que la communication et la collaboration produisent un travail de meilleure qualité, les schémas de modélisation sont généralement faits en groupe sur un tableau blanc.

Enfin, la collaboration avec les clients / utilisateurs est très serrée. Ce sont les utilisateurs qui expriment leurs besoins en écrivant des cas d'utilisation très sommaires. Toutes les précisions seront apportées par la suite au cours de conversation entre utilisateurs et développeurs.

Releases fréquentes

Dans un souci de souplesse face au changement, il est très important d'espacer au minimum les releases. Cette pratique permet en outre au client d'avoir au moins une partie utilisable de son application sans attendre la fin du projet.

Les livraisons fréquentes contraignent toutefois à réduire les livraisons au strict minimum utile : le code source commenté et le manuel d'utilisation.

Souplesse

Crystal est une méthode très souple, tant au niveau des procédures à suivre que des normes à utiliser. Par exemple, les normes de codage et de nommage sont très peu contraignantes voire inexistantes pour certains points : ceci n'est pas problématique dans la mesure où la taille de l'équipe reste inférieure à 6 personnes et où ces personnes ont l'habitude de travailler ensemble, d'aider les autres à revoir leur code ce qui rend les normes quasi naturelles. L'équipe est relativement libre de s'organiser à sa guise : parmi les développeurs par exemple, on peut distinguer deux types d'approche. Certains commenceront par prendre une feuille de papier et un crayon pour essayer de schématiser la manière dont ils vont construire et organiser leur code avant de démarrer réellement la programmation. D'autres au contraire fonceront tête baissée dans l'écriture du code ce qui implique pour eux de consacrer plus de temps par la suite au refactoring du code. La méthode Crystal n'est pas directive sur la manière de procéder : les personnalités de

chacun sont prises en compte et tant que le travail résultant est le même à la fin, la méthode reste d'une grande souplesse.

3.4.3 Processus Crystal

Spécifications

Une première phase consiste à interroger le client et les utilisateurs pour qu'ils expriment leurs besoins. Une pratique assez répandue consiste à observer les utilisateurs dans leur travail pour mieux connaître leurs besoins et leur environnement. Les spécifications sont collectées sous forme de cas d'utilisation sommaires : il s'agit de décrire en une phrase une interaction entre l'utilisateur et le système. En collaboration avec les utilisateurs, les différents cas d'utilisation sont classés par ordre de priorité ce qui permet de savoir quelles fonctionnalités ont le plus de valeur et quelles fonctionnalités sont à développer en premier.

Conception et planning

Une première ébauche de conception est faite dans cette phase, c'est à dire au tout début du projet. Cela inclut le choix des technologies qui seront utilisées pour la réalisation et une première ébauche de l'architecture pour se donner une vision globale.

Enfin, juste avant d'entrer dans la phase itérative, il convient de planifier les itérations qui vont suivre. Crystal recommande des itérations d'une longueur d'environ 2 ou 3 mois, chacune produisant un livrable fonctionnel.

Itérations

C'est au cours de cette phase que se fait la réalisation proprement dite de l'application.

Le premier travail consiste à clarifier et préciser les spécifications au cours de discussions plutôt informelles entre utilisateurs et développeurs.

Ensuite, les développeurs font tous ensemble un travail de conception pour déterminer la manière dont ils vont mettre en place les objets de base sur lesquels va s'appuyer le système.

Les développeurs se répartissent les différentes parties. Dans les deux semaines qui suivent le début d'une itération, il convient de mettre en place une maquette qui permettra de faire une démonstration aux utilisateurs. Cette pratique permet bien souvent de déceler des incompréhensions dans les besoins exprimés.

Les développeurs sont ensuite libres de procéder comme ils l'entendent pour développer la partie dont ils ont la charge.

Les tests sont omniprésents dans le processus de développement : l'architecture est conçue de manière à pouvoir supporter des tests régressifs et des tests automatiques. Les classes de tests font partie intégrante du code et sont souvent écrite avant le code lui même.

Le refactoring du code est lui aussi permanent. Il se fait en général en binôme pour optimiser la qualité du code.

Deux ou trois fois par itérations, des démonstrations sont faites aux clients / utilisateurs ce qui permet de toujours bien recadrer le projet en conformité avec les exigences du client.

L'écriture du manuel utilisateur est effectuée soit juste avant la livraison d'une release, soit dès le début de l'itération ce qui force les développeurs à définir plus précisément dès le début ce vers quoi ils s'engagent.

L'itération se termine par l'intégration et la mise en production de l'application qui est testée une fois de plus avant d'être livrée au client.

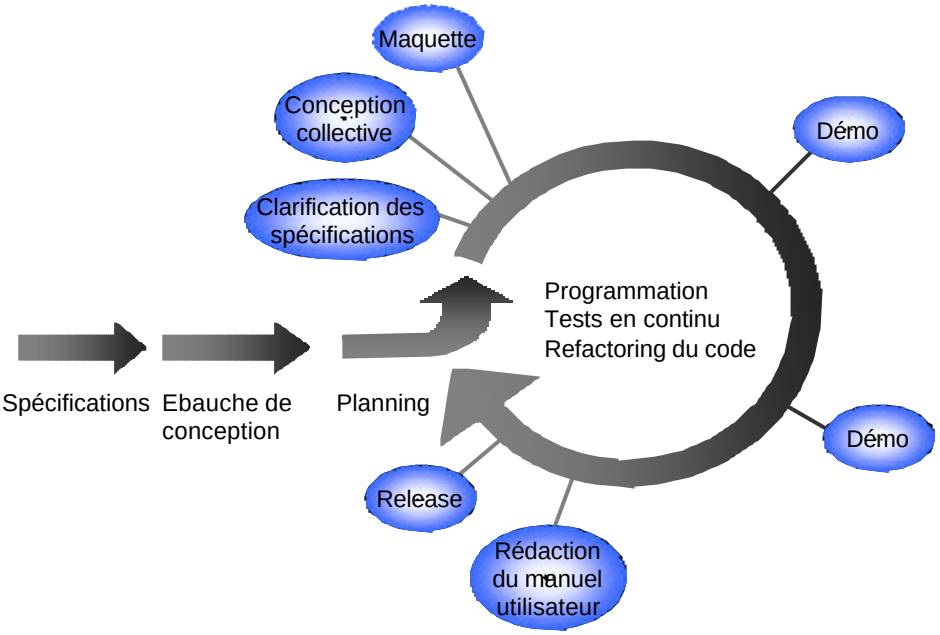


Figure 12 - Processus Crystal

3.4.4 Forces et faiblesses

Crystal Clear présente tous les avantages des méthodes agiles : flexibilité par rapport au changement, rapidité, livraisons fréquentes, etc.

Par contre, dans la version que nous avons décrite ici, elle reste limitée à des équipes de taille inférieure à 6 personnes, en raison de l'absence d'une personne ou d'un groupe chargé de la coordination. Les processus et pratiques de Crystal sont très souples et conviennent parfaitement pour des petites structures. Mais ce qui fait leur efficacité dans les projets de petite taille cause leur inadéquation pour des projets plus importants.

En fait, les approches Crystal et eXtreme Programming sont très proches, à ceci près qu'eXtreme Programming nécessite beaucoup plus de discipline et est plus contraignante. D'un côté, cette particularité rend eXtreme Programming certainement un peu plus productive que Crystal Clear, mais d'un autre côté, tout le monde n'est pas forcément prêt à en accepter les principes alors que Crystal Clear sera certainement acceptée plus facilement.

3.5 SCRUM

3.5.1 Les origines de Scrum

La méthode Scrum est née de la coopération de deux sociétés : Advanced Development Methods (ADM) et WMARK Software (éditeur d'environnement de développement orienté objet). Ces deux compagnies déploraient le manque d'adéquation des anciennes méthodologies avec la programmation orientée objet et les architectures à base de composants. Scrum est le résultat de la volonté de mettre en place des processus empiriques, contrôlés par des mesures quantitatives.

Note : le terme Scrum est emprunté au rugby et signifie "mêlée". La méthode porte ce nom car dans un processus Scrum se tient chaque jour une réunion informelle, appelée Scrum, pour faire le point sur ce qui a été fait, ce qui va être fait le lendemain et ce qu'il reste à faire pour la suite.

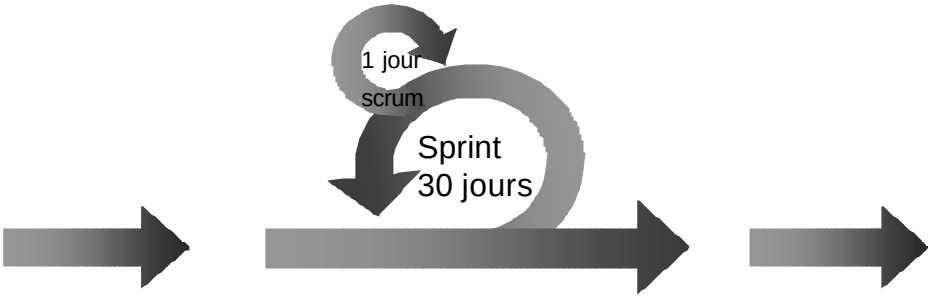
3.5.2 Le processus Scrum : vue générale

Le cycle de vie d'un projet Scrum peut être divisé en trois parties :

- La phase d'initiation est une phase linéaire aux processus explicitement connus, et dont les inputs et outputs sont bien déterminés.
- Le "sprint" de développement est un processus empirique : beaucoup des processus de la phase de sprint sont inconnus et non identifiés. Ce n'est pas pour autant le chaos car des contrôles, incluant notamment la prise en compte du risque, encadrent chaque itération de la phase de sprint pour ne pas tomber dans le chaos, tout en préservant la flexibilité. Le projet est ouvert sur son environnement jusqu'à la phase de clôture. Les livrables peuvent être modifiés à tout moment pendant les deux premières phases.
- La phase de clôture est aussi une phase linéaire dont les processus sont clairement définis.

Figure 13 - Vue globale du processus Scrum

Phase d'initiation



Planning & Architecture
- phases linéaires
- processus bien définis
- backlog
- conceptualisation
- analyse - conception
- architecture système

Sprints de développement
- phase non linéaire
- processus empirique
- flexibilité
- accueil du changement
- contrôles permanents pour éviter le chaos, notamment management du risque

Clôture
- préparation de la release
- documentation finale
- derniers tests
- livraison

La phase d'initiation comporte tout d'abord une activité de planning. Dans le cas du développement d'un nouveau système, il s'agit de réaliser la conceptualisation et l'analyse du système. La phase de planning doit aboutir à la mise en place d'un "backlog", c'est à dire une liste de tâches restant à effectuer, à la définition de la date de livraison et des fonctionnalités à livrer ainsi qu'à la formation de l'équipe de développement. C'est à l'occasion de cette phase que les aspects de management du risque et des coûts sont traités.

Cette phase est en général relativement courte et ne met pas plus de 10 personnes à contribution. Elle requiert pourtant des compétences variées, allant de la connaissance du marché, des utilisateurs potentiels, à des ressources techniques issues d'autres développement similaires ou utilisant les mêmes technologies.

Un travail sur l'architecture permet en outre de déterminer de quelle manière les éléments du backlog seront implémentés. Il s'agit en fait d'une part de déterminer l'architecture du système et d'autre part de débuter la conception détaillée.

Phase de "sprints"

C'est la phase de développement itératif. Elle sera décrite dans la partie suivante.

Phase de clôture

Quand l'équipe de management estime que les variables de temps, exigences, coûts et qualités concordent pour permettre le passage à une nouvelle release, ils déclarent la clôture de la release en cours. Cette phase de clôture prépare le produit pour une livraison : intégration, tests systèmes, documentation utilisateur, préparation de supports de formation, préparation de supports marketing, etc.

3.5.3 Les "sprints"

Le développement à proprement parler se fait au cours de ce que la méthode Scrum appelle des "sprints". Les sprints sont guidés par des listes de tâches à réaliser, classées par ordre de priorité : les "backlogs". Un sprint est une période de 30 jours environ durant laquelle l'équipe est isolée de toute influence extérieure, c'est à dire qu'aucun travail supplémentaire ne peut être ajouté à un sprint en cours, ni aucun modification du backlog ne peut être faite pour le sprint en cours. L'équipe est libre de déterminer le meilleur moyen de développer les fonctionnalités qui ont été retenues pour former la liste des tâches à effectuer pendant le sprint. Chaque jour, l'équipe entière participe à une réunion Scrum pour partager les connaissances acquises, faire un point sur l'avancement et donner au management une certaine visibilité sur la progression du projet.

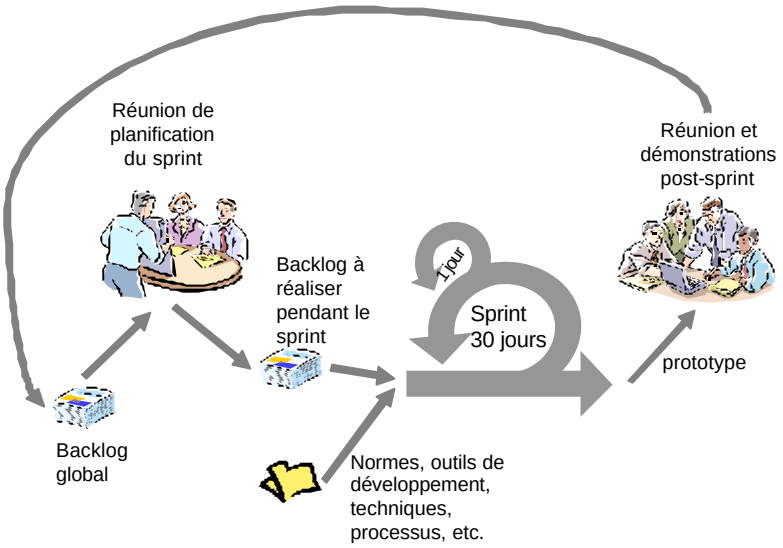


Figure 14 – Itérations de la phase de "sprints"

Backlog global

Le backlog global regroupe toutes les caractéristiques, fonctionnalités, technologies, améliorations et corrections qui constitueront la future release de l'application. Au cours d'un processus Scrum, le backlog n'est pas statique mais il évolue au fur et à mesure que l'environnement évolue. Les managers font constamment évoluer les spécifications pour s'assurer que le produit sera le plus approprié possible.

Tous les éléments qui constituent le backlog doivent être classés par ordre de priorité décroissante.

Chaque élément du backlog est estimé en termes de coût de développement, temps de développement (en jours), risque, complexité, etc. Les éléments du backlog peuvent avoir diverses origines et être dictés par des besoins du marché, des contraintes de marketing, des contraintes techniques, etc.

Sprint Backlog

Après chaque sprint, le suivant se prépare au cours d'une réunion de planification réunissant toute l'équipe. A cette occasion, développeurs et managers réunis définissent le backlog qui devra être réalisé au cours de l'itération suivante. Sur la base du backlog global qui aura été préalablement mis à jour et reclassé par ordre de priorité, l'équipe sélectionne les tâches qui seront accomplies au cours des 30 jours suivants. Si plusieurs équipes travaillent en parallèle, un backlog est établi pour chacune d'entre elles de manière cohérente. La sélection des tâches à accomplir et des fonctionnalités à développer se fait d'abord en fonction de l'ordre de priorité mais ce dernier peut être modifié pour des raisons de développement (par exemple, plusieurs tâches préalablement disjointes peuvent être rapprochées si leur développement est facilité par la mise en place d'une interface commune ou d'un module commun). Tout au long du sprint, chaque membre de l'équipe a la responsabilité de mettre à jour les éléments du backlog dont il est le propriétaire, ceci dans un souci de suivi permanent de l'avancement du projet.

Sprint

Un sprint, c'est à dire une itération de 30 jours de la phase de développement, est lui aussi subdivisé en itérations d'une journée terminées par une réunion quotidienne qui porte le nom de scrum (mêlée).

Au cours d'un sprint, l'équipe ne peut recevoir aucun changement du backlog venu de l'extérieur. Toutes les modifications qui interviennent ne prendront effet qu'à l'itération suivante. Cette disposition a pour but de protéger le travail de développement fourni par l'équipe du chaos qui règne à l'extérieur et de permettre aux développeurs de se focaliser sur un certain ensemble de fonctionnalités fixes à mettre en place. Par contre, les membres de l'équipe peuvent modifier le backlog pour atteindre l'objectif du sprint fixé au départ en termes de fonctionnalités.

Tout au long du sprint, le travail réalisé est mesuré et contrôlé de manière empirique. Chacun doit se focaliser sur les trois mêmes points : qu'est ce qui a été fait pendant la journée ? que reste-il à faire ? quels sont les obstacles qui gênent l'avancement du projet ? L'avancement du projet est évalué à travers quatre variables : coût, planning, fonctionnalités et qualité. Ces quatre variables ne sont pas indépendantes, c'est à dire qu'on ne peut pas fixer arbitrairement une valeur pour chacune d'entre elles. Grâce à cette surveillance de l'avancement, les déviations sont repérées assez tôt pour pouvoir être corrigées. Si le sprint devait demander plus de temps que prévu, on peut choisir soit d'augmenter le temps imparti, soit diminuer les fonctionnalités, soit de diminuer la qualité. En général, le coût ne peut pas subir d'augmentation pendant un sprint.

La productivité est améliorée par la petite taille des équipes (4 à 7 personnes incluant concepteurs, architectes, développeurs, responsables qualité) et par l'isolation des équipes pendant le sprint (espace de travail isolé, mais aussi perturbations extérieures éliminées). De plus, le travail collaboratif permet le partage des connaissances et du savoir-faire.

Le développement se fait de manière itérative : à l'intérieur de chaque sprint se retrouvent les activités de planification, architecture, conception, développement et implémentation.

Chaque sprint produit un livrable visible et fonctionnel (voir démonstration post-sprint)

Scrum (réunion quotidienne)

Au cours d'un sprint, une réunion quotidienne informelle de toute l'équipe a lieu, si possible toujours à la même heure et au même endroit. Ne peuvent intervenir que les personnes qui travaillent effectivement sur le développement. Les extérieurs y sont invités pour suivre l'avancement du projet mais n'interviennent pas, toujours dans ce souci de ne pas perturber le travail de l'équipe par le chaos venu de l'extérieur.

Cette réunion n'est pas sensée durer plus de 30 minutes et consiste en un tour de table où chacun doit répondre aux trois questions : qu'est ce qui a été fait depuis la veille ? que reste-il à faire ? comment le faire ?

Le manager ou Scrum Master a pour rôle de prendre les décisions que l'équipe est incapable de prendre par elle même et s'engage à apporter une solution à tout ce qui entrave le développement du projet, soit par ses propres connaissances, soit en faisant appel à des ressources extérieures.

L'utilité de telles réunions est assez évidente : elle permet d'une part, une synchronisation quotidienne de l'équipe et d'autre part, le partage des connaissances.

Réunion et démonstration post-sprint

La réunion post-sprint est l'occasion d'une part de montrer au client ce qui a été développé pendant les 30 jours précédents et d'autre part de confronter les résultats du travail de l'équipe avec la complexité et le chaos de l'environnement dans lequel l'application sera utilisée.

Cette réunion se déroule de manière informelle. L'équipe y présente ce qui a été fait, la manière dont elle a procédé pour atteindre les objectifs fixés et fait une démonstration de l'application, en l'état où elle est à la fin du sprint.

En se basant sur le backlog, c'est à dire sur ce qu'il reste à faire, sur un nouvel état des lieux du marché et des besoins des utilisateurs et sur le prototype issu du sprint précédent, cette réunion aboutit à la décision de publier ou non une release du logiciel et à la formulation des objectifs à atteindre à la fin de l'itération suivante.

3.5.4 Forces et faiblesses

Scrum présente des avantages certains par rapport aux méthodes plus traditionnelles : cette méthode permet effectivement de répondre au changement avec une certaine souplesse et permet de modifier le projet et les livrables à tout moment, de façon à livrer la release la plus appropriée possible.

D'autre part, bien que flexible, cette méthode se veut rigoureuse puisque le contrôle des processus est omniprésent. En effet, le projet est constamment

managé à la lumière des quatre variables précédemment citées que sont délais, coûts, fonctionnalités, et qualité.

Du point de vue de la progression, Scrum accorde une place importante au partage des connaissances au sein d'une équipe, notamment au cours des réunions quotidiennes. Ceci est bien entendu un facteur qui améliore la productivité et la capitalisation des savoirs.

Enfin, la pratique qui consiste à isoler l'équipe du chaos extérieur pendant le sprint autorise les développeurs à se concentrer pleinement sur les fonctionnalités qu'ils ont à développer, sans être perturbés par des changements incessants des spécifications dictés par le client ou les exigences du marché. Ainsi, les développeurs peuvent conserver toute leur énergie à concevoir les solutions les plus astucieuses pour atteindre leurs objectifs.

On peut toutefois se demander si cette pratique n'est pas en contrepartie une sorte de frein à l'agilité. En effet, si on imagine qu'une modification des spécifications est ordonnée par le client en cours de sprint, est-il nécessaire de continuer à développer en ignorant cela jusqu'au bout du sprint ? Prendre en compte les modifications au fur et à mesure de leur arrivée conduit, certes, à une baisse de productivité, mais d'un autre côté, limite la production de code voué à la destruction. Ce point particulier mérite donc d'être examiné attentivement avant de mettre en place cette méthode.

3.6 FEATURE DRIVEN DEVELOPMENT

3.6.1 Les grandes lignes de la méthode

Feature Driven Development est une méthode agile développée par Jeff de Luca et Peter Coad pour mieux gérer les risques existants dans les projets de développement. L'idée de base de la méthode est de procéder par itérations très courtes, de deux semaines, avec production à la fin de chaque itération d'un livrable fonctionnel. Les spécifications sont découpées le plus possibles en caractéristiques simples : les "features" que l'on peut regrouper en "feature sets".

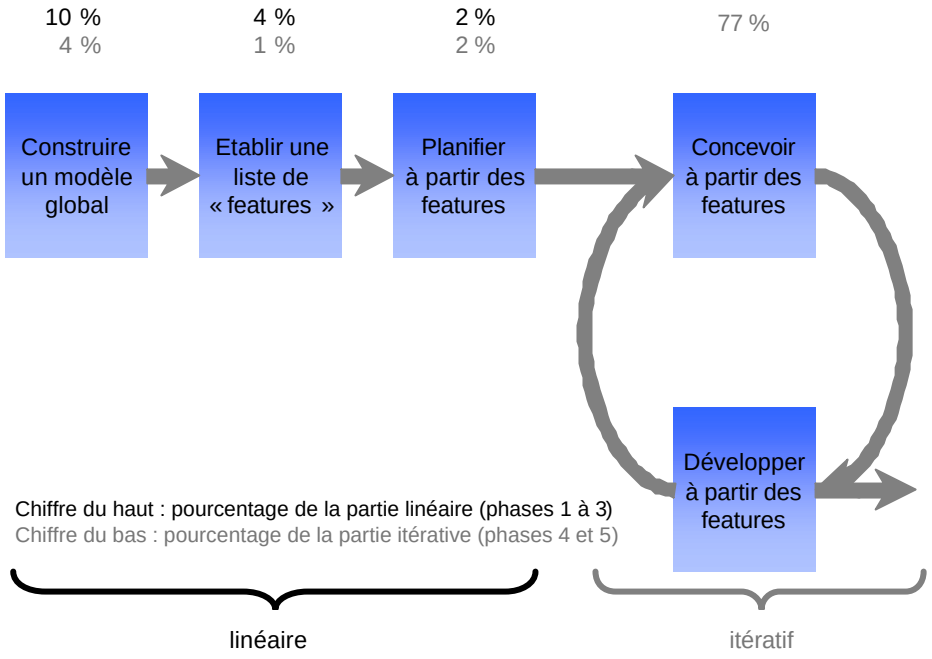
Pour les développeurs, procéder par itérations est très motivant puisqu'ils doivent être capables de livrer toutes les deux semaines des composants qui sont utiles au client, qui fonctionnent et sont porteurs de valeur pour lui.

Du point de vue des managers, cette façon de procéder permet de connaître facilement l'état d'avancement du projet et d'avoir du concret à montrer au client à chaque fin d'itération. Le fait de délivrer fréquemment des composants fonctionnels permet aussi une bonne gestion du risque.

Les clients ont une vue claire sur le planning et les différentes étapes du projet. Ils ont aussi des résultats concrets qui leur prouvent l'avancée du projet.

3.6.2 La notion de "feature" et de "feature set"

Cette notion est au centre de la méthode FDD puisque toute la conception et le développement sont faits autour des caractéristiques ("features") de l'application.



Feature désigne une fonctionnalité porteuse de valeur pour le client, susceptible d'être implémentée en deux semaines ou moins. Le formalisme utilisé pour les décrire est le plus simple possible : <action> the <result> <by, for, of, to> a(n) <object> (ex : calculate the total of a sale). Ce découpage en fonctionnalités simplissimes permet au client d'exprimer ce qu'il attend de manière très simple.

Ces features sont regroupées en groupes qui participent à une même fonction plus globale. Le formalisme utilisé pour décrire les feature sets est alors : <action><-ing> a(n) <object> (ex : making a product sale to a customer).

Au cours du développement du modèle global, une liste informelle de features est établie sur consultation de personnes qui connaissent le domaine d'application et de divers documents. Par la suite, une liste détaillée est établie. Cette façon de procéder permet de confronter des gens du métier pour développer un business model commun avant d'établir une liste détaillées de fonctionnalités. Cela permet aussi aux développeurs de connaître un peu mieux le domaine et la façon dont les choses sont liées. Cette pratique encourage la créativité, l'innovation et l'exploration et conduit à la découverte de nouvelles fonctionnalités qui apporteront un avantage concurrentiel.

3.6.3 Les 5 phases d'un projet FDD

Ce paragraphe passe en revue les différentes étapes qui structurent le projet et la façon dont elles s'enchaînent.

Figure 15 - Les 5 phases d'un projet Feature Driven Development

Phase 1 : Développer un modèle global

Critère d'entrée

Le client doit être prêt à débuter le projet. A ce stade, il peut avoir une ébau-

che de liste de besoins mais il n'est pas sensé savoir précisément quels sont ses attentes. Souvent il ne sait pas faire la différence entre les fonctionnalités dont il un besoin absolu et celles qu'il serait bien d'avoir.

Tâches		
Former l'équipe de modélisation	Managers	Obligatoire

L'équipe modélisation est constituée à temps plein de développeurs et de personnes connaissant le domaine. Il est conseillé de faire participer à tour de rôle tous les membres de l'équipe de façon à ce que chacun ait une opportunité d'observer et de participer.

Etude du domaine	Equipe modélisation	Obligatoire

Une personne du métier fait un court topo expliquant aux développeurs les caractéristiques du domaine pour lequel ils devront développer l'application.

Etude documentaire	Equipe modélisation	Optionnel

L'équipe parcourt tous les documents à sa disposition (modèles, exigences fonctionnelles, modèles de données, manuels utilisateurs, etc.)

Elaborer une liste informelle de features	Architecte Développeurs senior	Obligatoire

Cette liste informelle est une première ébauche de celle qui sera établie au cours de la phase n°2.

Développer un modèle en petits groupes	Equipe modélisation	Obligatoire

Chaque sous groupe établit un diagramme de classes pour un composant.

Développer un modèle	Equipe modélisation Architecte	Obligatoire

Chaque sous groupe présente le résultat de son travail. L'architecte peut proposer d'autres alternatives. L'ensemble de l'équipe choisit une modélisation pour base et, en reprenant ce qui a été fait, élabore un diagramme global clair et annoté.

Lister les alternatives	Equipe modélisation	Obligatoire

Pour faire référence plus tard, un des participants note toutes les alternatives qui ont été considérées lors de la modélisation.

Vérification		
Validation interne et externe	Equipe modélisation	Obligatoire

Permet de clarifier et de vérifier la bonne compréhension du domaine, des besoins en terme de fonctionnalités et l'étendue du projet.

Critère de sortie

L'équipe doit fournir les diagrammes de classes, une liste informelle de features, et de notes sur des modélisations alternatives. Tout ceci est soumis à révision et acceptation par le chef de projet de développement et l'architecte.

Phase 2 : Établir une liste détaillée de features classées par priorité

Critère d'entrée		

L'équipe chargée de la modélisation doit avoir passé avec succès la première phase.

Tâches		
Former l'équipe chargée d'établir la liste détaillée des features	Chef de projet	Obligatoire

L'équipe feature est constituée de développeurs et de personnes connaissant le domaine.

Identifier les features Former les feature sets	Equipe features	Obligatoire

Sur la base de la liste informelle établie à l'étape 1, l'équipe transforme les méthodes en features et surtout, lors de brainstormings, sélectionne et ajoute des features qui correspondent encore mieux aux besoins et demandes du client.

Les features sont regroupées en features sets et en major feature sets.

Classer les feature sets et les features par ordre de priorité	Equipe features	Obligatoire

Un sous groupe de l'équipe se charge de classer les features en 4 catégories : A (indispensable), B (intéressant à ajouter), C (à ajouter si possible), D (futur).

Eclater les features trop complexes	Equipe features	Obligatoire

Les développeurs et l'architecte sont chargés de découper les features qui ne peuvent pas être développées en moins de deux semaines en plusieurs features plus légères.

Vérification		
Validation interne et externe	Equipe features	Obligatoire

Permet de clarifier et de vérifier la bonne compréhension du domaine, des besoins en terme de fonctionnalités et l'étendue du projet.

Critère de sortie

L'équipe doit fournir une liste détaillée de features, regroupées en feature sets et classées par ordre de priorité. Celle-ci est soumise à révision et acceptation par le chef de projet de développement et l'architecte.

Phase 3 : Planifier à partir des features

Critère d'entrée		

L'étape 2 a été terminée avec succès

Tâches		
Former l'équipe planning	Chef de projet	Obligatoire

L'équipe est constituée du chef de projet développement et des dévelop-

peurs seniors.

Séquencer les features	Equipe planning	Obligatoire

L'équipe planning détermine l'ordre dans lequel les diverses fonctionnalités seront développées et fixe les dates auxquelles chacun des feature sets devra être implémenté.

Affecter des classes à leurs propriétaires	Equipe planning	Obligatoire

En utilisant comme guide la séquence et le poids des différentes features, l'équipe planning désigne les propriétaires des différentes classes.

Affecter les feature sets aux développeurs seniors	Equipe planning	Obligatoire

En utilisant comme guide la séquence et le poids des différentes features, l'équipe planning désigne les développeurs seniors propriétaires des différents feature sets.

Vérification		
Auto validation	Equipe planning	Obligatoire

Une validation externe est faite si besoin par des managers seniors.

Critère de sortie

L'équipe doit fournir un planning détaillé et les dates butoirs de chaque itération. Tout ceci est soumis à révision et acceptation par le chef de projet de développement et l'architecte.

Phase 4 : Concevoir à partir des features

Critère d'entrée

L'étape 3 a été passée avec succès.

Tâches		
Former l'équipe DBF (Design By Feature)	Développeur senior	Obligatoire

Formation d'un groupe responsable du design détaillé pour chaque feature.

Etude du domaine	Equipe DBF, domaine	Optionnel

Une personne du métier fait un court tour d'horizon expliquant aux développeurs les caractéristiques du domaine pour lequel ils devront développer l'application.

Etude des documents référencés	Equipe DBF	Optionnel

L'équipe passe en revue tous les documents se rapportant à la feature dont ils ont en charge la conception.

Construire un diagramme de séquence	Equipe DBF	Obligatoire

L'équipe construit un diagramme de séquence formel et détaillé pour la feature en question. Ce diagramme viendra s'ajouter au modèle du projet.

Etablir les prologues de classes et méthodes	Equipe DBF	Obligatoire

Chaque propriétaire de classe met à jour les types de paramètres, les types renvoyés, les exceptions, les envois de messages…

Inspection du design	Equipe DBF	Obligatoire

L'équipe feature, aidée si besoin de personnes extérieures, révise la conception qui a été faite

Lister les actions d'inspection du design	Scripte	Obligatoire

Pour chaque propriétaire de classe, les étapes de la révision sont consignées.

Vérification		
Vérification	Equipe feature	Obligatoire

Fait en interne sur la base des diagrammes de séquence.

Critère de sortie

L'équipe doit fournir le diagramme de séquence détaillé, les diagrammes de classes mis à jour ainsi qu'une trace d'autres alternatives de design intéressantes.

Phase 5 : Construire à partir des features

Critère d'entrée

L'étape 4 a été passée avec succès, c'est à dire que les features qui doivent être implémentées au cours de l'itération en cours ont été conçues..

Tâches		
Implementer classes et méthodes	Equipe Features	Obligatoire

Développement, conformément au diagramme de séquence et de classes. Des classes de tests sont ajoutées systématiquement.

Inspection du code	Equipe Features	Obligatoire

Le code est passé en revue, avant ou après les tests unitaires. Si besoin, cette tâche peut recevoir une aide extérieure.

Liste des modifications apportées au code	Scripte	Obligatoire

Pour chaque classe, les modifications sont listées et devront être mises à jour par le propriétaire de la classe.

Tests unitaires	Equipe Features	Obligatoire

Chaque propriétaire de classe teste son propre code.

Préparation pour l'intégration	Equipe DBF	Obligatoire

Une fois toutes les classes testées, on procède à leur intégration.

Vérification

Inspection du code et tests unitaires	Equipe feature	Obligatoire
Critère de sortie		
L'équipe doit délivrer un composant fonctionnel, conformément aux spécifications.		

3.6.4 Forces et faiblesses

FDD présente les avantages des méthodes agiles, à savoir gestion des risques, flexibilité par rapport au changement, rapidité, livraisons fréquentes, etc.

FDD a déjà été utilisé avec des équipes de taille conséquente pouvant aller jusqu'à une vingtaine de développeurs. Le facteur limitant cependant la taille d'une équipe est le nombre de développeurs seniors à disposition.

La propriété du code revenant aux propriétaires de classes a l'avantage de ne permettre des modifications que par une personne qui a une vision globale d'une classe. On peut opposer à cela le principe de propriété collective du code en vigueur dans eXtreme Programming qui nécessite une plus grande discipline mais peut s'avérer plus rapide si un développeur a besoin de modifier du code écrit par un autre membre de l'équipe.

L'inspection du code permet d'apporter un regard neuf sur chaque portion du code et permet ainsi de produire un code de meilleur qualité. Cependant, la personne qui inspecte le code ne le connaît pas a priori et il lui faut donc un minimum de temps pour se familiariser avec ce code. Encore une fois, il peut s'avérer plus judicieux de passer à la programmation en binôme pour éviter cette perte de temps et faire la révision du code en même temps que son écriture. FDD n'est d'ailleurs par fermée à ce genre de pratiques qui poussent l'agilité encore plus loin.

3.7 RECAPITULATIF : COMPARATIF DES METHODES AGILES

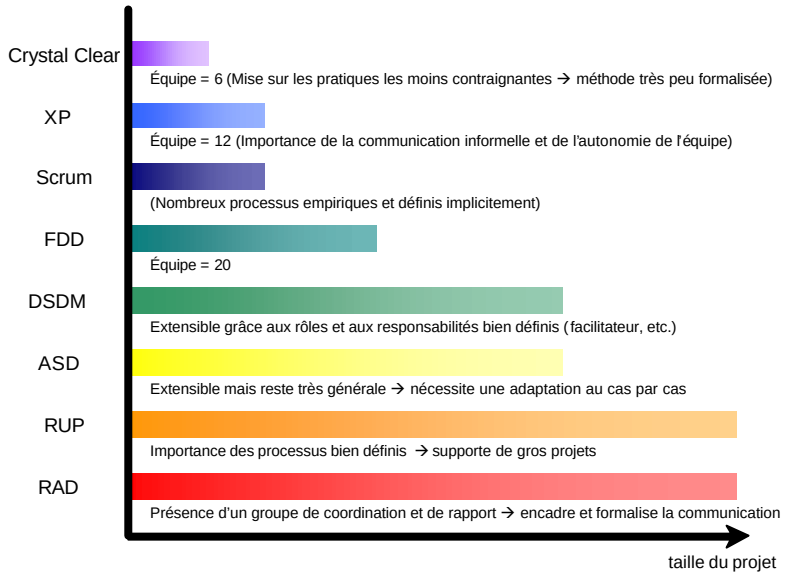
3.7.1 Adaptation des méthodes à la taille des projets et des équipes

Toutes les méthodes mentionnées dans les paragraphes précédents ne sont pas applicables dans toutes les situations. En effet, selon qu'elles prévoient des rôles bien définis, notamment des rôles chargés de coordonner et de faciliter la communication ou qu'elles laissent au contraire l'équipe assez autonome, ces méthodes seront plus ou moins efficaces en fonction de la taille de l'équipe et donc de la taille du projet.

eXtreme Programming conseille de limiter la taille des équipes à une douzaine de personnes, ceci pour plusieurs raisons. Premièrement, XP laisse aux développeurs une très forte dose d'autonomie et de responsabilités. Si cela se révèle particulièrement productif avec une équipe de taille réduite, dès que la taille de l'équipe augmente, une telle autonomie devient au contraire un frein. En outre, la communication est une valeur clé d'XP. Elle se fait de manière informelle mais devient beaucoup plus délicate lorsque l'équipe s'agrandit. Notons aussi que la pratique de la programmation en binôme et surtout la rotation des binômes nécessite que tous les membres

de l'équipe s'entendent bien et se connaissent bien ce qui est d'autant plus improbable que la taille de l'équipe est grande.

DSDM de son côté est plus facilement adaptable à des équipes plus importantes, grâce à une définition très précise des rôles et des responsabilités. Qui plus est, la méthode DSDM prévoit, en plus des rôles de sponsor exécutif et de chef de projet, un facilitateur et un rapporteur qui se chargent d'organiser et de formaliser la communication au sein de l'équipe (organisation et



animation des ateliers, synthèse, capitalisation et diffusion de l'information).

Adaptive Sofware Development peut aussi bien être utilisée sur de petits projets que sur de gros projets, à condition d'être correctement adaptée. En effet, plus que des processus bien définis, ASD véhicule des principes et des idées directrices qui restent valables sur de larges échelles. Ensuite, il faut bien évidemment définir la manière dont la méthode sera mise en œuvre de manière plus ou moins précise et formelle.

Crystal Clear est certainement la moins formalisée des méthodes passées en revue dans cette étude. Pour cela, elle ne s'adapte qu'aux petites équipes (moins de 6 personnes). Par contre, il existe d'autres méthodes non détaillées ici appartenant à la famille des méthodes Crystal qui permettent de s'adapter à de plus vastes projets.

Scrum n'est applicable que pour des petits projets, ceci parce que les procédures et les interactions qui ont lieu dans le cadre d'un sprint de 30 jours sont très implicites et empiriques. Si le cadre général du cycle de vie est assez précis, il règne une certaine liberté à l'intérieur même de chaque phase ce qui rend difficile l'utilisation de cette méthode dans le cadre d'équipe de grande taille.

Quant au RAD, en raison de la présence du groupe d'animation et de rapport et de la possibilité de paralléliser les tâches (c'est à dire de former plusieurs petites équipes travaillant chacune sur un point précis), il peut s'appliquer aussi bien à des petits projets qu'à de plus importants.

Figure 16 - Adaptabilité des méthodes à la taille des projets

3.7.2 Degré d'exigence des méthodes agiles pour le client

Toutes les méthodes agiles étudiées dans ce document présentent des similitudes au niveau de la relation entre le client et l'équipe de développement.

Bien souvent, avec les méthodes traditionnelles, le client n'était en contact qu'avec le chef de projet qui se chargeait, avec le client de collecter, clarifier et synthétiser les exigences pour établir les spécifications. Avec les nouvelles pratiques des méthodes agiles, le client entre en relation non seulement avec le chef de projet, mais aussi avec le développeurs. Une des caractéristiques communes des méthodologies agiles est de renforcer la proximité entre le client et l'équipe de développement. En effet, le client est amené à travailler directement avec les développeurs à la fois pour l'expression de ses exigences et pour le suivi de l'avancement à la fin de chaque itération. En rentrant plus dans les détails, les processus mis en œuvre dans chacune des méthodes diffèrent légèrement mais le principe reste le même : dans tous les cas le client met ses exigences et ses besoins par écrit sous une forme assez sommaire qui décrit les interactions entre l'utilisateur et le système puis les précisions se font au fur et à mesure de l'avancement du projet. Dans tous les cas aussi, le feedback du client vers les développeurs est très important puisque toutes les méthodologies mettent en place des pratiques permettant, à la fin de chaque itération courte, de faire le point, de recadrer et repréciser les besoins.

Ensuite, au niveau de la mise en œuvre, chaque méthode propose ses propres pratiques avec un degré d'implication plus ou moins fort du client.

eXtreme Programming, par exemple, est la méthode qui met en place les relations les plus étroites entre le client et les collaborateurs du projet puisque le client est intégré à l'équipe. Il prend en charge l'expression de ses besoins sous la forme de user stories, établit ses priorités, participe au planning game, conçoit et met en œuvre les tests fonctionnels. Le client prend donc part activement dans le projet, ceci d'autant plus que sa présence est requise sur site à plein temps tout au long du projet.

DSDM insiste plus sur la notion d'utilisateur que sur la notion de client. En réalité, il s'agit juste de mettre en lumière le fait que le client doit apporter une vision d'utilisateur par son implication dans le projet. L'implication active des utilisateurs est essentielle pour éviter des retards dans la prise de décision. A ce propos, DSDM préconise la simplification des procédures de gestion des modifications et de prise de décision pour les rendre moins contraignantes et plus souples. Le rôle du client dans le projet est important : il ne se contente pas de fournir les informations nécessaires au déroulement du projet, il prend en charge des mini tests de recette, s'assure de la qualité de la documentation utilisateur et de la formation des utilisateurs n'ayant pas participé au projet.

Pour Adaptive Software Development, le client intervient sur la phase d'initialisation pour la définition des spécifications. Il participe aussi, à la fin de chaque itération aux réunions des "customer focus groups" qui ont pour but d'explorer une version fonctionnelle de l'application et de lister les demandes de modifications émanant du client. ASD, ne définit pas de tâches aussi précises pour le client qu'eXtreme Programming ou DSDM, ce qui n'empêche

pas que son implication reste essentielle et permanente du fait de la progression itérative.

Dans un projet Crystal, la communication et la collaboration entre les développeurs et le client / utilisateur est primordiale, et se fait de manière très peu formalisée. Le client exprime ses besoins sous forme de cas d'utilisation sommaires, toutes les précisions étant apportées par la suite au fur et à mesure des itérations. Au cours de chaque itération, des démonstrations sont faites au client pour lui permettre de recadrer les spécifications du projet.

L'intervention du client dans un projet Scrum est beaucoup plus ponctuelle. Il intervient pour la définition des exigences dans la phase d'initiation puis participe tous les 30 jours aux réunions et démonstrations post-sprints pour suivre le projet et faire part des modifications éventuelles à apporter. Son intervention apparaît beaucoup moins permanente et continue que dans les autres méthodes, puisque personne ne peut modifier les spécifications pendant un sprint, ceci pour préserver l'équipe des développeurs du chaos qui règne à l'extérieur.

Enfin, Feature Driven Development procède sensiblement de la même manière. Le client exprime ses besoins non pas sous forme de cas d'utilisation, mais sous forme de features, et toutes les deux semaines, un point est fait sur l'avancement. Étant donné que ce passage en revue a lieu tous les quinze jours, il permet un bon feedback du client et une bonne souplesse par rapport à des changements de spécifications.

Pour résumer, les méthodes agiles misent toutes sur une implication forte du client. La clé du succès réside dans une présence fréquente et régulière, voire quasi permanente du client auprès des développeurs. Chaque méthode a ensuite ses propres pratiques plus ou moins formalisées et laisse plus ou moins de tâches à la charge du client.

3.7.3 Facilité de mise en œuvre et contraintes en interne

Les différentes méthodes que nous avons passées en revue présentent toutes des contraintes plus ou moins fortes qui font qu'elles sont plus ou moins faciles à mettre en œuvre.

eXtreme Programming est généralement reconnue comme étant la méthode présentant les pratiques les plus exigeantes. XP demande de la part des développeurs beaucoup de courage : courage d'accepter le changement, de prendre ses responsabilités, d'être autonome, de s'astreindre à un processus de tests contraignant. Les ingénieurs de développement doivent aussi accepter de programmer en binôme ce qui n'est pas une pratique naturelle pour tous et s'astreindre à reprendre tous les jours le code qu'ils ont écrit pour le rendre plus propre, même s'il fonctionne déjà. En terme de compétences, les ingénieurs de développement se doivent d'être polyvalents puisqu'ils doivent avoir la double compétence de développeur et de concepteur. Le rôle du coach est aussi assez difficile à tenir dans un projet XP. En effet, son rôle est important puisqu'il porte la responsabilité globale du projet mais la difficulté réside dans le fait qu'il doit se montrer le moins intrusif possible et savoir donner progressivement de l'autonomie à l'équipe.

La méthode DSDM est, elle aussi, assez exigeante en termes de mise en œuvre. L'équipe constituée de développeurs et d'utilisateurs se voit confiée une certaine autonomie puisqu'elle doit être autorisée à prendre des décisions, notamment concernant la modification des spécifications. Comme dans XP les tests sont présents à tous les stades du cycle de vie ce qui est contraignant. DSDM nécessite aussi des compétences particulières comme celle de médiateur, chargé de l'animation des ateliers facilités au cours de la phase d'Etude du Business. DSDM prévoit des rôles définis très précisément (sponsor exécutif, visionnaire, facilitateur, rapporteur, etc.) et nécessitant certaines compétences précises ce qui joue en faveur de la méthode mais en contrepartie rend plus contraignante sa mise en œuvre.

Pour mettre en œuvre Adaptive Software Development, seule la phase d'apprentissage nécessite un état d'esprit particulier. Par ailleurs, le cycle de vie d'ASD est bien défini tout en laissant à l'équipe une grande souplesse d'organisation et les exigences en termes de compétences et de rôles particuliers sont bien moindres que dans les deux cas précédemment cités.

Crystal Clear insiste sur le caractère crucial de la communication et impose que l'ensemble de l'équipe soit localisée sur un même lieu. Hormis cette contrainte d'organisation, l'équipe reste relativement libre de s'organiser comme bon lui semble. En termes de processus, cette méthode est contraignante en raison de l'omniprésence des tests et du refactoring du code que doivent s'imposer les développeurs.

Scrum est une méthode assez peu contraignante en terme de processus puisque dans le cadre du sprint, beaucoup de procédures sont empiriques.

Feature Driven Development présente des processus très clairement définis, avec des phases explicitant clairement les tâches élémentaires à réaliser et des indicateurs bien définis permettant de passer d'une phase à une autre. De même, les rôles sont définis de manière précise (class owner, chief programmer, etc). FDD est donc relativement contraignante à mettre en place.

A partir de cette comparaison, il est possible de resituer qualitativement sur un graphique les méthodes les unes par rapport aux autres en terme de degré d'exigence en interne et de difficulté de mise en œuvre. La même représentation peut être faite pour classer les méthodes en fonctions de leur degré d'exigence par rapport au client.

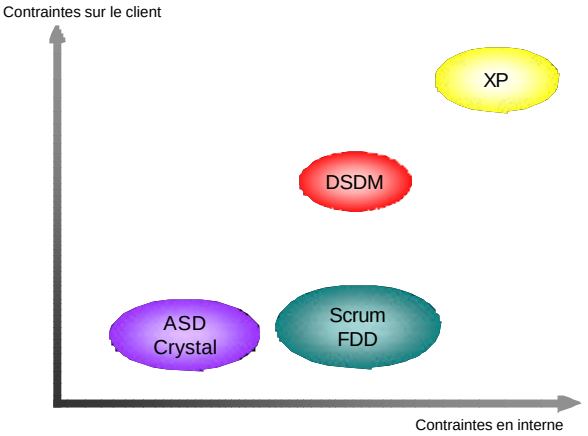


Figure 17 - Positionnement des méthodes agiles en fonction des contraintes de mise en œuvre

Il est difficile d'évaluer ensuite l'efficacité de chacune de ces méthodes. Si eXtreme Programming est actuellement la méthode qui monte en force, il faut bien garder à l'esprit qu'il n'existe stricto sensu ni bonne ni mauvaise méthode. Il est impossible de porter une méthode au rang de meilleure méthode agile, pour la simple et bonne raison que la réussite d'un projet dépend avant tout de l'adaptation de la méthode au contexte.

4. LES METHODES AGILES VUES PAR BUSINESS INTERACTIF

Les méthodes agiles présentées dans ce document appellent naturellement à réaction. Nous avons souhaité dans cette section vous faire part de notre retour sur le sujet, de nos expériences. Plus que tout, c'est le terrain qui est le véritable juge.

4.1 LES METHODES AGILES SONT ELLES VRAIMENT NOUVELLES ?

Les méthodes agiles introduisent en fait un certain nombre de concepts qui sont pour beaucoup du bon sens en action. Bon nombre d'équipes mettent déjà en œuvre une partie de ces principes, pas nécessairement de manière « extreme », mais avec des résultats sensibles. L'importance des tests, la nécessité de travailler en itérations courtes, le feedback client sont des éléments clés dans la réussite d'un projet. Les équipes mettent souvent en œuvre un certain nombre de principes « agiles » sans même le savoir, un peu à la manière de Mr Jourdain.

D'autres principes, plus extrêmes, comme le pair programming, constituent par contre un saut quantique important.

4.2 LA BONNE METHODE POUR LE BON PROJET

Il nous semble essentiel de ne pas s'enfermer dans une méthode, qu'elle soit agile ou non. L'expérience montre que l'on ne peut calquer un schéma identique sur l'ensemble des projets. La typologie des clients, la compétence et l'expérience des équipes, la nature de la relation entre la maîtrise d'ouvrage et la maîtrise d'œuvre sont des paramètres qui vont considérablement influer sur le plan assurance qualité qui va être mis en œuvre. Cela ne veut pas dire qu'il ne faut pas de méthode, bien au contraire. Simplement le cadre méthodologique doit être adapté à chaque contexte. Difficile de mettre en place Unified Process et UML sur des équipes Junior avec un projet informatique de gestion en VB, par exemple.

Les méthodes agiles obéissent à la règle. Adaptées à certains contextes, elles sont en revanche à proscrire dans d'autres. Nous restons persuadés à Business Interactif qu'un facteur de succès dans la mise en œuvre de ces méthodes est l'expérience forte de l'équipe, et notamment sa connaissance de méthodes dites «traditionnelles ». Sinon il y a fort à parier que le projet parte directement dans le mur. Extreme Programming, en particulier. Le niveau de responsabilité des développeurs nécessite une grande maturité. Même si le pair-programming permet de mettre dans la boucle un débutant en binôme avec un expérimenté, la présence de «vieux brisquards » est indispensable.

Nous préconisons une adaptation du plan assurance qualité générique en choisissant les «best practices » issues des différentes méthodes pour se constituer un cadre adapté aux projets mis en œuvre.

4.2.1 Un exemple : la modélisation de données

L'exemple le plus frappant concerne la modélisation de données. Sur des projets d'informatique de gestion, la modélisation de données Entités-Relations (pour ne pas dire Merisienne) est à notre avis incontournable. C'est un facteur clé de succès, pour la réalisation du projet mais aussi pour sa maintenance. Le risque de partir sur des méthodes complètement novatrices, appuyées sur UML ou même agiles (ces dernières préconisant de voyager « léger », et notamment de minimiser la documentation) est d'abandonner ce type de modélisation au profit unique de diagrammes UML ou autres. Seulement voilà : les outils gérant la persistance objet sont encore inadaptés aux grands projets d'informatique de gestion et peu répandus ; le risque d'échec est alors non négligeable.

Lors de son recrutement chez Business Interactif, chaque candidat passe quelques tests de modélisation de données. Soyons clairs : les résultats des tests sont souvent «extrêmes » dans leur faiblesse. Y compris chez des personnes qui pourtant pratiquent le développement d'applications de gestion depuis plusieurs années…inquiétant, et pourtant révélateurs d'un rejet de méthodes qui ont fait leurs preuves.

4.3 QUELLES SONT LES MAUVAISES METHODES ?

Les méthodes agiles ont vu le jour en réaction à de grands échecs de projets, souvent attribués aux méthodes qui avaient été mises en œuvre.

Nous sommes persuadés qu'il n'y a pas de mauvaises méthodes. Comme nous le disions, les échecs sont souvent liés au choix d'une méthode inadaptée à un contexte, ou a à une mauvaise mise en œuvre de la méthode : insuffisance de la formation, inexpérience de l'équipe.

4.4 QUELQUES POINTS CLES CROSS METHODES

Au delà de la méthode sélectionnée, il nous a semblé important de remonter quelques points clés dictés par l'expérience des projets, qui restent valables pratiquement dans tous les cas.

4.4.1 Petites équipes et outils véloces

Ce sont les petites équipes qui font les grands projets. Un exemple frappant a retenu notre attention dès la création de notre société. Nous sommes issus d'une culture C++ très forte, et à l'époque nous travaillions (comme beaucoup de développeurs C++) avec Borland C++ (devenu depuis C++ Builder). L'équipe qui réalisait le compilateur chez Borland était constituée d'une équipe de plus de 30 développeurs, tous reconnus comme des gourous du C++. Borland a sorti à ce moment de son chapeau Delphi, un outil de développement révolutionnaire, alliant la puissance du C++, la facilité de VB, avec la qualité syntaxique et objet du Pascal Objet. Delphi a été réalisé en un temps record par une équipe de 7 développeurs (le père de Delphi est aujourd'hui le papa de C#), prenant de court l'équipe C++ sur tous les fronts. Delphi a été écrit d'abord en C++, puis rapidement les développeurs Borland ont utilisé Delphi lui même pour écrire Delphi. Cet événement est resté clé dans notre histoire. Outre le fait que nous avons à l'époque adopté Delphi, nous avons compris que des équipes de 5-6 personnes armées des bons

outils, avec la bonne formation et la motivation pouvaient rivaliser avec des équipes de 20 personnes. Car en informatique 1+1 n'ont jamais fait 2. Depuis, nos projets, à l'image d'OOSHOP, se sont toujours appuyés sur des équipes qui tenaient plus des « voltigeurs » que des « artilleurs ». Nous restons persuadés qu'il s'agit d'un facteur clé de succès dans les projets, même si cela implique de découper les grands projets en plusieurs sous-projets.

4.4.2 Polyvalence des développeurs

La polyvalence des développeurs est également un point clé. Même si chacun, de par ses affinités et son expérience a un domaine de prédilection, le succès des projets passe par la polyvalence des développeurs. Front-office, back-office, développement deux-tiers, trois-tiers, delphi, java, VB, C#…Un bon développeur est un développeur qui s'adapte rapidement à tous les contextes, et qui accepte ces changements avec enthousiasme.

Cela ne veut pas dire qu'il faille survoler tout sans rentrer en profondeur. C'est malheureusement un symptôme trop fréquent chez les développeurs : deux mois passés sur un langage et l'impression d'être un expert…les pros savent que même après 10 ans passés sur un outil ou un langage on en découvre encore des facettes inexplorées. Mais il est important de tourner, de voir d'autres contextes, des technologies différentes…pour finalement passer à un niveau « meta » où les patterns cross-outils sont tellement maîtrisés que le temps d'adaptation à un nouveau contexte se fait en deux trois jours.

Polyvalence des développeurs signifie-t-elle propriété collective du code, comme le prône XP ? Cela reste un sujet d'interrogation. Nous pensons qu'il est essentiel qu'un développeur puisse comprendre le code d'un autre développeur, même si c'est sur un langage différent. En revanche la modification du code écrit par un autre doit être prise avec précaution. Des effets de bord dus à une maîtrise insuffisante du contexte sont fréquents.

Aujourd'hui cette polyvalence est insuffisamment pratiquée par les entreprises, alors qu'elle est essentielle.

4.4.3 Feedback client et itérations

Le concept d'itérations courtes est essentiel. L'effet tunnel est à bannir. Un projet, même un grand chantier, doit aboutir rapidement à des livrables sur lesquels le client peut donner un feed-back. Maquettes, prototypes, modules : une livraison étagée est souvent clé.

4.4.4 Gestion des risques

La gestion des risques est un point clé du plan assurance qualité. Le chef de projet doit avoir l'œil rivé sur les risques, en permanence. En particulier les risques en termes d'architecture et d'intégration, qui doivent être traités de manière prioritaire : inutile de partir bille en tête sur des parties maîtrisées si des risques d'architecture globale ne sont pas cernés et éliminés.

4.4.5 Maîtrise d'ouvrage et maîtrise d'œuvre jouent dans le même camp

Les relations entre la maîtrise d'ouvrage et la maîtrise d'œuvre sont essentielles. Ce sont souvent elles qui conditionnent le succès d'un projet. Problème : les deux parties ne jouent pas toujours dans le même camps semble-

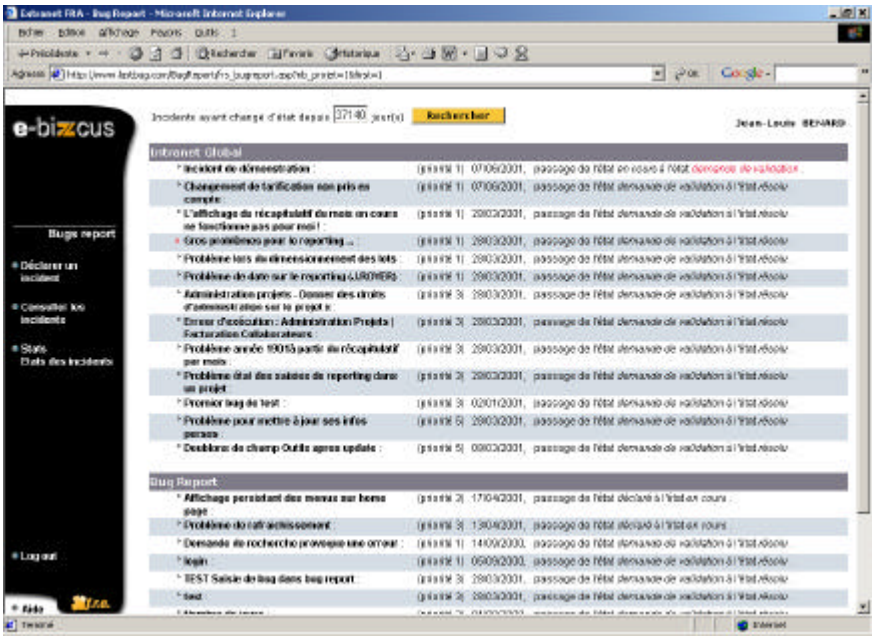
t-il. Seul un partenariat gagnant-gagnant et une réelle confiance entre les deux parties garantissent un avancement sûr des travaux.

Les contrats forfaitaires ne jouent pas forcément en faveur d'une mise en œuvre des méthodes agiles. Le forfait présuppose un accord préalable sur ce qui doit être réalisé, alors que les méthodes agiles, XP notamment, sont pour une redéfinition possible du périmètre : « le client a le droit de modifier ses exigences »…pas simple dans un cadre forfaitaire, ou chacun est en train de vérifier si l'autre ne déborde pas de son périmètre. Lorsqu'une réelle confiance s'instaure entre la maîtrise d'ouvrage et la maîtrise d'œuvre, il est possible de tisser des liens étroits entre client et équipe de développement : un objectif global à atteindre est défini pour une date donnée, et une équipe « projet » travaille pour atteindre cet objectif global, sur des modalités à mi-chemin entre le forfait et l'assistance technique.

4.4.6 Tester toujours et encore

Les méthodes agiles insistent sur un point qui est effectivement fondamental : le test. Tests unitaires, tests d'intégration, tests de performance : le test doit être une véritable culture, un exercice permanent. Que les tests soient écrits avant le développement (comme le préconise XP) ou non, que l'on s'appuie sur un framework de test de type Junit ou non, le test est incontournable. Malheureusement, des calendriers trop serrés et aussi une trop grande confiance se soldent souvent par une insuffisance du test. En avoir conscience, c'est déjà parcourir la moitié du chemin.

Dans le même registre, la prise en compte rapide des incidents est indispensable. Lorsque nous avons mis en place un Extranet clients permettant d'effectuer du reporting d'incident, clients et développeurs y ont trouvé leur compte : incidents formalisés et prise en compte « industrielle » des incidents accélèrent le processus d'aboutissement du projet.



L'extranet client de Business Interactif permet aux client de suivre en temps réel le traitement des incidents en cours.

4.5 LA CULTURE ET LES VALEURS

Indépendamment des méthodes mises en œuvre, la culture d'entreprise et la philosophie dans laquelle travaillent les équipes est probablement le point essentiel. Les chantiers informatiques sont des chantiers d'hommes, et une mise en œuvre appliquée et besogneuse de recettes de cuisine ne sert à rien si elle ne s'insère dans un mouvement plus global, qui est au cœur de la culture d'entreprise. A chacun de construire la sienne.

Chez Business Interactif, voici quelques valeurs clé qui sous-tendent la réalisation de nos projets. Nous restons persuadés, depuis la création de nos premières applications Internet en 1994, que ces valeurs sont fondamentales dans la réussite des projets.

4.5.1 La culture de l'humilité

Démarrer un projet avec des chevilles un peu trop musclées et une confiance aveugle en soi sont des facteurs d'échec. L'humilité, la remise en cause permanente de ses méthodes, de son savoir sont essentielles pour avancer.

4.5.2 Le pragmatisme

Faire simple, toujours simple. Les usines à gaz ne mettent jamais très longtemps à exploser. Le refactoring est un travail permanent, que ce soit au niveau du code ou des architectures. A ce titre, une méthode trop formaliste finit par nuire à la simplicité et à la réactivité. « Voyagez léger » : voilà probablement l'une des meilleures pratiques d'Extreme Programming.

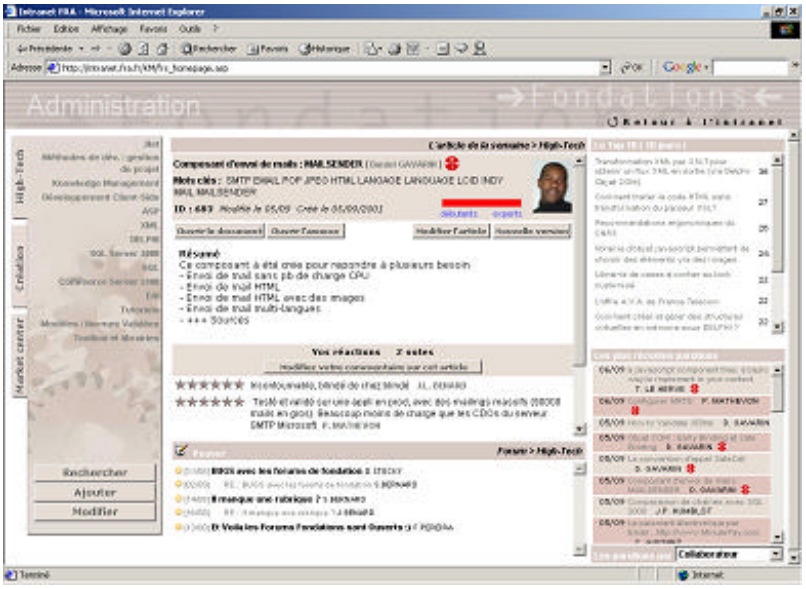
4.5.3 Le souci de la qualité

Un projet n'aboutit pas si l'équipe toute entière ne maintient pas un niveau d'exigence très élevé vis à vis de ses travaux : a-t-on fait vraiment ce qu'on pouvait faire de mieux ? Le mieux n'étant pas nécessairement la solution le plus belle techniquement parlant mais la plus optimale. Cela demande beaucoup d'énergie et beaucoup de motivation, en particulier lorsque le contexte devient difficile (délais tendus, relation complexe et politique entre la maîtrise d'œuvre et la maîtrise d'ouvrage…).

4.5.4 Le partage

Partager les connaissances est essentiel. Méthodes de développement, incidents rencontrés, le temps perdu a redécouvrir des éléments déjà maîtrisés doit être minimisé. La mise en place d'une plate-forme de gestion des connaissances à destination des équipes techniques est de ce fait capitale.

Au delà du partage des connaissance, une communication permanente et souvent informelle est indispensable entre tous les membres d'une équipe. Cela implique une culture forte du partage.



L'intranet de Business Interactif permet aux équipes de partager en temps réel leurs connaissances sur l'ensemble des technologies et des méthodes mises en œuvre.

4.5.5 Le courage

Le développement, c'est comme le bâtiment. La tâche est souvent difficile et ingrate, frustrante. Seul le résultat transcende le travail accompli. Bref, il faut du courage, pour tenir sur la durée, et accepter de faire des choses pas toujours drôles. Tous les développeurs le disent : c'est dans les moments difficiles que le rythme d'apprentissage est le plus rapide, encore faut il avoir un minimum de volonté pour affronter les difficultés.

4.6 CONCLUSION

Les méthodes agiles ne sont probablement pas le dernier soubresaut qui agite les monde de la gestion de projet. A l'image des technologies, les approches méthodologiques obéissent à des cycles. Les méthodes agiles s'inscrivent dans une phase de réaction à un formalisme trop poussé.

Faut-il pour autant rejeter les méthodes traditionnelles ? Rien n'est moins sûr. La meilleure méthode reste celle qui est le mieux adaptée au contexte. Merise, Unified Process, Extreme Programming : chacune apporte un regard différent sur la gestion de projet et le développement.

Toutes, en tout cas, convergent vers deux principes : celui de la modularité (les méthodes sont des boîtes à outils) et celui de la complémentarité (les méthodes ne sont plus systémiques et peuvent travailler de concert). A chacun de sélectionner les meilleurs outils…Voilà finalement l'un des enseignements que l'on peut tirer des méthodes agiles.

4.7 CHOISIR ET METTRE EN ŒUVRE LES METHODES

Business Interactif propose des formations sur la mise en œuvre des méthodes, que ce soit dans des contextes traditionnels ou agiles. N'hésitez pas à prendre contact avec nous sur le sujet.

Business Interactif intervient également dans l'accompagnement des Directions des Systèmes d'Information sur les aspects méthodologiques : quelle méthode choisir pour quelle projet ? Comment peut-on mettre en place un Plan Assurance Qualité adapté à un contexte précis ? Comment améliorer la communication au sein des équipes de développement ? Autant de sujets sur lesquels Business Interactif peut apporter un éclairage, souvent par la mise en œuvre et l'animation de sessions de quelques jours regroupant des acteurs clés dans l'entreprise. Contactez nous pour en savoir plus.

4.8 COLLABORER AVEC BUSINESS INTERACTIF

Ce white paper vous a donné envie de collaborer avec Business Interactif ? N'hésitez pas à nous contacter….

Contact commerciaux :

Alban Neveux – alban.neveux@businessinteractif.fr

Pierre-Edouard de Leusse – pdl@businessinteractif.fr

Ressources Humaines :

Hélène Helffer – recrutement@businessinteractif.fr