Jean-Louis **Bénard** • Laurent **Bossavit**
Régis **Medina** • Dominic **Williams**

Gestion de projet eXtreme Programming

EYROLLES

Gestion de projet eXtreme Programming

Tests unitaires, Refactoring, Intégration, Gestion de projet, ROI, Aspects contractuels

Comme toute méthode de développement, l'Extreme Programming propose un cadre pour l'ensemble des aspects du projet logiciel, de l'analyse des besoins aux tests en passant par la conception. Mais à la différence de processus prédictifs, XP ne se fonde pas sur une définition exhaustive et précoce des besoins. En découle une redéfinition de la relation entre clients et fournisseurs, avec de surprenants résultats en termes de qualité de code, de délais... et de retour sur investissement !

Diplômé de l'École Centrale Paris, fondateur de la SSII F.R.A. puis directeur technique de Business Interactif, **Jean-Louis Bénard** est président de Brainsonic et enseignant à l'École Centrale. Il accompagne régulièrement de grandes entreprises françaises dans des projets d'architecture du système d'information et dans la mise en place de méthodes de développement pragmatiques.

Laurent Bossavit est consultant et formateur indépendant. Ses quinze années d'expérience en tant que développeur, chef de projet ou architecte, l'ont amené à accompagner des entreprises, PME éditrices de logiciels ou grands comptes industriels ou financiers, lors de leur adoption d'XP. Conférencier, écrivain et enseignant, il contribue à l'évolution et à l'adoption de l'Extreme Programming.

Régis Medina intervient en tant qu'ingénieur indépendant sur des projets de développement en technologies objet. Pionnier de l'Extreme Programming en France, il en applique les principes dès 2000 en tant que chef de projet XP et met en évidence ses avantages dans le cadre de grands projets de télécommunications.

Dominic Williams développe depuis plus de dix ans des logiciels dans des domaines allant de l'océanographie à la gestion de biens en passant par la signalisation ferroviaire. Il pratique la méthode XP depuis 1999 et lui attribue ses plus belles réussites : des logiciels de qualité livrés plus vite par des équipes motivées et efficaces. Il promeut XP auprès des entreprises et grâce au projet Open Source XP Dojo.

architecte logiciel

Quelles règles pour la création logicielle ?
Quelles méthodes, quels outils ?
L'enjeu est de taille : garantir la souplesse et l'interopérabilité des applications métier.

Au sommaire

Conception : Nord Compo

EYROLLES

Code éditeur : G11561
ISBN : 2-212-11561-X

Gestion de projet eXtreme Programming

CHEZ LE MÊME ÉDITEUR

Dans la collection Architecte logiciel

P. ROQUES, F. VALLÉE. – **UML 2 en action**. *De l'analyse des besoins à la conception J2EE.*
N°11462, 3e édition, 2004, 380 pages + poster.

Modélisation et méthodes logicielles, programmation objet

P. ROQUES. – **UML 2 par la pratique, 3e édition**. *Cours et exercices.*
N°11480, 2004, 340 pages.

P. ROQUES. – **UML : modéliser un site e-commerce**.
N°11070, 2002, 168 pages (coll. *Cahiers du programmeur*).

D. CARLSON. – **Modélisation d'applications XML avec UML**.
N°9297, 2002, 354 pages.

A. COCKBURN. – **Rédiger des cas d'utilisation efficaces**.
N°9288, 2001, 320 pages.

H. BERSINI, I. WELLESZ. – **L'orienté objet**. *Cours et exercices en UML 2 avec Java, Python, C# et C++.*
N°11538, 2004, 520 pages

B. MEYER. – **Conception et programmation orientées objet**.
N°9111, 2000, 1223 pages.

I. JACOBSON, G. BOOCH, J. RUMBAUGH. – **Le Processus unifié de développement logiciel**.
N°9142, 2000, 487 pages.

R. PAWLAK, J.-P. RETAILLÉ, L. SEINTURIER. – **Programmation orientée aspects pour Java/J2EE**.
N°11408, 2004, 460 pages.

Développement Java et .NET

L. MAESANO, C. BERNARD, X. LEGALLES. – **Services Web en J2EE et .Net**.
N°11067, 2003, 1088 pages.

T. PETILLON. – **ASP.NET**. (coll. *Cahiers du programmeur*)
N°11210, 2003, 200 pages.

E. PUYBARET. – **Java 1.4 et 5.0**. (coll. *Cahiers du programmeur*)
N°11478, 2004, 400 pages

J. MOLIÈRE. – **Conception et déploiement Java/J2EE**.
N°11194, octobre 2003, 192 pages.

P.-Y. SAUMONT. – **Le Guide du développeur Java 2**.
Meilleures pratiques de programmation avec Ant, JUnit et les design patterns.
N°11275, 2003, 800 pages + CD-Rom.

K. DJAFAAR. – **Développement J2EE avec Eclipse et WSAD**.
N°11285, 2004, 640 pages + 2 CD-Rom.

L. DERUELLE. – **Développement Java/J2EE avec Jbuilder**.
N°11346, 2004, 726 pages avec CD-Rom.

Jean-Louis **Bénard** • Laurent **Bossavit**
Régis **Medina** • Dominic **Williams**

Gestion de projet eXtreme Programming

EYROLLES

ÉDITIONS EYROLLES
61, bd Saint-Germain
75240 Paris CEDEX 05
www.editions-eyrolles.com

Cet ouvrage a fait l'objet d'un reconditionnement à l'occasion de son deuxième tirage (nouvelle couverture). Le texte de l'ouvrage reste inchangé par rapport au tirage précédent.

Remerciements particuliers à Pierre Tran et Thierry Cros pour avoir favorisé la naissance de cet ouvrage.

Remerciements

Je tiens à remercier Hervé Ségalen pour m'avoir communiqué la passion des méthodes ; Muriel Shan Sei Fan pour son soutien sans faille à ce projet ; Pierre Tran qui nous a permis de croiser nos expériences ; et bien sûr Sophie, Jean, Thérèse et Christine pour leur patience et leurs encouragements.

Jean-Louis Bénard

Ce livre ne serait pas ce qu'il est si nous n'avions été précédés par Ron, Kent, Ward, Martin et tous les extrémistes de la première heure. Aujourd'hui encore je ne sais pas si je dois les maudire de m'avoir fait perdre mes illusions ou les remercier de m'avoir ouvert les yeux – à défaut d'être sage, j'opte pour la seconde voie. D'autres programmeurs et gens du métier sont responsables, souvent sans le savoir, des quelquesbonnes idées que j'ai pu glisser dans ces pages – les âneries sont naturellement de mon fait (sauf quand c'est la faute de Chet). Du cerveau à la page imprimée, il y a un chemin qui passe par le travail et la patience d'une équipe éditoriale : Muriel, Patrice, Anne, Sophie ont su en faire un véritable plaisir. Last but not least – comme bien des auteurs, c'est à ma famille que je réserve mes premiers instants de repos après avoir écrit le mot «fin», pour les remercier comme il se doit – Sophie, Axel, Erwan ; sans oublier «Nom de code Zébulon», ainsi qu'Alain et Ginette pour leurs encouragements.

Laurent Bossavit

Je remercie en premier lieu les Three extremos – Kent Beck, Ward Cunningham, Ron Jeffries –, dont j'admire le génie et surtout les qualités humaines pour avoir inventé une démarche qui transforme le compte à rebours infernal de chaque projet en une aventure collective passionnante. Merci aussi à ceux avec qui j'ai eu le plaisir de faire ce livre – je pense bien sûr aux autres auteurs, Jean-Louis, Laurent, Dominic, mais aussi à Muriel Shan Sei Fan, Patrice Beray, Sophie Hincelin et Anne Garcia pour la partie éditoriale. Je n'oublie pas non plus Thierry Cros, qui m'a permis d'embarquer sur ce projet il y a quelques mois.

Je souhaite également remercier ceux avec qui j'explore XP depuis quelques années : Marc Guiot, Thomas Novelle et Sébastien Crego – j'espère que nos aventures ne font que commencer. Un grand merci enfin à ma petite famille, Séverine, Chloé et Roméo, pour avoir eu la patience de vivre ces quelques mois avec un papa toujours collé à son écran.

Régis Médina

Je tiens à remercier d'abord mes coéquipiers, avec qui j'ai eu la chance de pratiquer XP : Fabrice, Michel, Michaël, Florent, Jean-Baptiste, François A., Franck, François M., Nicolas, Jacques, Patrice B., Patrice C., Cédric, Virgile et Alexandre. Vous m'avez appris tout ce qu'il y a de plus important, et travailler avec vous a été un réel plaisir. Merci aussi à Édith, Jean-Michel, Serge et Jean de nous avoir fait confiance, et d'être restés exigeants mais constructifs.

Je remercie mes coauteurs, Jean-Louis, Laurent et Régis ainsi que notre éditeur, Muriel Shan Sei Fan – vous avez fait de l'écriture de ce livre une expérience passionnante et enrichissante à de nombreux points de vue.

Enfin, je remercie de tout cœur ma petite famille : c'est en m'inspirant de ton intégrité, Pascale, et de ton dynamisme, Chloé, que je trouve la force morale et physique pour tout ce que j'entreprends.

Dominic Williams

Préface

Il est un aspect de l'Extreme Programming qui plonge certains lecteurs dans un désespoir mêlé de rage, et suscite l'admiration chez d'autres : sa vulnérabilité face au contexte culturel.

S'il existe bien une culture mondiale des « fêlés du code» *(geeks)* – et XP rend, par certains côtés, hommage aux traits de caractère et aux faiblesses humaines nécessaires pour devenir un maître programmeur –, il reste que l'activité de programmeur ne se conçoit que dans un contexte culturel plus vaste – celui d'une culture d'entreprise, d'une culture nationale…

Or, les éléments d'XP qui tentent de jeter un pont entre la culture des «geeks» et celle des entreprises américaines ne seront pas universellement applicables…

D'où ce livre, qui tire son efficacité de ce que ses auteurs sont partis des ingrédients fondamentaux de l'Extreme Programming, y ont mêlé leurs propres expériences de mise en œuvre en France, ont fait mijoter le tout au feu de leur dialectique collective, non sans l'avoir assaisonné d'une présentation originale.

Kent Beck

Table des matières

1

Introduction

Le contraire d'une vérité profonde peut très bien être une autre vérité profonde.

– Niels Bohr

L'Extreme Programming (XP) est un ensemble de pratiques qui couvre une grande partie des activités de la réalisation d'un logiciel – de la programmation proprement dite à la planification du projet, en passant par l'organisation de l'équipe de développement et les échanges avec le client. Ces pratiques n'ont en soi rien de révolutionnaire : il s'agit simplement de pratiques de bon sens mises en œuvre par des développeurs ou des chefs de projet expérimentés, telles que les livraisons fréquentes, la relecture de code, la mise en place de tests automatiques… La nouveauté introduite par XP consiste à pousser ces pratiques à l'extrême – d'où le nom de la méthode – et à les organiser en un tout cohérent, parfaitement défini et susceptible d'être répété.

Par opposition aux méthodes de développement dites «lourdes», qui imposent des activités et des produits indirects consommateurs de temps et de ressources, XP se définit comme un processus léger dans lequel l'équipe se focalise sur la réalisation elle-même. Toutes les autres activités sont réduites à leur strict minimum, sachant que la priorité y est donnée à l'agilité : XP s'adresse principalement à de petites équipes (moins de dix personnes) qui souhaitent réaliser rapidement des logiciels et réagir aisément au changement, sans faire pour autant de concessions sur la qualité du logiciel produit.

Dans la pratique, ce critère d'agilité proposé par XP se retrouve à trois niveaux :

- au niveau du code lui-même, qui est maintenu toujours aussi clair et simple que possible par une activité de remaniement permanente, et qui est soutenu par une batterie de tests automatiques permettant d'y introduire de nouvelles fonctionnalités sans craindre des régressions cachées ;

- au niveau de l'équipe de développement, dont les membres travaillent toujours en commun et sont capables d'intervenir sur toutes les parties de l'application ;
- au niveau de la gestion du projet, à travers une démarche itérative qui permet à tous les intervenants du projet – aussi bien l'équipe de développement que le client – d'améliorer continuellement son pilotage en s'appuyant sur l'expérience acquise en cours de route.

Mais à elle seule, cette agilité ne suffit pas à expliquer l'intérêt que suscite XP auprès des équipes qui le pratiquent. En effet, au-delà d'une recherche purement cartésienne de l'efficacité, XP est guidé par un ensemble de valeurs humanistes, une vision positive du développement logiciel fondée sur la conviction que l'efficacité s'obtient en prenant le temps de faire un travail de qualité, et en donnant une place de premier plan aux contacts humains – au sein de l'équipe elle-même, mais aussi entre cette dernière et son client.

Ainsi, même si la plupart des pratiques XP prises une à une sont déjà connues, elles ont ensemble le potentiel nécessaire pour créer un environnement tout à fait particulier, à la fois efficace et vivant – et l'expérience prouve que ceux qui ont connu l'environnement d'un projet XP réussi ont souvent bien du mal à revenir à des démarches plus traditionnelles par la suite.

Les limites des démarches « par phases »

L'Extreme Programming marque une rupture vis-à-vis des processus séquentiels, dérivés pour la plupart du classique « cycle en V » (figure 1-1) dont les différentes phases visent à produire successivement le cahier des charges de l'application, le document de spécifications, le document de conception générale, le document de conception détaillée, une collection de plans de tests… et éventuellement l'application elle-même !

De nombreuses variantes de cette approche ont vu le jour, parfois plus légères, parfois assouplies par l'introduction de cycles itératifs. Mais quoi qu'il en soit, le canevas d'un projet de développement reste toujours *grosso modo* le même : on cherche à définir très précisément ce que l'on souhaite construire avant de se lancer dans la réalisation proprement dite.

Or, si cette démarche a largement fait ses preuves lorsqu'il s'agissait de construire des ponts ou des immeubles, on ne peut lui attribuer le même succès dans le domaine du développement logiciel. Par exemple, si l'on en croit les résultats d'une étude menée en 1994 par une société de conseil américaine sur un échantillon de plus de 8000 projets, il ressort que seuls 16 % d'entre eux ont été finalisés dans le temps et le budget initialement prévus[1]. Pire, 32 % de ces projets ont été interrompus en cours de route.

Le diagnostic de ces dérapages est souvent le même : après des spécifications ambitieuses et une conception poussée, la construction s'enlise, souvent mise à mal par une remise en ques-

1. *The CHAOS Report*, réalisé par *The Standish Group* : http://www.pm2go.com/sample_research/chaos_1994_1.asp

tion des choix initiaux du projet qui oblige à revoir des pans entiers du modèle de conception proposé.

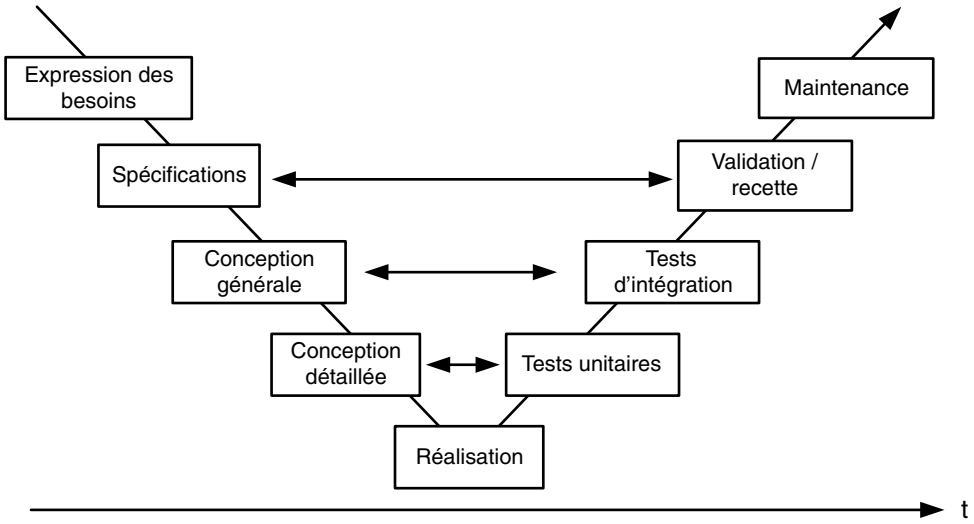

Figure 1-1. Le cycle en V

Mais si ces problèmes sont réellement chroniques, faut-il forcément critiquer la façon dont le processus est mis en œuvre par les équipes ? Ne faudrait-il pas plutôt se demander si le processus en question correspond réellement aux exigences du projet concerné ? L'approche linéaire se montre efficace dans certains contextes, mais il existe des cas dans lesquels les limites inhérentes à cette approche en font un piètre choix. Nous décrivons brièvement ces limites dans les sections qui suivent.

L'utopie des spécifications complètes et immuables

Les oublis, erreurs ou changements dans les spécifications constituent la source principale de problèmes pour la plupart des projets. Mais faut-il vraiment s'en étonner ? Mis à part certains cas d'applications très simples, ou encore dans de rares contextes parfaitement connus et maîtrisés, la définition *a priori* d'un logiciel un tant soit peu complexe est un exercice extrêmement difficile. Qui plus est, cet exercice déterminant prend place au début du projet, au moment où les différents intervenants en savent le moins sur le contexte précis de l'application – en particulier, le client a souvent bien du mal à se faire une idée précise de ce que pourrait être l'application quelques mois plus tard, surtout lorsqu'il n'existe pas d'applications équivalentes qui puissent servir de référence.

Remarque

Nous utilisons ici le terme de «client» d'une manière volontairement assez vague, désignant à la fois le donneur d'ordres, l'utilisateur final et celui qui finance le projet. Nous reviendrons sur ce terme précis au chapitre 2 consacré aux rôles des différents intervenants d'un projet XP.

Il n'est donc pas surprenant que ces spécifications soient remises en question à plusieurs reprises en cours de projet, notamment lorsque le client est mis en présence des premières versions du logiciel. Ces remises en question représentent une perte de temps, puisqu'une partie des efforts, fondée sur les spécifications initiales, est perdue, mais elles représentent aussi un risque pour la conception logicielle mise en place, qui absorbera plus ou moins bien les adaptations nécessaires.

Certes, les méthodes et les outils de modélisation ne cessent d'évoluer, le dernier élément majeur de cette évolution étant l'*Unified Modeling Language* initialement créé par Grady Booch, Ivar Jacobson et James Rumbaugh[1]. Mais ces outils n'apportent pas de changements réellement significatifs sur le déroulement des projets de développement, et il convient de se demander si la solution à ces problèmes de spécification et de conception passe nécessairement par un effort supplémentaire de modélisation… Ne serait-il pas envisageable au contraire d'adopter une approche radicalement différente qui accorde une importance moindre à cette phase initiale ?

L'effet tunnel

Autre travers chronique de l'approche linéaire, la visibilité réduite qui est accordée au client pendant la phase de construction. L'équipe peut rester isolée plusieurs mois et présenter un beau jour au client un produit qui correspond peut-être à ce qu'il avait spécifié, mais pas nécessairement à ce dont il a réellement besoin. Et la situation est souvent d'autant plus difficile à rattraper qu'elle est détectée tardivement dans le projet.

La solution à ce problème consiste naturellement à faire intervenir plus fréquemment le client au cours du projet, mais encore faut-il que cette intervention ne soit pas réduite à un simple constat : il faut lui donner réellement le pouvoir d'influer sur le cours du projet, quitte à revenir sur des décisions précédentes, et les changements de cap ainsi imposés doivent pouvoir être absorbés sans traumatisme par l'équipe de développement.

Des équipes compartimentées

L'un des objectifs de la phase de conception initiale est d'identifier un découpage de l'application qui permette de paralléliser sa réalisation. Cette approche offre des avantages indé-

1. Voir à ce propos l'ouvrage *UML en action* de Pascal Roques et Franck Vallée, Eyrolles, 2001.

niables lorsqu'il s'agit de mener de gros projets faisant intervenir des équipes aux spécialités réellement différentes, mais on peut légitimement douter de son adéquation au sein même d'équipes de taille raisonnable.

En effet, le modèle d'organisation d'équipe qui semble le plus répandu aujourd'hui – des développeurs travaillant seuls sur leurs machines, encadrés par un chef de projet qui se charge d'une communication «en étoile» – souffre de deux défauts principaux. Le premier tient à une mauvaise distribution de l'information dans l'équipe, qui contraint la répartition de la charge et limite ainsi la productivité globale de l'équipe. Le second défaut est dû à une spécialisation abusive des développeurs : ceux-ci sont cantonnés dans des parties réduites de l'application, jusqu'à ce qu'ils se lassent de faire toujours la même chose et quittent le projet.

Et si l'on amenait ces développeurs à travailler réellement ensemble, plutôt que de chercher une performance hypothétique dans la parallélisation et la spécialisation ?

Un code soumis à une paralysie progressive

Le dernier problème fréquemment associé aux processus lourds est une sorte de paradoxe : à force de chercher à prouver la qualité du logiciel par ses propriétés extérieures – la documentation, les tests –, il arrive que sa qualité interne, celle du code lui-même, finisse par laisser à désirer. Ce phénomène est souvent lié au modèle «éclaté» d'organisation de l'équipe : la qualité du code d'une partie de l'application dépendra uniquement de la compétence et de la rigueur du développeur qui y travaille.

L'importance de ce point particulier est souvent négligée par les décideurs du projet, qui considèrent d'une part que le code est exclusivement une affaire de développeurs, et d'autre part que les phases de validation ont justement pour objet de vérifier cette qualité. Seulement, un paramètre essentiel reste absent de ce raisonnement : la vitesse de développement. Plus le code dégénère, moins les extensions et les évolutions y sont aisées, et donc plus les développements coûtent cher. Ce problème est très difficile à déceler de l'extérieur ; soit il n'est jamais relevé, et l'application coûtera simplement plus cher que ce qu'il aurait dû en être, soit il apparaît un jour que l'application est devenue trop rigide, mais il est alors souvent trop tard pour corriger le tir et la récriture s'impose.

Remarque

Pour parer aux problèmes de qualité du code, il est par exemple possible d'organiser des séances de relecture de code croisée. Malheureusement, lorsqu'elles sont réalisées après l'écriture du code proprement dit, ces activités sont très coûteuses en temps, et peu d'équipes peuvent se permettre d'y consacrer l'investissement nécessaire.

Or, si le code – qui reste le livrable principal d'un projet de développement logiciel – fait l'objet d'attentions insuffisantes, c'est avant tout parce qu'il est généralement associé à une phase de *fabrication* du logiciel, qui devrait pouvoir découler des phases de conception

détaillées menées en amont. Mais l'expérience prouve que l'implémentation ne se réduit pas à cela : c'est au cours de l'implémentation que la validité des spécifications et de la conception est mise à l'épreuve. Pour que l'application puisse absorber les inévitables changements de spécifications et de conception qui interviennent dans un projet, il faut intégrer l'activité d'implémentation dans celles de spécification et de conception – la fabrication proprement dite étant l'affaire des compilateurs.

Un changement de référentiel

Si l'approche linéaire du développement présente certaines limites, il n'empêche qu'elle reste aujourd'hui à la base de la quasi-totalité des projets de développement – à tel point qu'il paraît bien difficile d'imaginer une autre façon de procéder. Mais pourquoi en est-il ainsi ?

L'une des idées principales qui soutiennent cette approche est que, plus on avance dans le projet, plus le coût des modifications dans le logiciel est élevé. Le coût du changement augmenterait même de manière exponentielle avec le temps, comme l'illustre la figure 1-2.

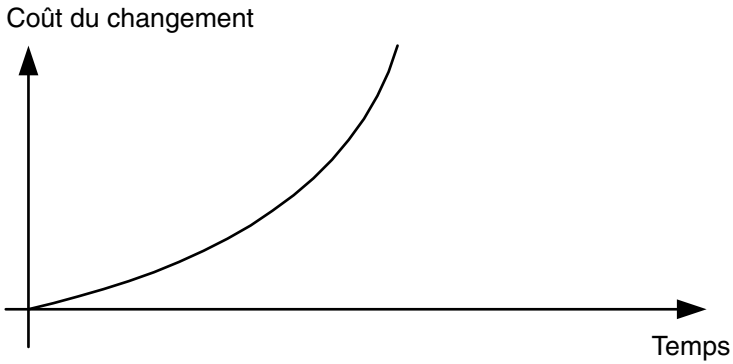

Figure 1-2. Évolution supposée du coût du changement dans un logiciel

Dans un tel contexte, il est tout à fait justifiable – et même recommandé – de chercher à définir complètement le logiciel avant d'entamer sa construction. Partant de cela, une approche linéaire plus ou moins inspirée du cycle en V finit d'ailleurs par s'imposer.

Remarque

Concernant l'hypothèse d'une augmentation exponentielle du coût du changement avec le temps, on peut cependant se demander dans quelle mesure une approche fondée sur cette supposition ne tend pas à générer des logiciels qui la respectent, créant ainsi un phénomène de renforcement réciproque.

Or, ce qu'ont découvert les créateurs d'XP – qui étaient avant tout des experts de la conception et de la programmation objet –, c'est qu'en appliquant certains principes de conception et de programmation, il devenait possible de garder l'application suffisamment «souple» pour que les changements restent aisés tout au long du projet. La courbe d'évolution du coût du changement présentait alors un autre profil, pour adopter celui présenté en figure 1-3.

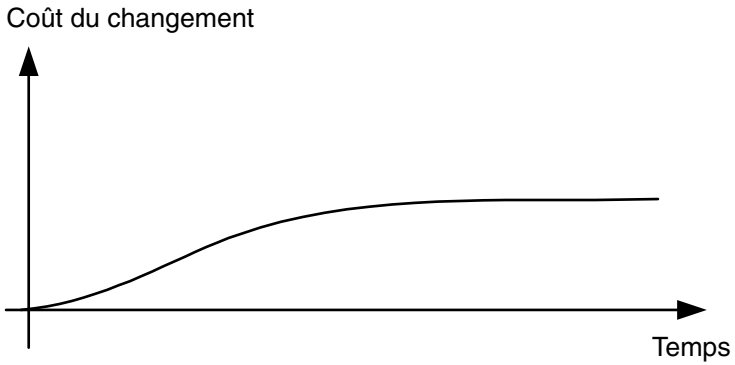

Figure 1-3. L'évolution du coût du changement dans le logiciel revue par les auteurs d'XP

Dans un tel contexte, la stratégie gagnante ne consiste plus à tout figer dès le début, mais au contraire à diffuser la prise de décisions tout au long du projet pour tirer parti de l'expérience acquise en cours de route. Cela est valable pour la conception du logiciel – les choix de conception sont faits au moment où l'implémentation le réclame – mais aussi pour les spécifications elles-mêmes – le client précise les détails de ses besoins au moment où les développeurs implémentent les fonctionnalités correspondantes. De la sorte, les décisions ne sont plus prises sur la base d'analyses et de suppositions abstraites, mais à partir des informations concrètes, disponibles au moment même où elles doivent être appliquées.

L'Extreme Programming prend place dans ce nouveau référentiel. Il propose un ensemble de pratiques concrètes qui permettent, d'une part, d'obtenir la souplesse requise et, d'autre part, d'exploiter cette souplesse pour servir au mieux les intérêts du client et de l'équipe de développement.

Les pratiques d'XP

XP définit treize pratiques «canoniques», que nous classons ici en trois grandes catégories : celles relatives à la programmation, celles relatives au fonctionnement interne de l'équipe et celles qui sont liées à la planification et aux relations avec le client.

Les pratiques de programmation

Au cœur des pratiques XP, les pratiques de programmation (voir chapitres 3 et 4) que nous présentons ci-après permettent d'améliorer continuellement la conception et le code de l'application pour qu'elle reste toujours aussi claire et simple que possible :

- **Conception simple** (*simple design*) : les développeurs implémentent toujours la solution la plus simple qui puisse fonctionner. En particulier, ils n'inventent pas de mécanismes génériques si le besoin immédiat ne l'exige pas.
- **Remaniement** (*refactoring*) : les développeurs n'hésitent pas à revenir sur le code écrit pour le rendre plus «propre», le débarrasser d'éventuelles parties inutilisées, et le préparer à l'ajout de la fonctionnalité suivante. D'une manière plus générale, cette pratique propose une démarche de conception continue qui fait émerger la structure de l'application au fur et à mesure du développement.
- **Développement piloté par les tests unitaires** (*test-first programming, unit tests, developer tests*) : les développeurs écrivent des tests automatiques pour le code qu'ils produisent, et ce au moment même d'écrire le code en question. Cela leur permet d'une part de mieux cerner le problème avant d'écrire le code, et d'autre part de constituer progressivement une batterie de tests qui les autorise ensuite à apporter rapidement des changements dans l'application, tout en conservant une certaine sérénité.
- **Tests de recette** (*acceptance tests, customer tests*) : le client précise très explicitement ses besoins et les objectifs des programmeurs – en participant à la rédaction de tests de recette. Comme les tests unitaires, les tests de recette doivent être automatiques afin de pouvoir vérifier tous les jours la non-régression du produit.

Les pratiques de collaboration

Dans une équipe XP, tous les développeurs travaillent ensemble (voir chapitre 5) et interviennent sur la totalité de l'application. Cela garantit la qualité de cette dernière à travers les relectures croisées que cela engendre, mais cela rend également le travail plus motivant et offre une plus grande souplesse dans l'affectation des tâches. Les pratiques qui régissent cette organisation sont les suivantes :

- **Programmation en binôme** (*pair programming*) : lorsqu'ils écrivent le code de l'application, les développeurs travaillent systématiquement à deux sur la même machine – il s'agit là d'une forme «extrême» de relecture de code, dans laquelle les deux développeurs collaborent activement pour résoudre les problèmes qu'ils rencontrent. Les binômes changent fréquemment, ainsi chacun est amené à travailler tôt ou tard avec tous les autres membres de l'équipe.
- **Responsabilité collective du code** (*collective code ownership*) : tous les développeurs de l'équipe peuvent être amenés à travailler sur toutes les parties de l'application. De plus, ils

ont le devoir d'améliorer le code sur lequel ils interviennent, même s'ils n'en sont pas les auteurs initiaux.

- **Règles de codage** (*coding standards*) : les développeurs se plient à des règles de codage définies par l'équipe elle-même, de manière à garantir l'homogénéité de leur code avec le reste de l'application, et ainsi à faciliter l'intervention d'autres développeurs.
- **Métaphore** (*metaphor*) : les développeurs n'hésitent pas à recourir aux métaphores pour décrire la structure interne du logiciel ou ses enjeux fonctionnels, de façon à faciliter la communication et à assurer une certaine homogénéité de style dans l'ensemble de la conception, l'idéal étant de décrire le système dans son ensemble par une métaphore unique.
- **Intégration continue** (*continuous integration*) : les développeurs synchronisent leurs développements aussi souvent que possible – au moins une fois par jour. Cela réduit la fréquence et la gravité des problèmes d'intégration, et permet de disposer à tout moment d'une version du logiciel qui intègre tous les développements en cours.

Les pratiques de gestion de projet

Les pratiques de programmation et de collaboration (voir chapitre 6) permettent de créer un contexte dans lequel une démarche de spécification et de conception purement itérative devient viable. Les pratiques suivantes montrent comment XP exploite cet avantage afin de s'assurer que l'équipe et son client restent en phase tout au long du projet, de façon à converger au plus tôt vers un produit adapté aux besoins du client :

- **Livraisons fréquentes** (*frequent releases*) : l'équipe livre des versions du logiciel à un rythme régulier, aussi élevé que possible – la fréquence précise étant fixée par le client. Cela permet à l'équipe comme au client de s'assurer que le produit correspond bien aux attentes de ce dernier et que le projet est sur la bonne voie.
- **Planification itérative** (*planning game*) : la planification du projet est réalisée conjointement par le client et l'équipe de développement, au cours de séances dédiées, organisées régulièrement tout au long du projet.
- **Client sur site** (*on-site customer, whole team*) : le client est littéralement intégré à l'équipe de développement pour arbitrer les priorités, et définir précisément ses besoins, notamment en répondant en direct aux questions des programmeurs et en bénéficiant du feedback immédiat d'une application aux livraisons fréquentes.
- **Rythme durable** (*sustainable pace*) : l'équipe adopte des horaires qui lui permettent de conserver tout au long du projet l'énergie nécessaire pour produire un travail de qualité et mettre en œuvre efficacement les autres pratiques.

Les quatre valeurs d'XP

Si elles sont indispensables aujourd'hui pour mettre l'Extreme Programming en œuvre, les pratiques que nous venons de présenter ne suffisent pas pour autant à le définir. Il ne s'agit en définitive que de *techniques*, destinées à faire émerger un environnement de travail marqué par les quatre qualités érigées en valeurs par XP et qui en font l'essence : la communication, la simplicité, le *feedback* et le courage.

La communication pour une meilleure visibilité

Du point de vue d'XP, un projet de développement est avant tout un effort *collectif* de création, dont le succès dépend d'une part de la capacité de ses différents intervenants à s'accorder sur une vision commune de ce qui doit être produit, et d'autre part de leur capacité à synchroniser leurs actions individuelles pour atteindre l'objectif commun. Or, ces deux conditions dépendent en majeure partie de la qualité de la communication qui lie ces intervenants entre eux. Mais sur le fond, il n'y a finalement pas grand-chose de spécifique à XP sur ce point.

Ce qui démarque l'approche XP dans ce domaine, c'est l'accent mis sur la communication directe, sur le contact humain. Certes, la communication orale présente des faiblesses en termes de structuration de l'information et de traçabilité, mais elle tire sa force de sa simplicité et des interactions rapides entre les interlocuteurs qui permettent de converger rapidement sur les informations essentielles. En outre, le contact direct permet de véhiculer des informations beaucoup plus personnelles – cela donne ainsi l'occasion d'identifier et de résoudre des blocages qui relèvent de positions individuelles, qui sans cela pourraient compromettre les chances de succès du projet.

Au quotidien, la communication directe permet donc d'obtenir une « bande passante » nettement supérieure à l'écrit, et XP exploite largement cet avantage dans la plupart de ses pratiques. Ainsi, qu'il s'agisse de l'identification des besoins, de la planification, de la répartition des tâches ou encore de la programmation proprement dite, la plupart des pratiques d'XP sont conçues pour amener les intervenants du projet à communiquer directement et résoudre les problèmes ensemble.

Remarque

Cette approche constitue l'une des forces majeures d'XP, mais c'est aussi d'un certain point de vue sa principale faiblesse. D'une part, ce besoin de communication est le principal facteur limitatif au regard de la taille de l'équipe et, d'autre part, cela rend XP très sensible au type de relations humaines qui ont cours dans le projet : appliquer XP dans un projet marqué par des antagonismes internes forts, c'est courir à la catastrophe – mais on peut légitimement se demander comment un tel projet peut bien réussir, indépendamment du processus adopté.

Cependant, si l'accent est placé sur la communication directe, la communication écrite n'est pas laissée de côté pour autant. Mais elle apparaît sous une forme différente : toutes les informations ayant trait à l'implémentation et la conception se retrouvent dans le code lui-même – dont la clarté fait l'objet de nombreux efforts – et dans la batterie de tests qui l'accompagne, et celles relatives aux besoins du client sont consignées d'une manière on ne peut plus formelle, sous la forme de tests de recette automatiques. Si le client souhaite obtenir des documents supplémentaires, il lui suffit de définir et de planifier les tâches correspondantes comme pour tout autre besoin.

La simplicité comme garantie de productivité

Une personne qui arrive sur un projet XP ne doit pas s'étonner de s'entendre répéter les quelques « mantras » suivants : « La chose la plus simple qui puisse marcher » (*The simplest thing that could possibly work*), « Tu n'en auras pas besoin » (*You ain't gonna need it* – et son acronyme YAGNI), ou encore « Une fois et une seule » (*Once and only once*). Tous trois sont l'expression d'une même valeur : la simplicité.

Cette simplicité touche d'abord à la réalisation même du logiciel, dans laquelle un soin particulier est apporté au code pour le débarrasser de toute complexité superflue. En particulier, les mécanismes de généricité – le « péché mignon » des développeurs – ne sont encouragés que lorsqu'ils servent un besoin concret et immédiat, et en aucun cas un besoin futur plus ou moins imaginaire. Les développeurs sont donc invités à implémenter *la chose la plus simple qui puisse marcher* et, en ce qui concerne ces fameux mécanismes génériques, ou encore ces outils « magiques » qui en font plus que ce qu'on leur demande, *ils n'auront pas besoin* ! En revanche, simple ne veut pas dire simpliste : les solutions adoptées doivent être aussi simples que possible, mais toutes les duplications doivent être éliminées de sorte que chaque information, chaque mécanisme ne soit exprimé qu'*une fois et une seule* – ce principe est garant de la facilité de modification du code sur le long terme.

Cet effort de simplicité s'applique également au client, à qui l'équipe demandera de définir ses besoins et ses priorités avec une grande précision pour éviter d'implémenter des choses inutiles et pour pouvoir se focaliser sur les besoins réellement importants. La simplicité est également recherchée dans le choix des outils (de programmation ou de gestion de projet) et dans la méthode de travail elle-même.

En définitive, XP est fondé sur un pari : « faire simple un jour, avec la possibilité de revenir en arrière le lendemain si un besoin différent apparaît. L'éventuel coût supplémentaire induit sera, d'une part réduit parce que l'application est restée assez simple pour évoluer facilement, et d'autre part sera bien peu de choses au regard des économies réalisées à ne pas faire ce qui n'aurait servi à rien. » C'est au succès de ce pari qu'une équipe XP doit sa vitesse de développement et son ouverture au changement.

Le feedback comme outil de réduction du risque

Malgré ce que peuvent laisser supposer son nom et ses apparences immédiates, l'Extreme Programming est avant tout un processus de réduction du risque dans le projet. Le risque est en effet soigneusement contrôlé à tous les niveaux, par la mise en place de boucles de *feedback* qui permettent à l'équipe de développement, comme à son client, de savoir à tout moment dans quel état se trouve réellement le projet, et de pouvoir rectifier le tir au fur et à mesure pour mener le projet à son terme avec succès.

L'exemple le plus saillant de ce principe concerne la pratique des livraisons fréquentes, qui donne à tous les intervenants du projet un feedback régulier sur l'état d'avancement du projet et l'état réel du produit. Mais le feedback ne se borne pas à l'observation : la pratique de la planification itérative permet de tirer parti des informations recueillies pour, à la fois, améliorer la planification elle-même et faire converger le produit vers une solution mieux adaptée aux besoins réels du client.

L'activité de programmation fait également l'objet de divers mécanismes de feedback, tout d'abord à travers les tests unitaires mis en place, qui donnent aux développeurs des indications immédiates sur le fonctionnement du code qu'ils écrivent. Le feedback est également intégré à l'activité de conception, qui est menée tout au long de la réalisation pour s'appuyer sur l'état réel de l'application plutôt que sur l'idée que l'on pourrait en avoir *a priori*. Enfin, les développeurs s'appuient en permanence sur le feedback de leur binôme pour s'assurer de la validité et de la qualité du code qu'ils produisent.

Ce feedback permanent est un facteur de qualité, puisque les intervenants du projet améliorent sans cesse leur travail à partir de l'expérience qu'ils accumulent. Mais c'est également un facteur de vitesse : une équipe XP peut programmer vite, tout en restant sereine parce qu'elle sait qu'elle dispose de nombreux mécanismes pour s'assurer que le projet reste sur la bonne voie.

Le courage de prendre les bonnes décisions

L'expérience fait apparaître que la mise en œuvre des pratiques XP requiert une certaine dose de cran. En effet, il faut du courage pour se lancer dans un projet sans avoir au préalable tout spécifié et conçu dans le détail, même si l'on sait que le processus suivi comporte de nombreux mécanismes de feedback.

Il faut également du courage pour se borner à réaliser des choses simples, se focaliser uniquement sur les besoins du moment en se disant qu'on pourra adapter l'application à de nouveaux besoins, le moment venu. Il en faut aussi pour accepter de jeter une partie du code qui est devenue inutile, ou récrire une partie de code jugée trop complexe.

Enfin, il faut du courage pour appliquer les principes de communication et de feedback, en particulier lorsqu'il s'agit de maintenir une transparence complète, même lorsque les nouvelles ne sont pas bonnes, ou encore lorsqu'il s'agit de travailler ouvertement avec son binôme en acceptant de lui montrer nos propres limites ou lacunes.

Plus généralement, nous avons pu constater qu'il faut du courage pour mettre en place des méthodes nouvelles sur des projets dont les enjeux sont toujours stratégiques.

Racines historiques et acteurs d'XP

XP est principalement l'œuvre de Kent Beck et Ward Cunningham, deux experts du développement logiciel, et plus particulièrement de la conception objet et des *patterns*. Avant le développement d'XP, Kent Beck devait surtout sa notoriété dans le monde Smalltalk à son livre, *Smalltalk Best Practice Patterns*.

La naissance d'XP

La naissance d'XP a eu lieu sur un projet Chrysler – le «C3» (*Chrysler Comprehensive Compensation System*) – où Kent Beck intervenait initialement pour travailler sur l'amélioration des performances du produit. Après avoir convaincu la direction que le produit souffrait de maux plus profonds, Kent Beck s'est vu confier la responsabilité d'encadrer l'équipe pour le récrire. C'est là qu'il a mis au point les pratiques d'XP, avec l'aide d'un autre acteur important d'XP : Ron Jeffries.

Le projet C3 fut réalisé en Smalltalk ; si, pour de nombreux observateurs, la démarche XP s'est largement affranchie du langage et s'applique aussi bien à Java ou C++, XP conserve de fortes affinités, en particulier, avec Smalltalk[1] et, d'une façon plus générale, avec la famille des langages «dynamiques».

Parmi les autres personnages influents d'XP, on trouve notamment Martin Fowler, auteur de livres sur l'UML et plus récemment du livre de référence sur le Refactoring (l'une des pratiques de base d'XP). On y rencontre également Robert C. Martin, président d'Object Mentor, une société de consultants de haut niveau spécialisés dans la conception objet.

XP est également le fruit d'un effort collectif supporté par le Wiki Wiki Web[2], un site collaboratif mis au point par Ward Cunningham. Les échanges menés sur ce site ont permis aux auteurs d'XP d'affiner la définition et la présentation de leur démarche, avant de pouvoir la rendre accessible au reste de l'industrie à travers des ouvrages dédiés.

D'une certaine manière, on a pris acte de la naissance d'XP en octobre 2000 avec la parution du livre *Extreme Programming Explained* de Kent Beck, dans lequel il expliquait les fondements de sa démarche. Les livres *Extreme Programming Installed* et *Planning Extreme Programming* ont suivi peu après, donnant un éclairage résolument plus pratique et concret de la démarche.

1. Voir à ce propos l'ouvrage *Squeak* de Xavier Briffault et Stéphane Ducasse, Eyrolles, 2002.
2. http://www.c2.com/cgi/wiki?ExtremeProgrammingRoadmap

Les inspirations d'XP

L'Extreme Programming est issu du mouvement des *patterns* – celui qui a donné naissance, entre autres, aux *Design Patterns* – dont Kent Beck et Ward Cunningham sont des acteurs centraux. Ce mouvement est en grande partie influencé par les travaux de Christopher Alexander, un architecte qui prône une approche progressive de la construction de bâtiments, basée sur une forte implication de ses habitants.

Parmi les travaux qui ont inspiré XP, on trouve principalement le langage de patterns «*Episodes*» de Ward Cunningham, et dans une moindre mesure les *Organizational Patterns* de Jim Coplien. L'importance du facteur humain dans XP a en partie été inspirée par le livre *Peopleware* de Tom DiMarco et Timothy Lister.

Les premières entreprises à adopter largement XP ont été des sociétés de conseil telles que Object Mentor, RoleModel Software, ou encore Industrial Logic. De nombreuses autres initiatives ont vu le jour – quelques-unes en France et en Europe, la plupart aux États-Unis – mais, à l'heure où nous rédigeons ces lignes, l'Extreme Programming en est encore à une phase de lancement pour ce qui est de son application à grande échelle.

Guide de lecture

Ce livre est organisé en trois parties, qui donnent des éclairages complémentaires sur l'Extreme Programming.

La première partie est consacrée à la présentation détaillée d'XP : les rôles des différents intervenants, les diverses pratiques que nous venons d'introduire. Elle s'adresse principalement à ceux – chefs de projets ou développeurs – qui souhaitent en acquérir une compréhension fine en vue de l'appliquer sur leurs propres projets. À cette fin, nous avons pris soin de compléter la présentation purement descriptive des pratiques par des retours d'expériences tirés de notre propre mise en œuvre de la démarche.

La deuxième partie a pour objet de situer l'Extreme Programming dans le cadre plus général de l'entreprise. Les thèmes abordés couvrent la transition vers XP, les aspects économiques et contractuels d'XP, mais aussi la place d'XP vis-à-vis des démarches d'assurance qualité et des autres processus de développement actuellement utilisés dans les entreprises.

Enfin, la troisième partie de cet ouvrage a pour objectif d'illustrer le déroulement particulier d'un projet XP d'une manière vivante, à travers deux études de cas. La première concerne un projet Web «idéal», décrit sous la forme d'une fiction, inspirée toutefois de faits réels. La seconde est un réel retour d'expérience d'un projet XP mené dans un cadre industriel.

Cet ouvrage est principalement destiné à être lu de bout en bout, mais les lecteurs pourront se focaliser sur certains chapitres en fonction de leurs centres d'intérêts et du rôle qu'ils peuvent être amenés à jouer dans un projet XP :

- Un développeur intégré dans un projet XP pourra se concentrer sur l'ensemble des chapitres de la première partie (chapitres 2 à 6).
- Un «décideur» qui s'interroge sur l'opportunité d'introduire XP dans une ou plusieurs de ses équipes pourra se focaliser sur la deuxième partie de cet ouvrage, et éventuellement la troisième s'il souhaite savoir plus concrètement de quoi il retourne.
- Une personne amenée à jouer le rôle de client d'une équipe XP pourra se concentrer sur le chapitre 2, qui décrit les rôles des intervenants d'un projet XP, sur le chapitre 6 dédié aux pratiques de gestion de projet, ainsi que sur les chapitres 8 à 10 dédiés aux aspects économiques, contractuels et méthodologiques d'XP.

Finalement, le seul intervenant qui n'aura pas droit aux raccourcis est le chef de projet. Comme nous allons le voir dans le chapitre suivant, XP lui réserve un rôle particulier dans l'équipe : celui de «coach», dont l'une des activités principales est de guider les autres intervenants dans la mise en œuvre de la démarche. Il doit pour cela avoir une connaissance très fine des facteurs qui font le succès d'XP, et le livre entier ne sera pas de trop pour l'aider dans sa tâche.

Première partie

Les pratiques de l'Extreme Programming

L'Extreme Programming établit aujourd'hui un ensemble de treize pratiques qui régissent de manière assez stricte les activités d'un projet de développement. Ces pratiques concrètes forment une *méthode* de développement, c'est-à-dire une démarche complète pour spécifier, concevoir, programmer et tester un logiciel. Cette méthode sert de guide au quotidien pour les différents acteurs du projet. Pourtant, les auteurs d'XP préfèrent parler, plutôt que de méthode, de *système de pratiques*, ou encore de *discipline de développement*.

En effet, il ne s'agit pas d'appliquer «mécaniquement» un ensemble de recettes. Il s'agit bien plutôt de mettre l'accent sur un réajustement permanent du projet, non seulement en ce qui concerne la réalisation du logiciel, mais également pour tout ce qui touche au processus suivi par l'équipe – que ce processus soit défini formellement ou non. Il est donc attendu d'une équipe XP qu'elle remette sans cesse en question ses méthodes de travail – plutôt qu'elle ne suive des instructions à la lettre.

Ainsi les treize pratiques définies aujourd'hui par XP ne doivent-elles pas être perçues comme une fin en soi. Il s'agit avant tout de techniques destinées à favoriser la recherche de l'efficacité de l'équipe, tout en veillant au respect des quatre valeurs d'XP : communication, simplicité, *feedback* et courage. Ces pratiques continuent à évoluer sous l'impulsion de la communauté XP, et les équipes elles-mêmes sont invitées à les adapter à leurs besoins pour gagner encore en efficacité – à condition toutefois de savoir déjà mettre en œuvre l'ensemble des pratiques existantes.

Les cinq chapitres qui suivent ont pour objectif de vous aider à atteindre cette première étape. Nous y décrivons l'ensemble des pratiques de manière concrète, tout en veillant à mettre l'accent sur leur justification, sur le fait qu'elles forment un tout cohérent, susceptible d'améliorer de façon significative la productivité de l'équipe.

Pour commencer, le chapitre 2 décrit les rôles des différents acteurs d'un projet XP. Ce chapitre offre une vue d'ensemble d'XP, permettant à chacun de voir où il intervient, et quelles sont les pratiques qui le concernent plus particulièrement. Ensuite, les chapitres

suivants entrent dans le détail des pratiques proprement dites. Et, comme l'approche XP repose en premier lieu sur l'ouverture au changement du logiciel lui-même, nous présentons ces pratiques «de l'intérieur vers l'extérieur» : tout d'abord les pratiques de programmation, avec un petit zoom sur la pratique des tests unitaires, ensuite les pratiques de collaboration de l'équipe, et enfin les pratiques de gestion du projet.

2

Organisation de l'équipe

C'est étonnant ce que l'on peut accomplir lorsqu'on ne se préoccupe pas de qui s'en voit attribuer le mérite.

– Harry S. Truman

Les chefs de projet expérimentés le savent bien : la constitution d'une bonne équipe peut garantir la réussite des projets les plus difficiles. À l'inverse, les problèmes humains peuvent compromettre un projet même s'il ne comporte aucune difficulté technique et s'il lui est alloué un budget confortable. Il ne suffit pas de réunir un certain nombre d'individus, aussi brillants ou compétents soient-ils pour former une bonne équipe. L'esprit d'équipe qui les anime va sensiblement influer sur le bon fonctionnement du projet. Il faut y être attentif, notamment lors du recrutement des membres de l'équipe, et ensuite tout au long du projet. En contrepartie, la méthode XP est ainsi faite qu'elle facilite et renforce cet esprit d'équipe, et surtout permet d'en tirer des bénéfices et des richesses insoupçonnables. Les nombreuses pratiques collaboratives y sont pour quelque chose.

Cependant, spontanéité, esprit d'équipe et travail collaboratif ne signifient pas que le chaos règne et que le déroulement du travail est laissé au hasard. Une équipe XP est au contraire très organisée, à chacun de ses membres est attribué un rôle assorti d'attributions, responsabilités, droits et devoirs bien précis.

Dans ce chapitre, nous allons détailler cette organisation, avec un double objectif : d'une part, aider à constituer une équipe XP en sachant quelles compétences techniques et humaines sont requises de chaque intervenant, et, d'autre part, permettre à ces derniers de connaître précisément leurs attributions, d'épouser le rôle qui leur est dévolu. Pour ce faire, nous allons procéder à une description rapide des activités associées à chaque rôle, de manière à donner une vue d'ensemble concrète des pratiques avant de les décrire plus en détail dans les chapitres suivants.

Les principaux rôles XP

Avec les pratiques, les rôles sont une des pierres angulaires de la méthode et font véritablement partie de la culture XP. Ils sont au nombre de six : programmeur, client, testeur, tracker, manager et coach. Il en va des rôles comme des pratiques : certains sont «officiels» au sens où il est aisé d'en reconnaître la mise en œuvre sur chaque projet (client, coach, programmeur), d'autres dépendent davantage du contexte (testeur, manager), ou peuvent être plus subtils et difficiles à attribuer ou à prendre en charge (tracker).

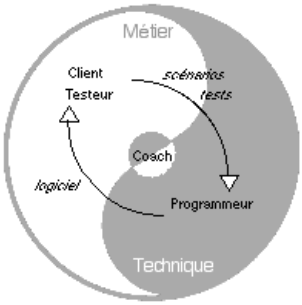

Figure 2-1. Les principaux rôles XP

Outre les définitions individuelles de ces rôles, nous reparlerons dans ce chapitre de leur répartition en évoquant le «cumul des mandats» – bien souvent, une même personne assume plusieurs rôles.

Mise en garde

Les six principaux rôles XP sont définis avec précision *dans le contexte XP*. Mais les noms de ces rôles peuvent prêter à confusion car ils ont par ailleurs d'autres significations, voire connotations.

Quant à la correspondance des rôles XP avec des rôles traditionnels, elle n'est pas toujours immédiate. Il faut bien se garder, par exemple, de croire que le rôle classique de chef de projet correspond exactement à celui du manager XP. Certains chefs de projet adopteront plutôt le rôle de coach. Pour prendre un autre exemple, un programmeur XP est loin d'être un technicien inexpérimenté ou incapable de mener des activités d'analyse et de conception. Tous les programmeurs XP, qu'ils soient débutants ou architecte concepteur, font de la planification, de la conception et du test outre la programmation à proprement parler. Et, comme la méthode XP met justement le code et ses artisans au premier plan, c'est par ce rôle que nous allons commencer.

Programmeur

L'importance donnée à la programmation et au code lui-même exprime deux convictions :

- En génie logiciel, l'activité qui correspond à la fabrication n'est pas la programmation mais la compilation. La programmation fait partie intégrante de la conception.
- Un code clair, structuré et simple, modifié autant de fois que nécessaire pour qu'il soit compréhensible de tous, et allégé de ses parties redondantes ou superflues, est *la meilleure* forme de conception, couvrant tous les aspects et points de vue, exprimant, sans ambiguïté et dans un langage compris à la fois des hommes et des machines, ce que doit faire un logiciel. Un code peut même exprimer une conception de haut niveau bien mieux que ne le ferait quelque notation ou AGL que ce soit.

Les tests et la proximité du client permettent de coder utilement

Pour avoir un logiciel, il faut du code, et les programmeurs en sont les créateurs et artisans. Le premier rôle du programmeur va donc consister à «s'occuper» du code : l'écrire, le connaître, le modifier, gérer son existence, sa sauvegarde et ses versions, gérer sa transformation en code exécutable.

Encore faut-il que le code fasse vraiment ce qu'on veut. Pour cela, il faut le *tester*, et c'est aussi le programmeur qui s'en charge. N'oublions pas que parmi les quatre valeurs d'XP figurent le feedback et le courage. Le programmeur effectue des tests[1] parce qu'il veut des retours – il lui est nécessaire de savoir quand le programme marche – et parce qu'il fait preuve d'une certaine forme de courage : il considère que ce qui n'est pas démontré par un test, et donc toute fonctionnalité qui n'a pas été sérieusement testée, n'existe pas. Tout simplement. Entre ce que l'on pense avoir programmé (ou ce qu'on pense comprendre du code que l'on lit) et ce qui est réellement programmé, il y a souvent des nuances. Quant aux décalages entre ce que l'on voulait programmer et ce qui aurait dû être programmé… La nécessité de procéder à des tests n'en apparaît que plus évidente.

Mais justement, comment le programmeur peut-il s'assurer qu'il teste ce qui doit l'être ? En posant des questions au client et en écoutant les réponses. C'est le client qui détient la connaissance concrète de ce que doit faire le système : quel est le résultat de tel calcul ? quels sont les cas particuliers ? Le programmeur doit savoir écouter, comprendre, et ce activement, en aidant le client à définir son besoin, à faire des choix en connaissance de cause. La communication est une valeur cardinale de la méthode XP, et, pour communiquer, il est plus important, et surtout plus difficile, d'écouter que de parler (nous avons deux oreilles et… une seule bouche !).

Fort opportunément, les pratiques XP fournissent un cadre méthodologique très précis pour faciliter et canaliser la communication, comme nous le verrons dans les chapitres suivants

1. Ou en écrit, comme nous le verrons aux chapitres 3 et 4.

consacrés à la description détaillée des pratiques XP. Il faut retenir pour le moment qu'un programmeur doit savoir écouter, et en particulier écouter le client. Cela fait partie de ses attributions.

La conception sert à coder durablement

Coder, tester, écouter… Ces trois activités suffisent presque à réaliser un bon logiciel. Pendant quelques jours, quelques semaines peut-être, cela fonctionne. Mais à mesure que le programme grossit et devient plus complexe, il devient plus difficile de coder et de tester. Les spécifications changent et l'on exige du programme des prouesses pour lesquelles il n'avait pas été conçu initialement. L'on souhaite alors modifier une partie du code, mais d'autres parties ne marchent plus et l'on assiste au phénomène dit de régression : le code devient fragile.

Pour faire une petite modification ou ajouter un test, on est contraint de toucher à tous les modules ou de modifier un grand nombre de tests existants : le code est soudain rigide. Enfin, lorsqu'on trouve l'opportunité de réutiliser un morceau de code existant pour une autre partie du programme, on se rend compte qu'il est difficile d'extraire ce code, pourtant utile, de son contexte d'origine sans tout casser : le code est devenu immobile.

Pour éviter cela, il faut réfléchir à la *conception* (*design*). Certes, la conception est une activité classique du génie logiciel, mais elle répond, au sein d'une démarche XP, à des objectifs différents.

L'une des originalités de la méthode XP réside en effet dans les objectifs qu'elle fixe à la conception. La conception ne sert pas à montrer au client ou au manager ce que l'on va coder ou ce que l'on a codé. Elle ne sert pas à produire des documents remplis de schémas, censés aider un nouvel arrivant à comprendre le code. Elle ne sert même pas aux développeurs comme phase de réflexion pour dégrossir le système avant de commencer à coder. En XP, on l'aura compris, la conception a pour seul but de garantir le long terme.

> **Remarque**
> Il ne faudrait pas en conclure qu'en XP, la conception revêt moins d'importance que dans d'autres méthodes. Elle est certes pratiquée différemment, et est l'affaire de tous, du moins de tous les programmeurs. En effet, la conception est encore l'une des responsabilités qui incombent au rôle de programmeur.

Les programmeurs interviennent sur la conception de trois façons :

- collectivement, lors des séances de planification, afin de déterminer les tâches et d'estimer leurs difficultés ;
- lorsqu'ils écrivent les tests unitaires du code ;
- lorsqu'ils pratiquent le remaniement (*refactoring*).

Les programmeurs sont responsabilisés

Le programmeur XP est donc un programmeur très complet. Il est à la fois codeur, testeur, concepteur et analyste. Ses qualités humaines importent autant que ses compétences techniques. Ces exigences peuvent sembler excessives, voire irréalistes. Heureusement, tous les programmeurs d'une équipe XP ne doivent pas être des développeurs confirmés, rompus aux activités d'analyse, de conception et de tests. Au moins un ou deux, toutefois, doivent répondre à ces critères, qui doivent avoir envie de diffuser leurs compétences et de partager les responsabilités. Il faut aussi que les développeurs moins expérimentés aient envie d'apprendre et d'assumer toutes les responsabilités du rôle de programmeur. La méthode leur fournit en contrepartie un cadre idéal pour apprendre. XP effectue un nivellement par le haut. C'est une école de l'excellence.

XP tire ainsi parti d'un fait bien connu : c'est souvent en responsabilisant les individus qu'ils donnent le meilleur d'eux-mêmes. Dans cet esprit, les programmeurs sont naturellement amenés à écouter le client, à effectuer des tests et à réfléchir à la conception. Toujours dans cet esprit, la méthode XP confère aux seuls programmeurs la responsabilité de l'estimation des charges et des délais. Ni le client, ni le manager, ni le coach ne peuvent influencer les programmeurs dans leurs estimations, pas plus qu'ils ne peuvent leur « mettre la pression ». L'estimation des délais est laissée aux programmeurs eux-mêmes : c'est là une liberté certaine mais aussi une lourde responsabilisation.

Le rôle de programmeur XP est donc varié – au risque d'en effrayer certains, tellement il semble exiger de compétences et de qualités humaines. Heureusement, les différentes pratiques XP décrites plus loin fournissent un cadre très précis qui facilite l'apprentissage ou la diffusion de ces compétences.

La charte des droits du programmeur

Le programmeur retrouve une place centrale dans un projet XP et sa situation est même protégée par la charte suivante :

« Développeur, vous avez le droit…

- de savoir ce qui est demandé, avec des priorités clairement déclarées ;
- de fournir un travail de qualité en toute occasion ;
- de demander et recevoir de l'aide de la part de vos pairs, de vos clients ;
- d'émettre et de réviser vos propres estimations de coûts ;
- d'accepter des responsabilités, qui ne peuvent vous être imposées ;
- de travailler à un rythme de travail durable. »

Pratiques XP du programmeur

Si vous pensez être ou souhaitez devenir un programmeur XP, voici les pratiques XP qui vous concernent au premier chef (et auxquelles vous pouvez donc vous reporter dès maintenant) :

- la programmation en binôme (voir chapitre 5) ;
- les tests unitaires (voir chapitres 3 et 4) ;
- la conception simple (voir chapitre 3) ;
- le remaniement (voir chapitre 3) ;
- la responsabilité collective du code (voir chapitre 5) ;
- les règles de codage (voir chapitre 5) ;
- l'intégration continue (voir chapitre 5) ;
- le rythme durable (voir chapitre 6).

Client

Puisque le programmeur doit écouter le client, décrivons maintenant plus précisément le rôle de ce dernier.

Important
Le rôle de client XP n'est pas forcément tenu par le client contractuel (voir aussi chapitre 9).

Un projet XP sert à développer un logiciel. Le *client* a pour responsabilité de définir ce que doit faire ce logiciel, et de quelle façon, et de communiquer ces informations aux programmeurs. Jusque-là, rien de surprenant. On pourrait même imaginer que c'est un rôle facile.

La charte des droits du client
Dans un projet XP, le client est roi – mais pas despote. Ses droits et privilèges sont détaillés dans cette charte :
« Client, vous avez le droit…
- à un plan d'ensemble, montrant ce qui peut être accompli, pour quand, à quel coût ;
- d'obtenir le plus de valeur possible de chaque semaine de programmation ;
- de voir des progrès sur une application qui marche, comme doivent le prouver les tests répétables que vous spécifiez ;
- de changer d'avis, de substituer des fonctionnalités et de changer vos priorités sans en payer un prix exorbitant ;
- d'être informé des modifications portées au calendrier de réalisation, assez tôt pour avoir la possibilité de réduire le périmètre fonctionnel et retomber ainsi sur la date de livraison initiale ;
- d'annuler le projet à tout moment et de disposer d'une application utile et utilisable en contrepartie de votre investissement à ce jour. »

Pourtant, le client doit lui aussi appliquer les principes XP : faire preuve de simplicité et de cran, et favoriser la communication et le feedback au sein du projet.

En effet, il est essentiel que ces quatre valeurs soient respectées par le client. Par exemple, un client qui n'est pas convaincu des vertus de la simplicité risque de déployer des trésors d'imagination et d'inventer des fonctionnalités inutiles ou irréalisables ; il entretiendra une «vision» grandiose d'un système «intégré», d'une «vitrine technologique»… Un tel client souhaitera disposer de tous les gadgets possibles et imaginables, sans pouvoir dire en quoi ils lui sont utiles, mettant en danger l'aboutissement du projet. Les cimetières de projets logiciels sont pleins de systèmes intégrés et de vitrines technologiques.

Nécessité du feedback pour le client

Un client qui ne communique pas ne saura jamais expliquer aux développeurs ce qu'il souhaite et a donc peu de chances d'être satisfait. Mais un client qui communique très bien peut se tromper, attacher beaucoup d'importance à une fonctionnalité, ou être persuadé que telle ou telle fonction graphique est la plus ergonomique, jusqu'au moment où le système lui est livré tel qu'il l'avait souhaité… et qu'il prend conscience de son erreur. C'est là qu'intervient le feedback. Un feedback important et rapide lui permettra de découvrir ses erreurs avant qu'il ne soit trop tard.

Il n'y a pas que le client qui peut se tromper. Peut-être a-t-il raison de croire en tel ou tel gadget, telle ou telle vision, que les programmeurs souhaitent sacrifier à l'autel de la simplicité. Dans ce cas, il obtiendra gain de cause, parce que le client est roi, mais aussi parce que l'équipe est disposée à faire un essai sachant qu'elle aura un feedback rapide sur la pertinence des idées du client. Et les résultats obtenus pourraient bien convaincre les programmeurs du bien-fondé de ce que leur demande le client !

La communication entre le client et les programmeurs est donc capitale, pour que les programmeurs comprennent les besoins du client, afin d'instaurer un feedback fort et rapide, qui à son tour permettra de canaliser le projet sur la simplicité.

Le client sur site

La forme la plus extrême de communication consiste à réunir les protagonistes en permanence dans la même pièce, et c'est la solution proposée par la méthode XP. Cela ne suffit pas, et la méthode XP fournit tout un protocole favorisant et encadrant cette communication. Mais la proximité entre le client XP et les développeurs est incontournable et est en elle-même un facteur radical de communication. Finis, les cahiers des charges vagues ou incompréhensibles, les courriers de questions techniques qui restent sans réponse plusieurs semaines – lorsque les réponses arrivent. Finies aussi les démos plus ou moins truquées, les réunions d'avancement où les angoisses secrètes sont passées sous silence. Client et programmeurs sont face à face, à nu. Et, pour cela, il faut un minimum de courage.

Dans cette situation, le client se trouve donc sur le même site, de préférence dans le même bureau, en tout cas à portée de voix des programmeurs. Il est aisé d'imaginer qu'une commu-

nication orale et informelle s'instaure, voire devient primordiale. C'est tout à fait l'objectif recherché. Cela ne signifie pas pour autant que la communication orale soit suffisante. Au contraire, la méthode XP exige du client une discipline rigoureuse, celle des scénarios client.

Le client spécifie par des scénarios client…

Le client commence par rédiger les fonctionnalités qu'il attend du logiciel sous forme de *scénarios client* (*user stories)*. Chaque scénario client est une description informelle, narrative, d'une fonctionnalité du système, d'une interaction entre le système et un utilisateur. De préférence, il s'agit d'une histoire complète, avec un début et une fin. L'utilisateur doit avoir tiré un bénéfice du logiciel, à moins que le scénario client ne décrive un cas d'erreur ou un cas dégradé. Mais le principal est de ne pas se formaliser, de faire le plus simple possible. Si la notion de scénario n'est pas étrangère aux cas d'utilisation d'UML (*use case*), on est tout de même loin du formalisme qui leur est souvent associé. Pour favoriser la simplicité, et pour d'autres raisons qui deviendront apparentes lors de la description des séances de planification (*planning game*) au chapitre 6, le client rédige chaque scénario client à la main sur une fiche bristol de taille A5.

Le client doit éviter de s'enfermer des semaines durant pendant qu'il rédige la totalité de ses scénarios. En effet, il faut que les programmeurs aient connaissance le plus tôt possible des principaux scénarios client, afin de démarrer la phase d'exploration (voir chapitre 6), qui consiste à définir plus précisément la liste des besoins et à évaluer les coûts de développement correspondants. En outre, le client sera certainement amené à récrire ces scénarios, soit pour y consigner des précisions relatives à des questions des programmeurs, soit parce que ces derniers lui demanderont lors des séances de planification d'en fusionner plusieurs ou au contraire d'en scinder un, trop compliqué – l'objectif étant d'obtenir des scénarios client dont les durées d'implémentation évaluées soient relativement homogènes.

L'idéal est de commencer par préparer une liste des titres donnés à chaque scénario client, de réunir tout le monde et de raconter oralement ce qu'on entend mettre derrière chaque titre. Cela suffira déjà aux programmeurs pour appréhender le logiciel dans son ensemble, pour identifier les scénarios les plus risqués et commencer la phase d'exploration. Si l'équipe comprend un coach ou quelques programmeurs expérimentés avec la méthode XP, cette discussion permettra même de réorganiser les scénarios en les fusionnant ou en les scindant. Il y a de fortes chances aussi pour que la discussion aide le client lui-même à clarifier ses idées, à identifier ses priorités, à prendre conscience de la difficulté que présentent certaines fonctionnalités ou d'avoir des idées de fonctionnalités auxquelles il n'avait pas encore pensé.

Ensuite, le client peut rédiger une dizaine de scénarios client, en commençant peut-être par ceux qui sont le plus clairs pour lui. Il peut se faire aider du coach pour commencer. Il réunit à nouveau tout le monde pour faire une lecture de ces principaux premiers scénarios. Les programmeurs posent des questions jusqu'à ce que tout leur paraisse clair. Le client utilise ce feedback pour améliorer sa façon d'écrire ou d'organiser ses scénarios, tout en affinant progressivement ses besoins et sa compréhension de ce que les programmeurs peuvent

fournir. Et ainsi de suite pendant toute la phase d'exploration, avec une seule nuance : dès que le client commence à se familiariser avec l'écriture des scénarios client, il se concentre sur ceux qu'il considère comme les plus importants (plutôt que ceux qu'il trouve plus faciles à rédiger).

Remarque

La phase d'exploration peut s'arrêter avant que le client n'ait achevé tous ses scénarios. Ce qui importe, en revanche, c'est que la liste des titres de scénarios ait commencé à se stabiliser et que chacun sache à peu près ce qui se cache derrière chaque titre.

C'est alors que se tient la première séance de planification (*planning game)* visant à produire le premier plan de livraison. Auparavant, le client aura pris soin d'attribuer à chaque scénario une priorité : indispensable, essentiel ou utile. Le déroulement exact d'une séance de planification est décrit au chapitre 6. Ce qui nous intéresse à ce stade, c'est le rôle du client, et, précisément, lors de cette séance, son rôle est de répartir ses scénarios client dans le temps. Le projet est découpé en tranches de deux ou trois semaines, les itérations. En simplifiant un peu, on notera que c'est à ce moment que les programmeurs disent au client combien de scénarios ils peuvent réaliser dans chaque itération, et que le client décide de la répartition des scénarios dans chaque itération.

Le client peut même changer d'avis, d'autant plus qu'à la fin des trois semaines, il est déjà en possession d'un système opérationnel, utile, et dont les fonctionnalités sont celles qu'il a identifiées comme étant les plus urgentes. Bref, il aura du feedback. Et c'est en se rendant compte à quel point ce feedback est utile qu'il commencera à modifier sa répartition des scénarios dans les itérations, voire à les récrire !

Remarque

Le fait que des fonctionnalités puissent être réalisées aussi rapidement est grisant pour le client mais, ce qui le ramènera chaque fois à la raison, c'est que ce sont les programmeurs, et eux seuls, qui pourront se prononcer sur le délai nécessaire à la réalisation de chaque scénario.

Le client est donc seul habilité à décider des fonctionnalités du système. Les programmeurs n'influent sur ses décisions qu'en lui fournissant des estimations honnêtes des délais, et surtout en lui donnant un feedback concret toutes les deux ou trois semaines avec une nouvelle version du logiciel.

…et spécifie encore par des tests de recette

Une fois que le client a décidé quels scénarios seraient réalisés dans la prochaine itération, un nouveau pan de son activité commence. Car les programmeurs peuvent parfois être – par nécessité professionnelle – des gens pointilleux. Pour pouvoir programmer, ils ne se conten-

tent pas des scénarios. Ils veulent des chiffres. Des valeurs. Des données concrètes. Pour lever les ambiguïtés des scénarios, pour en préciser les contours, pour que les choses soient absolument claires, ils vont demander au client de spécifier des tests de recette.

Dans la méthode XP, les tests de recette sont des données d'entrée pour les programmeurs. Le client devra dire dès le départ ce qu'il considère comme étant une preuve que le système fait ce qu'il demandait. Si ce qu'il demande n'est pas démontrable, les programmeurs ne peuvent être contraints de le faire. En contrepartie, le client n'aura pas à se satisfaire des promesses des programmeurs. Dès la fin de l'itération, les tests de recette doivent tous passer avec succès. Sinon, on considère que les scénarios client ne sont pas terminés. Environ à mi-chemin d'une itération, le client, avec l'aide du testeur (voir plus bas), doit fournir aux programmeurs un jeu de tests de recette correspondant aux scénarios client affectés à l'itération.

Qui peut jouer le rôle du client ?

Qui peut jouer le rôle de client ? L'idéal est que ce soit le vrai client : celui qui paie (maître d'ouvrage), ou quelqu'un qui le représente (assistant à maîtrise d'ouvrage, ou représentant des utilisateurs). À la base, c'est ce que requiert la méthode XP, et c'est dans ces cas de figure qu'on en tire un maximum de bénéfices, tant pour le client que pour les fournisseurs. Cela ne va pas sans implications contractuelles, lesquelles sont étudiées au chapitre 9.

Il n'est parfois pas envisageable que le vrai client (payeur ou utilisateur final) joue ce rôle, pour des raisons contractuelles ou géographiques, par exemple. Dans ce cas, il faut désigner un client artificiel qui puisse tenir ce rôle auprès de l'équipe. Cela peut être un donneur d'ordres, un chef de projet par exemple, ou un ingénieur chargé des spécifications. Ce qui compte, c'est qu'il soit investi de suffisamment de pouvoir pour prendre des décisions. L'avantage de cette approche est qu'on peut choisir un individu possédant les qualités requises d'un client XP : simplicité, communication, courage.

En revanche, ce client artificiel sera lui-même confronté aux problèmes classiques de communication et de négociation avec le client final. Malgré tout, la méthode XP porte ses fruits : les programmeurs disposent par le biais du client XP d'un guichet unique, un interlocuteur toujours disponible, habilité à trancher pour définir les détails du logiciel demandé. L'obtention – parfois difficile – des informations auprès du client final ne repose ainsi que sur une seule personne. Et très rapidement, le client XP dispose d'un début de logiciel, incomplet au regard des besoins de départ mais en parfait état de marche, qu'il pourra présenter au client final afin d'obtenir du feedback et des informations. Lorsque le client final commencera à percevoir que les fournisseurs sont productifs, disposés à être souples, à accepter ses changements de spécifications en cours de route, il sera plus enclin à «jouer le jeu», voire à se détendre un peu lors des discussions sur les spécifications.

Pratiques XP du client

Voici les pratiques de la méthode XP qui concernent le plus directement le client (outre le fait même d'avoir un client dans l'équipe) :

- planification itérative comprenant la rédaction des scénarios client et les séances de planification (*planning game*) [voir chapitre 6] ;
- tests de recette (voir chapitre 3) ;
- intégration continue (voir chapitre 5) ;
- rythme durable (voir chapitre 6).

Testeur

Le client doit donc préciser les scénarios client en spécifiant des tests de recette. Une exigence forte de la méthode XP est que tous les tests, et ce tant les tests unitaires qui sont l'affaire des programmeurs que les tests de recette spécifiés par le client, doivent être automatisés. Cela implique la mise en place d'outils informatiques spécifiques et l'écriture de scripts de test : c'est là qu'intervient le rôle de testeur.

Le testeur est le bras droit du client. On a vu que la seule maîtrise concrète que peut exercer le client sur les programmeurs, c'est de fournir des tests de recette que le logiciel doit réussir. En consignant les tests dans ses scripts, le testeur est donc le garant de ce que fera réellement le logiciel.

Pendant la phase d'exploration, le testeur doit mettre en place son arsenal d'outils. Tester fonctionnellement un système peut en effet nécessiter plus d'un outil. Cela dépend beaucoup du nombre d'interfaces, ainsi que des technologies utilisées par les programmeurs. Pendant que ces derniers explorent leurs propres outils, le testeur se tient informé de leurs choix technologiques et prépare sa parade.

La testabilité doit être considérée par tous, client et programmeurs compris, comme un facteur essentiel de réussite du projet. Le testeur peut ainsi déconseiller telle ou telle fonctionnalité au client en raison des difficultés que l'on peut rencontrer à la tester, ou déconseiller telle ou telle technologie aux programmeurs. Il participe aux réunions de conception des programmeurs pour s'assurer que le logiciel demeure testable. Il peut même exiger d'eux certains développements spécifiques sans lesquels il ne pourrait pas automatiser les tests de recette (c'est souvent le cas pour les tests d'interfaces graphiques).

Une fois que les itérations sont commencées, le testeur travaille avec le client en début d'itération pour définir et programmer les tests de recette. Ces derniers doivent correspondre aux scénarios client, qui doivent chacun comporter au moins un test de recette. Le plus tôt possible, au plus tard à mi-parcours de chaque itération, le testeur fournit aux programmeurs un nouveau jeu de tests correspondant aux scénarios client sur lesquels ils travaillent actuellement. Pour les programmeurs, ce jeu de tests devient la carotte qui les fait avancer. Ils savent que, si ces tests passent, ils en auront terminé !

Cette façon de travailler est très motivante pour les programmeurs, qui sentent concrètement qu'ils progressent chaque fois qu'un test fonctionnel de plus est réussi, et elle est très satisfai-

sante pour le client, qui voit petit à petit son logiciel prendre vie. C'est la responsabilité du testeur d'entretenir cette motivation par tous les moyens possibles, de façon typique en affichant un grand graphique au mur qui montre la progression des tests de recette, en faisant sonner une cloche chaque fois qu'un nouveau test de recette passe, ou tout autre «truc», tous les moyens sont bons ! L'important est de communiquer à chacun la sensation de progrès, de réussite.

Qui peut être testeur ?

Quel est le profil idéal d'un testeur ? On l'a vu, c'est un rôle qui allie la connaissance du métier et des fonctionnalités du système aux compétences techniques requises par l'outillage de test. Techniquement, le testeur est souvent un programmeur hétéroclite, bricoleur, capable de combiner différents outils pour arriver à ses fins. Il est rigoureux, car il est le garant de ce que fait le logiciel. Il est intègre, car ce sont les tests de recette qui font foi de l'avancement technique du projet. Il faut qu'il sache déceler et mettre au jour les erreurs du logiciel sans être rabat-joie, sans entamer le moral des programmeurs, sans briser l'esprit d'équipe.

Pratiques XP du testeur

Le testeur est donc plus particulièrement concerné par les pratiques XP suivantes :

- les scénarios client et les métriques (suivi des tests) de la planification itérative (voir chapitre 6) ;
- les tests de recette (voir chapitre 3) ;
- l'intégration continue (voir chapitre 5) ;
- le rythme durable (voir chapitre 6).

Tracker

Au début de chaque itération, une séance de planification (*planning game*), décrite plus en détail au chapitre 6, permet de définir un plan d'itération. Ce plan comporte toutes les tâches que les programmeurs devront effectuer pour réaliser les scénarios client sélectionnés par le client. En face de chaque tâche, un nom et une estimation de la charge sont mentionnés.

Le rôle du tracker consiste à faire le suivi de ces tâches en cours d'incrément. Au moins deux fois par semaine, ou plus fréquemment encore, selon les circonstances, le tracker se rend auprès de chaque programmeur pour discuter avec lui de l'avancement de sa tâche en cours. L'objectif est de détecter les difficultés le plus tôt possible pour y remédier.

D'une façon typique, le tracker a sous les yeux le plan de l'itération. Il demande au programmeur comment sa tâche progresse. Celui-ci lui raconte où il en est, les problèmes qu'il a rencontrés ou qu'il connaît encore. Le tracker consulte le plan d'itération et remarque que la tâche devait prendre trois jours. Deux jours se sont écoulés. Qu'en pense le programmeur ?

Combien de temps lui faut-il pour terminer la tâche en question ? Tout l'art du rôle de tracker réside dans la façon de procéder. Il doit éviter de mettre le programmeur sur des charbons ardents. Il doit savoir délier les langues. Son rôle est de détecter les problèmes avant qu'il ne soit trop tard, avant que le délai ne soit passé. En l'occurrence, il doit faire en sorte que la tâche de trois jours en prenne quatre et non six, simplement parce que le problème aura été pris assez tôt.

Remarque

Le tracker ne prend aucune mesure de son propre chef. Il sert seulement de révélateur. Une fois qu'il a discuté avec les programmeurs, s'il pense avoir détecté un problème potentiel, une inquiétude, il en réfère au coach. C'est à ce dernier d'envisager les solutions, ce qu'il fera d'ailleurs en concertation avec les programmeurs.

Qui peut être tracker ?

Le tracker idéal sera incarné par un individu plutôt affable, qui ne donne pas l'impression de contrôler, et à qui on se confie volontiers. Ce ne sera probablement pas un supérieur hiérarchique des programmeurs. De préférence, mieux vaut qu'il ne soit pas programmeur lui-même ; il faut qu'il en connaisse assez pour comprendre ce que lui racontent les programmeurs, sans toutefois que sa tournée ne se transforme en discussion technique.

Même s'il n'y participe pas, le tracker est concerné par la pratique XP des séances de planification (*planning game*), car il doit comprendre comment les estimations de charge sont faites. Le plan d'itération doit lui être transmis dès que la séance de planification est terminée.

Manager

Le manager est le supérieur hiérarchique des programmeurs. Il est aussi parfois le supérieur hiérarchique du client, lorsqu'il s'agit d'un développement en interne, ou lorsque le client XP n'est pas le client final. C'est le manager qui s'occupe matériellement des programmeurs : il leur fournit un bureau, des machines, il les paie à la fin du mois. Il recrute des programmeurs supplémentaires ou un testeur lorsque l'équipe en exprime le besoin.

Le manager demande aussi des comptes. Il demande à l'équipe XP de faire ses preuves. Les résultats doivent être concrets et visibles, c'est une des promesses de la méthode : productivité, visibilité, reporting fréquent et honnête. Grâce à lui, l'utilisation de la méthode XP ne peut pas devenir un exercice de style, une arlésienne. Il se tient informé des plans de livraisons et d'itération, et veille à leur respect. Il participe donc aux séances de planification, peut-être pas à chaque itération mais au moins chaque fois que le plan de livraison doit être modifié.

Pour que la méthode XP réussisse, le manager doit avoir le courage de laisser se dérouler le processus. Il doit éviter par exemple d'exiger que telle fonctionnalité soit prête à telle date si les programmeurs annoncent que c'est impossible. Il peut et doit exiger que les engagements

soient respectés, que les avantages promis par la méthode se confirment, mais sans que cette dernière ne soit dévoyée. Cela lui demandera parfois du cran.

Lorsque le logiciel à développer fait partie d'un projet plus large (par exemple, dans le cas d'un système à logiciel prépondérant mais avec des composants matériels, ou d'autres logiciels développés indépendamment), le manager participe aux réunions techniques ou réunions d'avancement générales du projet. Il transmet au monde extérieur l'état des lieux du développement logiciel, et résume à l'équipe logicielle les informations en provenance de l'extérieur.

Il vérifie de temps en temps que le client est toujours satisfait !

Qui peut être manager ?

Traditionnellement – s'il y a transition depuis une gestion de projet classique –, qui assume le rôle de manager ? Il s'agit en général du chef du service auquel appartient l'équipe de développement. Dans un projet de taille importante, où le logiciel développé en XP n'est qu'un sous-projet, le rôle de manager peut être joué par le chef du projet global, voire par un responsable ou coordinateur technique.

> **Remarque**
> Contrairement au coach, le manager ne fait pas partie intégrante de l'équipe. Il suit l'avancement du projet, en fournit les moyens humains et matériels, et demande des comptes. C'est au coach, qui travaille quant à lui au sein de l'équipe, de la faire fonctionner et de trouver les solutions en cas de problème.

Pratiques XP associées au rôle de manager

Le manager ne participe (partiellement d'ailleurs) qu'aux séances de planification, mais il doit avoir une compréhension générale de la méthode XP. Les pratiques qui le concernent plus directement sont les suivantes :

- les scénarios client, les séances de planification et les métriques (avancement des tests de recette, vélocité de projet…) de la planification itérative ;
- le rythme durable.

Coach

Nous évoquons le rôle du coach en dernier non pas parce qu'il est moins important, bien au contraire, mais parce qu'il revient au coach de réunir tous les rôles précédents et de faire en sorte que cela marche, que la magie XP opère.

Le coach est le véritable garant du processus lui-même. Il s'attache à vérifier que chacun joue son rôle, que les pratiques XP sont respectées, que l'équipe fait ce qu'elle a dit qu'elle ferait.

Si c'est la première fois que la méthode XP est utilisée, c'est au coach d'articuler la mise en place de la méthode, de sélectionner les pratiques prioritaires, de veiller à ce que chacun comprenne son rôle. Cela deviendra ensuite l'affaire de chacun d'endosser son rôle, de maîtriser les pratiques XP qui le concernent directement. La méthode elle-même et le processus utilisé par le projet deviendront ainsi, peu à peu, l'affaire de tous. Mais plus encore celle du coach ! Il doit avoir une compréhension plus profonde, plus intime de la méthode XP pour trouver des solutions de rechange, pour peaufiner des adaptations sans dénaturer l'esprit de la méthode XP.

Au début du projet, le coach doit donc être partout. Il organise et anime les séances de planification et les *standup meetings*. Il aide le client à rédiger ses premiers scénarios. Il travaille avec les programmeurs jusqu'à ce qu'ils aient perçu les principes de la programmation XP. Il rassure le manager en lui expliquant pourquoi la méthode fonctionne.

Toutefois, l'objectif absolu du coach, c'est que l'équipe fonctionne sans lui. Petit à petit, il doit s'effacer, être moins directif, laisser l'équipe découvrir ses propres solutions techniques, laisser chacun s'imprégner de la méthode XP. Lors des réunions techniques, il n'impose pas sa solution ; il favorise la créativité du groupe. Au besoin, il arbitrera mais, plutôt que d'expliquer sa solution, il amènera l'ensemble du groupe à la trouver.

Le coach doit aussi avoir le courage de dire les choses telles qu'elles sont. Il a même le droit de se fâcher, de temps en temps, si cela ne rompt pas la cohésion du groupe, ni ne frustre un individu.

Qui peut être coach ?

Il est fréquent, mais pas indispensable, que le coach ne soit pas seulement un expert de la méthode XP, mais un expert technique, un programmeur chevronné et un architecte. Si c'est le cas, son but, là encore, est que l'équipe devienne autonome. Il doit faire en sorte que les programmeurs rentrent tous dans un cycle d'amélioration continue, de recherche de l'excellence. La méthode XP fera beaucoup pour faciliter les choses, grâce notamment à la programmation en binôme, mais il peut être souhaitable d'aller plus loin, en organisant par exemple des topos techniques réguliers.

À tout moment, même une fois que l'équipe est devenue autonome, le coach doit être ouvert et disponible. C'est à lui que le tracker viendra faire part d'un problème. Les membres de l'équipe doivent se sentir libres de lui demander conseil. Le coach doit être une sorte de mentor méthodologique et technique pour l'ensemble des autres rôles.

La première qualité qui est exigée pour jouer le rôle de coach est d'être intimement convaincu par la méthode XP ! À part cela, un bon coach doit être communicateur, pédagogue et sensible. Il est important qu'il sache déceler l'inquiétude, l'incompréhension ou la démotivation des membres de l'équipe. Il doit faire preuve de beaucoup de sang-froid, et rester calme lorsque tout le monde est en train de paniquer.

Évidemment, le coach doit connaître parfaitement l'ensemble des pratiques et s'imprégner de la culture et de l'esprit qui animent l'Extreme Programming.

Répartition des rôles

Nous avons présenté les six rôles principaux, indispensables à la pratique de la méthode Extreme Programming. Pourtant, il est tout à fait possible de pratiquer XP avec moins de six personnes. Cela implique que certaines d'entre elles cumulent plusieurs rôles. Certaines combinaisons ne posent aucun problème. Il est néanmoins utile que l'individu se souvienne (et que le coach lui rappelle) qu'il joue bien plusieurs rôles, et qu'il sache à tout moment quel rôle il est en train de jouer.

Inversement, certains rôles peuvent, ou même doivent dans la mesure du possible, être joués par plusieurs personnes.

Plusieurs rôles pour une même personne

Lorsque le client possède les compétences techniques requises, il est très fréquent qu'il joue aussi le rôle de testeur. Toutefois, le client ne peut être combiné avec aucun autre rôle que testeur.

Non seulement un programmeur ne doit jamais jouer le rôle de client, mais il est préférable qu'il ne joue pas le rôle de testeur. Dans le cas de très petites équipes où le client n'aurait pas les compétences techniques de testeur, c'est toutefois envisageable à condition de trouver une façon claire, un certain formalisme, par lequel le client exprime lui-même les scripts de tests de recette, qu'un programmeur n'a plus qu'à traduire dans un langage de programmation.

Le manager ne devrait jouer aucun autre rôle, à moins qu'il ne soit particulièrement proche des programmeurs (ce qui est exceptionnel pour un manager), auquel cas il peut jouer le rôle de tracker. Il est d'ailleurs assez difficile de trouver un bon tracker. Ce ne peut pas être le coach, puisque celui-ci doit parfois savoir se fâcher avec les programmeurs. Ce ne peut pas être le client ou le testeur, car les programmeurs risquent de leur cacher certains problèmes. Il est préférable de prendre une personne extérieure au projet. Cela pourrait être un programmeur qui travaille à un autre projet, à condition qu'il sache éviter que sa tournée de tracking ne se transforme en une discussion technique sans fin. Malgré tout, dans certains projets XP, le rôle de tracker a été institué comme un rôle tournant à chaque itération parmi les programmeurs du projet, et certains coachs font eux-mêmes le tracking.

Que le coach soit ou non programmeur, cela semble relever de l'appréciation personnelle. La difficulté, pour un coach qui est en même temps programmeur, réside dans sa capacité à se détacher du projet, à donner plus d'autonomie à l'équipe. S'il est en même temps programmeur, il travaillera de plus en plus en tant que programmeur, et deviendra de plus en plus utile, au risque de rester incontournable !

Le tableau présenté ci-après indique les bonnes combinaisons (✔), les mauvaises (✘) et celles qui sont envisageables mais non sans risque (~).

Tableau 2-1. Combinaisons de rôles XP

	Programmeur	Client	Testeur	Tracker	Manager	Coach
Programmeur		✘	~	~	✘	~
Client	✘		✔	✘	✘	✘
Testeur	~	✔		✘	✘	✘
Tracker	~	✘	✘		~	~
Manager	✘	✘	✘	~		✘
Coach	~	✘	✘	~	✘	

Il n'y a sans doute pas de règle absolue en la matière, car la personnalité de chacun est déterminante. Il est important de bien comprendre chaque rôle individuel avant d'envisager des combinaisons, et de s'assurer que l'individu qui cumule des rôles n'en vienne pas à sacrifier une composante importante d'un de ses rôles.

Plusieurs personnes pour un même rôle

Le rôle de programmeur est en général celui qui est joué par le plus grand nombre de personnes ; on peut en compter une dizaine sans que pour autant il soit nécessaire d'adapter la méthode. En outre, pour pratiquer la programmation en binôme, il faut être au moins deux, même si l'un des deux peut être amené à cumuler les rôles de coach et de programmeur.

Le rôle de tracker nécessite en général une seule personne… à un moment donné. Il arrive que ce rôle soit désigné par rotation parmi les programmeurs, car il peut être difficile de trouver une ressource qui puisse dégager 30 minutes tous les deux jours pour discuter avec les programmeurs ; le coach n'est pas tenu de remplir ce rôle.

Coach et manager, en revanche, sont forcément représentés par des personnes uniques, le premier en raison de son rôle de pivot, de coordination et d'arbitrage, le second en raison de son rôle hiérarchique.

Le cas du client est moins tranché. Si la configuration typique désigne une seule personne pour jouer ce rôle, il est tout à fait possible qu'il soit joué par une équipe. Tout d'abord, nous avons souligné la forte synergie qui lie le client et le testeur. Ce tandem peut très bien se considérer comme une équipe de deux personnes qui cumulent le rôle de client et testeur… et travailler en binôme, comme les programmeurs !

Sur certains projets, il est probable qu'aucune personne ne réunisse les compétences métiers couvrant tous les aspects requis du logiciel pour pouvoir assurer le rôle du client, ou que ce rôle implique une charge de travail trop importante pour une seule personne. C'est le cas par exemple si cette dernière ne peut pas se consacrer à temps plein au projet, ou lorsqu'il y a plus de dix programmeurs. Dans ces cas, «le client» est en fait une équipe de clients.

Important

Une chose est alors capitale : cette équipe doit parler d'une seule voix. Il est donc préférable que ses membres se réunissent au préalable pour préparer chaque séance de planification et pour désigner un porte-parole.

Comparaison avec une organisation d'équipe classique

En fait, il n'existe pas d'organisation classique, ni de rôles standards. Un chef de projet, par exemple, peut aussi bien être un développeur intégré à une équipe de trois qui sert d'interlocuteur, d'interface, avec la hiérarchie, ou un ingénieur d'affaires qui ne gère que les aspects contractuels et/ou relations clients, ou encore un développeur confirmé qui ne fait plus que de la documentation et du suivi de planning. On retrouve les mêmes variations pour tous les autres rôles classiques : responsable technique, responsable de module, ingénieur qualité, testeur, valideur, etc. Cela dépend de la culture d'entreprise, des circonstances (taille du projet, habitudes des individus), parfois aussi des clients et de leurs exigences.

Il est intéressant toutefois de souligner l'esprit dans lequel la répartition des rôles se fait au sein d'un projet XP, et en quoi cela peut différer de certaines organisations (même s'il faut rappeler que l'Extreme Programming n'a rien inventé, certaines organisations suivant sûrement les mêmes principes…).

En premier lieu, XP s'attache à séparer ce qui relève des compétences métier (en anglais *business* ou *domain knowledge*) de ce qui relève des compétences techniques. Par exemple, ce n'est pas aux développeurs, même s'ils sont expérimentés dans leur domaine d'application, de décider de l'importance relative des fonctionnalités, ou d'introduire des fonctionnalités dont ils sont persuadés que le client aura besoin (rien ne les empêche de le proposer, mais la décision revient au client). Inversement, ce n'est pas au client à estimer les charges et les délais pour le développement.

Cette séparation stricte n'est pas toujours observée dans les organisations classiques, où il est fréquent que le chef de projet soit à la fois responsable de la rédaction des spécifications et de l'établissement du budget et du planning.

Si l'expérience d'un développeur dans son domaine d'application l'autorise à penser qu'il est mieux placé que le client pour dire ce qu'il faut faire, il faut envisager de le mettre dans l'équipe client ; auquel cas, il ne fera plus de développement (ses compétences techniques étant alors certainement utiles à l'écriture des tests de recette…).

Une autre caractéristique très forte d'une organisation XP est qu'il n'y a aucune hiérarchie ni aucune séparation des tâches au sein des programmeurs. Il n'y a pas d'architecte, ni de concepteurs, ni de testeurs. Cette particularité est due à la conviction que toutes les activités des programmeurs relèvent en fait de la conception, et doivent donc être pratiquées de manière intégrée, mais aussi à la forte croyance dans la puissance du travail collectif dans le contexte d'une équipe qui se structure naturellement et spontanément, et évolue avec le temps.

Enfin, la pratique de responsabilité collective du code a un très fort impact sur l'organisation d'équipe. Il est fréquent d'observer, dans les projets traditionnels, un partitionnement en sous-projets, modules et sous-modules, souvent au point que chaque développeur est responsable, individuellement, d'une partie du logiciel (module ou liste de classes…). Cela engendre tout un ensemble de pratiques (livraisons internes, demandes de modifications, fiches de versions, branches, etc.) qui sont inconnues sur un projet XP. Sans parler des problèmes causés par le départ d'une personne…

Quelques comportements contre-indiqués

L'une des « faiblesses » de la méthode XP, si l'on peut appeler cela une faiblesse, est que l'importance accordée à l'esprit d'équipe et au travail en binôme rend le fonctionnement de l'équipe particulièrement vulnérable face aux comportements qui ne s'intègreraient pas à la méthode. On peut citer quelques-uns de ces comportements :

- une personne qui s'attribuerait le mérite en cas de succès tandis qu'en cas de problèmes, elle en rejetterait la faute sur les autres ;

- un codeur même extrêmement compétent qui ferait des choses que personne ne comprend et ne voudrait pas travailler avec des programmeurs moins compétents. S'il refuse de modifier ses habitudes de codage au profit des conventions définies par l'équipe, il vaut mieux l'écarter du projet ;

- un programmeur qui souhaiterait occuper le statut de chef ou être perçu par les autres comme l'architecte du système ; acceptant toutes les tâches, participant à toutes les réunions, il cherche à se rendre indispensable ; connaissant le système par cœur et dans tous ses recoins, il cherche à travailler seul en évitant de partager ses connaissances ou ses compétences.

De tels comportements détruisent l'esprit d'équipe, et le coach comme le manager doivent faire preuve de fermeté (et d'une certaine forme de courage) pour veiller à ce qu'ils ne se produisent pas.

Dispersion géographique

Contrairement aux équipes qui sont constituées sur un mode d'organisation plus classique, il n'est absolument pas possible qu'une équipe XP soit répartie sur plusieurs sites, même peu éloignés géographiquement. Comme c'est le cas pour les équipes trop petites, il sera toujours possible d'utiliser certains aspects de la méthode, mais il ne s'agira plus vraiment d'un développement en XP.

En revanche, d'autres méthodes agiles (voir chapitre 10) permettent ce genre de développement. Le mode de développement *open source*, lui-même, qui est le cas le plus extrême où des développeurs sont dispersés géographiquement, est d'ailleurs considéré par l'Alliance agile[1] comme étant conforme à ses principes.

Le cas particulier du rôle de consultant

Au sein d'une équipe XP, la diffusion des connaissances et le travail collectif sont tellement importants qu'il est rare qu'elle soit composée d'experts en une technologie particulière. Pourtant, l'équipe devra parfois s'appuyer sur des connaissances pointues pour résoudre un problème donné. C'est le moment de faire appel à un consultant.

L'intervention d'un consultant sur un projet XP doit se dérouler comme suit :

1. Tout d'abord, l'équipe fait appel à un consultant sur un problème précis. Tellement précis qu'elle aura déjà créé un test vérifiant ce qu'elle souhaite obtenir. À défaut, la première tâche confiée au consultant consistera à détailler un tel test.

2. Ensuite, le consultant ne travaille pas seul. Un ou plusieurs programmeurs (probablement deux) collaborent en permanence avec lui. En effet, l'équipe ne veut pas simplement que son problème soit résolu : elle sait qu'elle rencontrera d'autres problèmes analogues et elle veut être en mesure de les résoudre seule la prochaine fois.

3. Enfin, lorsque le consultant a terminé, l'équipe souhaitera probablement recommencer elle-même ce qu'il a fait. Le consultant ne doit pas s'en offenser : c'est une façon assez répandue de travailler et de faire de la conception chez les programmeurs XP.

Quelle taille pour les équipes XP ?

Il est (trop) souvent répété que la méthode XP ne s'applique qu'à des équipes de moins de dix personnes. Ce mythe provient de l'attitude honnête et pragmatique des pionniers d'XP qui, à la question de la taille maximale d'une équipe XP, ont fait valoir : «Nous savons que cela marche pour des équipes comptant jusqu'à dix personnes. »

Depuis, des équipes XP de trente personnes ont fait part de leur succès dans l'utilisation de la méthode XP sans adaptation particulière. En outre, à l'instar de gros projets qui mettent en œuvre d'autres méthodes, il reste possible de diviser le projet en sous-projets, chacun pratiquant la méthode XP. Un coach expérimenté en XP n'aura même aucun mal à imaginer comment coordonner ces sous-projets en respectant l'esprit et les valeurs de l'Extreme Programming.

1. Voir le glossaire et l'aide-mémoire XP en annexe.

Enfin, étant donné la productivité d'une équipe XP de dix ou quinze personnes et la simplicité vers laquelle tend la méthode pour la résolution tant du problème que de la solution, rares sont les projets qui ne seraient pas à la portée d'une telle équipe. À moins que les délais ne soient tellement irréalistes que l'échec est presque assuré, quelle que soit la méthode employée.

Un des arguments avancés pour expliquer cette limitation théorique à dix personnes est que la réussite de la méthode dépend d'une bonne communication entre tous les membres de l'équipe tandis que les chemins de communication possibles se multiplient lorsque le nombre de personnes dans l'équipe augmente.

Cet argument serait valable si la communication au sein d'un projet XP consistait, chaque fois que l'on avait quelque chose à dire ou à demander, à aller voir l'un après l'autre tous les autres membres de l'équipe !

En fait, la communication repose sur plusieurs mécanismes, qui peuvent, tous, bien fonctionner au-delà de dix personnes :

- Toute l'équipe est rassemblée dans un même bureau, dans lequel on a installé une table de réunion. Tout le monde bénéficie plus ou moins directement de toute l'information qui circule, qu'elle soit informelle (échanges spontanés) ou formelle (réunions).
- Les programmeurs travaillent en binôme, en changeant régulièrement de binôme. Ils apprennent donc tous à se connaître, et diffusent ainsi leurs connaissances du logiciel.
- La responsabilité du code est collective, et les programmeurs sont amenés à intervenir sur toutes les fonctions et toutes les technologies mises en œuvre. Ils sont donc concernés par toutes les discussions, toutes les réunions. En outre, ils ont le droit et même le devoir de lire l'ensemble du code, et de le modifier en cas de besoin.

Le tout, dans les grandes équipes, est donc de veiller à ce que ces mécanismes soient préservés. Il faut par exemple éviter la séparation en plusieurs bureaux, s'assurer que les binômes tournent complètement et ne pas favoriser les spécialisations fonctionnelles ou techniques.

Monter en charge progressivement

Pour atteindre une taille d'équipe importante, il est préférable de monter en charge progressivement, en commençant par exemple avec quatre développeurs, puis en ajoutant un binôme toutes les deux ou trois itérations.

Cela permet de débuter le travail sur une plus petite partie du problème, avec une solution plus simple. Commencer de but en blanc avec dix ou douze programmeurs risque de faire dégénérer le travail en phase d'architecture sans valeur métier, et souvent avec la mise en place d'une infrastructure logicielle complexe et superflue. C'est d'autant plus vrai si les programmeurs n'ont pas d'expérience (et de convictions) préalables de la méthode XP.

Cette considération (l'apprentissage de la méthode, voir chapitre 7) est une autre raison de monter progressivement en charge. En effet, l'enseignement (et l'apprentissage) d'XP se fait par la pratique, aux côtés du coach ou de programmeurs qui ont une expérience préalable. L'apprentissage bénéficie ainsi d'une forte communication orale et d'un feedback immédiat. Après deux ou trois itérations, les programmeurs formés pourront eux-mêmes en former d'autres, mais au début il est difficile pour le coach de former plus de quatre programmeurs sans expérience de la méthode.

Enfin, et surtout, une montée en charge progressive permet d'éviter un surdimensionnement de l'équipe. Un projet qui démarre avec quinze personnes pourrait ne rien accomplir de plus qu'une équipe de huit personnes fonctionnant mieux. C'est particulièrement vrai avec la méthode XP, qui peut produire très rapidement des résultats impressionnants. Il faut donc commencer avec une équipe réduite, mesurer sa vélocité et affiner les estimations avant de décider s'il faut, ou non, monter en charge. Le cas échéant, il faut procéder itérativement : ajouter un binôme, remesurer la vélocité (voir chapitre 9) après une ou deux itérations et affiner encore les estimations…

Éviter la formation de sous-équipes et la spécialisation

Il existe une tendance naturelle, dans tout groupe de personnes, à former des sous-groupes basés sur une fréquentation antérieure, des affinités humaines ou techniques. Dans une équipe XP, cela n'a pas de conséquences tant que le groupe n'est pas trop important, car les rotations des binômes permettent de connaître tout le monde, et les réunions, même courtes et spontanées, impliquent toute l'équipe. Mais on hésite à interrompre 20 personnes pour une discussion de 10 minutes sur un point de conception. Si on laisse les binômes se former librement, certaines personnes peuvent être amenées à ne jamais travailler ensemble. Petit à petit, on court le risque de voir se former des sous-groupes de personnes qui se connaissent mieux, communiquent plus entre elles, et tendront inéluctablement à se spécialiser sur une technologie ou sur une fonctionnalité.

Il peut être utile de mettre en place un mécanisme permettant de s'assurer que les binômes tournent de manière systématique. Il sera toujours possible de faire une exception lorsque tel ou tel développeur a d'une façon spécifique besoin de «binômer» quelques heures avec une personne particulière. De même, afin d'éviter de mobiliser toute l'équipe autour de petites réunions de conception, il peut être intéressant de mettre en place un mécanisme de rotation, afin que ce ne soient pas toujours les mêmes qui soient sollicités pour ce genre de réunion, ou que des sous-équipes spécialisées ne se forment.

On notera que cette recommandation s'applique lorsqu'une spécialisation n'est pas nécessaire, et qu'elle est donc susceptible de nuire à la communication. Néanmoins, dans le cas de projets d'intégration conjuguant plusieurs technologies ou progiciels sur lesquels l'équipe ne peut assurer une totale polyvalence, on adoptera une démarche intermédiaire consistant à définir des équipes XP plus réduites, chacune compétente sur son domaine d'intervention.

L'équipe XP minimale

Peut-on faire de l'Extreme Programming tout seul ? Pas tout à fait. Pour mettre en œuvre l'ensemble de la méthode XP, il faut être au moins trois : un client, un programmeur et un coach/programmeur. Si l'on supprime le client (c'est-à-dire si le client n'existe pas sous la forme requise par la méthode, puisqu'un client existe forcément, même si c'est le programmeur lui-même…), une paire de programmeurs peut utiliser toutes les techniques de programmation (binôme, conception simple, tests unitaires, etc.), mais sans interaction avec le client et tout ce que cette dernière engendre (spécifications par feedback progressif, planification, etc.). Avec un client et un seul programmeur, on retrouve cette relation client-programmeur, mais le programmeur ne peut plus travailler en binôme. Quant à un programmeur seul, sans client au sens XP, il ne pourra guère que s'attacher à une conception simple, à programmer à l'aide de tests unitaires et à faire du refactoring. Ce n'est pas inutile ; c'est même une bonne façon de commencer à apprendre la méthode XP et à prendre de bonnes habitudes de programmation, mais il ne s'agit plus vraiment de méthode XP, proprement dite.

Comment s'y prendre ?

La définition des rôles peut servir de fil conducteur à la mise en place d'un projet XP. Les personnes affectées au projet sont-elles déjà connues ? Représentez-vous chaque personne dans un ou plusieurs rôles XP, discutez-en de préférence, jusqu'à ce que chacun se voie attribuer un rôle dans lequel il se sente à l'aise. Si tous les rôles sont attribués, vous avez une équipe prête à démarrer en XP ! Le plus souvent, vous découvrirez peut-être qu'il vous manque une personne, qui ait une certaine compétence ou une certaine personnalité. Vous en déduirez qu'il faut envisager une formation ou ajouter un membre à l'équipe.

Si vous devez constituer l'équipe en recrutant (que ce soit en interne ou en externe), utilisez les rôles XP pour guider vos recherches. Présentez la méthode et les rôles, et demandez au candidat celui qu'il souhaite jouer et pourquoi ? Quels sont ses atouts ? Pourrait-il jouer un autre rôle si son rôle préféré était déjà attribué ? Pensez également à l'importance de la programmation en binôme dans la pratique du recrutement (voir chapitre 5).

Une fois que l'équipe est formée, il faut veiller à ce que la répartition des rôles soit connue de tous. En effet, les rôles XP sont aussi un guide précieux pour l'apprentissage de la méthode. Personne ne peut tout absorber, tout comprendre d'un seul coup. Dans un premier temps, chacun pourra se concentrer sur l'apprentissage de son propre rôle. Quelles sont mes responsabilités ? Quelles sont les qualités humaines et techniques dont je dois faire preuve ? Quelles sont les pratiques XP qui me concernent le plus ? C'est le rôle attribué qui sert à chacun de guide pour répondre à toutes ces questions.

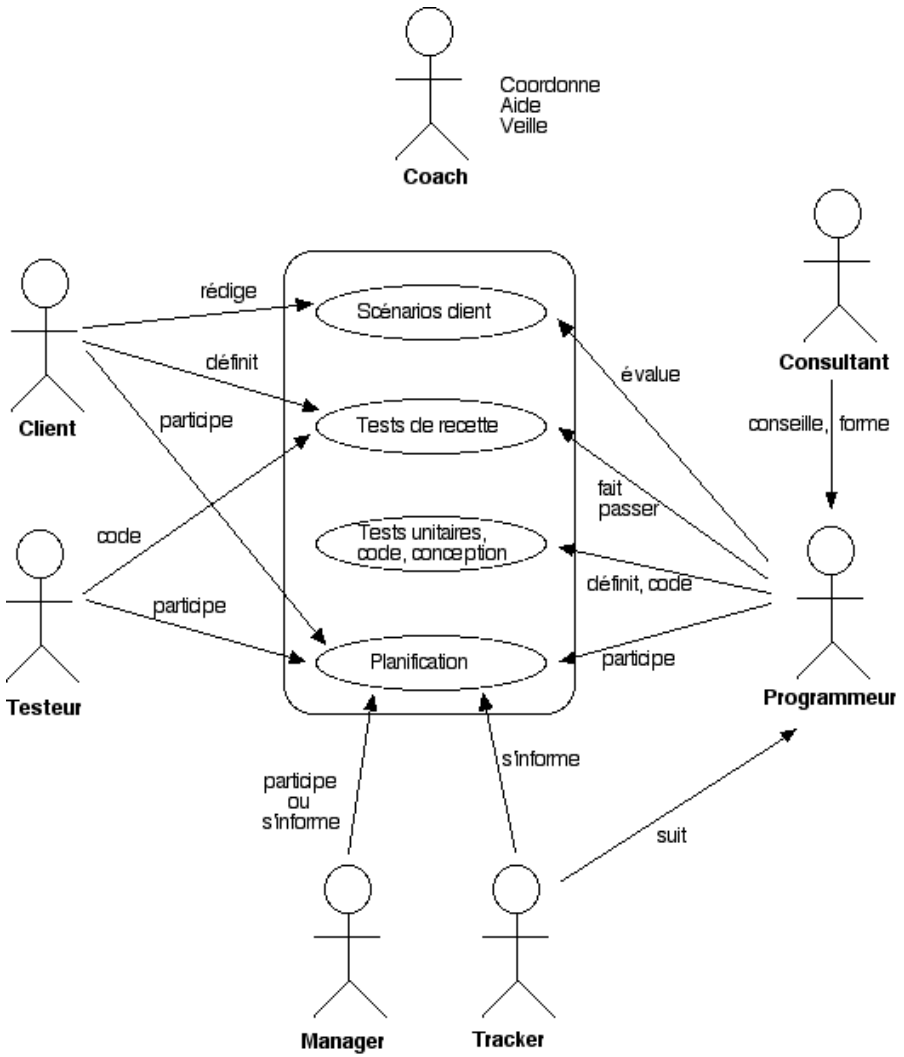

Figure 2.2 : Responsabilités respectives des rôles XP

3

Programmation

Il semble que la perfection soit atteinte, non quand il n'y a plus rien à ajouter, mais quand il n'y a plus rien à retrancher.

– Antoine de Saint Exupéry

Si les différentes pratiques qui forment l'armature d'XP abordent tous les aspects, ou presque, du déroulement d'un projet informatique, l'accent est toujours mis sur l'activité de programmation proprement dite, comme le laisse deviner le nom même d'Extreme Programming.

Nous allons donc aborder maintenant les pratiques relatives à la programmation, dans l'ordre logique des préoccupations qui s'imposent à un programmeur au cours de la réalisation d'un scénario client. Il convient d'insister sur le fait qu'il s'agit d'un ordre logique et non chronologique : la description qui suit n'est pas une liste d'étapes qu'il suffirait de suivre dans l'ordre (premièrement, écrire tous les tests du système ; deuxièmement, écrire le code proprement dit, et ainsi de suite).

Remarque

L'une des principales caractéristiques d'XP est de récuser une organisation des projets en phases bien découpées et réalisées séparément, d'une durée de plusieurs semaines ou plusieurs mois chacune, et dans laquelle un seul type d'activité est pratiqué : analyse des besoins, conception, programmation et enfin tests dans le schéma usuel.

Dans un projet XP, au contraire, ces activités apparaissent de façon homogène tout au long du projet. Du début à la fin, le travail du programmeur reste à peu près le même, tout comme les pratiques mises en œuvre. Par ailleurs, celles-ci se succèdent selon des cycles bien plus courts, de l'ordre d'une heure ou d'une journée plutôt qu'une semaine ou un mois.

Nous commencerons par une rapide synthèse de l'enchaînement des pratiques au sein d'une tâche de programmation dans XP. Nous donnerons ensuite une description plus approfondie de chacune de ces pratiques, avant d'aborder finalement un certain nombre de questions importantes sur leur interprétation ou leurs conséquences, telles que ce qu'il convient de faire du code existant ou sur le rôle joué par la documentation dans XP.

Survol des pratiques de programmation

Tout projet débute avec l'expression plus ou moins détaillée, plus ou moins précise, d'un besoin. Pour répondre à ce besoin, on s'apprête à produire un ensemble de fonctionnalités, chacune de ces fonctionnalités reposant sur un certain nombre de lignes de code. Les programmeurs comme le client chercheront à s'assurer que chacune de ces fonctionnalités et chacune de ces lignes de code contribuent bien à satisfaire les besoins et ne contiennent pas d'erreurs : on utilisera pour cela des *tests automatisés*, qui seront *écrits avant le code* correspondant.

Un projet, quel qu'il soit, s'inscrit dans une certaine durée. Dans cet intervalle de temps, tout peut survenir : certains détails du code qui a été écrit peuvent être oubliés, la définition de ce que le système doit faire peut évoluer... Il faut donc pouvoir facilement revenir sur ce qui a déjà été fait, pouvoir le comprendre ou le faire évoluer. C'est d'autant plus difficile que le système est compliqué. Il faut rechercher *la conception la plus simple possible*, et également l'exprimer de la façon la plus simple possible.

À mesure que le système s'étoffe, que de nouvelles fonctionnalités se greffent aux anciennes et que de nouvelles lignes de code s'ajoutent à celles déjà écrites, le système devient malgré tout plus complexe, plus difficile à comprendre et à faire évoluer – situation propice aux erreurs. Par chance, de bonnes idées peuvent aussi survenir suite à l'écriture du code d'une nouvelle fonctionnalité, permettant d'améliorer la façon dont une fonctionnalité plus ancienne est structurée. Pour maîtriser en permanence la qualité du code, on recourra à l'utilisation régulière d'une technique presque magique – le *remaniement* (*refactoring*).

Développement piloté par les tests

Un logiciel est écrit pour être utilisé. Ce qui nous amène nécessairement à la conclusion suivante : tout logiciel écrit est inévitablement testé, au sens large du terme. En effet, dans le pire des cas, c'est le client lui-même, après livraison du logiciel, qui est amené à constater les erreurs éventuelles du logiciel.

Bien entendu, chacun sait à quel point il est désagréable de faire l'expérience d'un tel scénario catastrophe. Généralement, les tests sont effectués bien avant – par le programmeur lui-même, ou par une tierce personne.

Remarque

Nous préciserons plus loin le sens particulier qu'XP donne aux termes « test » et « tester ».

À première vue, tester un logiciel n'a rien de bien sorcier. Dans le cas d'un traitement de texte, par exemple, il suffit de le lancer, de saisir une lettre ou quelques paragraphes d'un chapitre, puis d'utiliser les diverses commandes qui sont à notre disposition – mise en gras ou en italique, grossissement ou diminution des caractères, sauvegarde d'un document et lecture du même fichier. On observe ensuite si une commande a bien eu l'effet escompté : par exemple, la commande de mise en gras ne rendrait-elle pas, par hasard, l'italique ? L'insertion du 100e paragraphe – manipulation normale entre toutes – n'aurait-elle pas pour effet désastreux de redémarrer la machine ou de « planter » le programme ?

Le test d'un logiciel comprend donc en première analyse des tâches assez simples, répétitives et dont on peut juger du résultat (« ça marche » ou « ça ne marche pas ») sans mettre en jeu plus de jugeote que ce dont un ordinateur est capable. Bref, la plupart des tests peuvent être automatisés, même si dans certains cas la mise en œuvre automatisée des tests – en raison de l'infrastructure réseau qui doit être mise à disposition, ou de l'intégration de briques non homogènes – nécessite un peu de préparation.

Automatisation des tests

La présence de tests automatisés apporte deux avantages significatifs. En premier lieu, il en va des tests comme de toute tâche automatisée : ils peuvent être effectués plus rapidement et avec de moindres risques d'erreurs par l'ordinateur. Le temps consacré aux tests est par conséquent mieux employé – donc, plus rentable.

En second lieu, lorsque les tests automatisés sont effectifs pendant le développement d'un système, ils permettent de prévenir les *régressions* : une situation hélas assez courante où – parfois par négligence mais le plus souvent en raison d'effets de bord mal maîtrisés – une fonctionnalité qui « marchait » précédemment « ne marche plus ».

Définition

C'est la première caractéristique des tests dans XP : ils sont automatisés. C'est par ailleurs une obligation : on ne peut pas appeler « test » en XP une procédure de vérification qui ne serait pas automatisée. Toute fonctionnalité pour laquelle on dispose d'un ou plusieurs tests automatisés est dite « testée » ou « sous tests ».

Si les tests automatisés sont une telle panacée, on peut se demander pourquoi ils ne sont pas plus souvent utilisés. La raison invoquée le plus fréquemment est le manque de temps. L'écriture de tests automatisés pour un système que l'on a passé plusieurs mois à développer peut elle-même demander un temps considérable ; dans la plupart des cas, on a prévu de passer un

certain temps à tester le logiciel, mais bien moins qu'il n'en faudrait pour automatiser cette étape.

Notre XPérience

À elle seule, l'utilisation de tests de régression automatisés a des effets immédiats et importants sur la qualité du code. En comparant «l'avant» et «l'après» XP, nous estimons, sur des projets comparables, que le nombre de défauts signalés dans le code écrit a été divisé par cinq.

Écriture préalable des tests

XP est «extrême» dans la mesure où l'on choisit de donner la priorité à des choses très simples qui apportent des résultats probants, amenant ainsi à inverser la démarche habituelle : les tests seront écrits en premier – avant même le code qu'ils sont destinés à tester.

Parmi les avantages qu'il y a à attendre de cette approche à première vue paradoxale, l'un est évident : en se donnant comme règle de ne pas écrire de code sans avoir écrit des tests, on est du moins assuré de disposer, une fois le projet terminé, d'une série de tests automatisés. Les programmeurs ne souffriront plus du dilemme habituel : «Les tests manuels nous font perdre un temps considérable mais nous n'avons plus le temps de les automatiser ! »

Dans le même ordre d'idée, cette démarche conduit également à concevoir son code de façon qu'il soit facilement «testable» . Elle permet d'éviter une autre difficulté classique de l'écriture de tests automatisés en fin d'un projet : les parties du système que l'on souhaite pouvoir tester s'avèrent fréquemment trop enchevêtrées, dépendantes les unes des autres, pour que cela puisse se faire aisément.

Remarque

Cet enchevêtrement est en soi le signe d'une conception imparfaite. Lorsqu'un programmeur cherche à écrire du code qui puisse être facilement testé – ce qui est nécessairement le cas lorsqu'il écrit le test en premier lieu –, sa conception s'en trouve presque automatiquement améliorée.

L'écriture des tests avant le code est une des idées maîtresses dans XP ; elle conduit à modifier en profondeur certaines habitudes de programmation. Nous aurons également l'occasion de détailler ces conséquences ; pour l'instant, il est important de noter que les tests automatisés se déclinent, en général, en deux grandes catégories.

1. Les tests fonctionnels servent à exprimer de façon formelle le «quoi» – ce que le système doit faire. On peut même les considérer comme une forme de spécification.

2. Les tests unitaires servent à préciser le «comment», et remplissent notamment un rôle relatif à la conception.

Tests fonctionnels et tests de recette

Important

Cette section pourra être lue avec profit par les développeurs mais aussi par tout leur entourage *non* technique – notamment leurs clients, qui sont concernés au premier chef.

La *recette* est l'une des étapes cruciales de tout projet informatique : c'est le moment où, selon que le logiciel qui a été produit correspond ou non à ce qui était prévu, il est accepté, ou refusé. Dans le cas d'une prestation, cette étape détermine en partie la possibilité pour le prestataire de facturer son travail – ce qui suffit pour ne pas la prendre à la légère ; dans le cas d'une réalisation à usage interne ou d'un projet d'édition, elle constitue un «examen de passage» qui conduit à se prononcer sur le succès – ou l'échec – du projet.

Bien entendu, la recette dépend en premier lieu de l'absence d'anomalies constatées au moment des tests. Mais elle dépend également de l'adéquation des fonctions livrées – aussi parfaitement réalisées soient-elles – avec celles demandées par le client.

Le fait de procéder à des tests automatisés permet de se prémunir contre le premier problème : si ces tests sont pertinents, les anomalies éventuelles seront détectées bien avant que la recette n'ait lieu – dès lors que les programmeurs auront pris l'habitude d'exécuter les tests automatisés régulièrement au cours de leur travail.

La détection du second type de problème (inadéquation des fonctionnalités livrées) est bien plus délicate ! Comment peut-on, en effet, être certain que la fonctionnalité testée est bien celle que le client a demandée ? Une réponse simple s'impose : c'est le client lui-même qui a toute autorité pour définir la façon dont cette adéquation sera vérifiée. Or, si le client est en mesure de vérifier le logiciel livré, c'est qu'il est également en mesure de fournir aux programmeurs une description détaillée de cette vérification, susceptible d'être réalisée sous forme de tests automatisés.

Comme nous aurons l'occasion de le voir dans le chapitre 6, consacré à la gestion du projet, chaque fonctionnalité distincte du système est accompagnée d'un test qui permet, sur toute la durée du projet, de vérifier que la fonctionnalité en question a bien été implémentée comme on le demandait, et qu'elle reste opérationnelle une fois qu'elle est implémentée. Les descriptions des tests sont fournies aux développeurs le plus tôt possible au cours de la période de travail qui recouvre la fonctionnalité concernée : idéalement, avant que ne commence le travail d'implémentation ; dans le pire des cas, à mi-chemin de l'itération correspondante[1].

On résume ce mode de fonctionnement en disant que «le client écrit les tests de recette». Cette expression est une simplification qui peut en réalité recouvrir de nombreux cas pratiques

Définition

Dans un projet XP, on appelle ainsi *tests fonctionnels* la partie des tests qui a pour objet de préciser ce que sont les fonctionnalités du logiciel. On peut les appeler *tests de recette* dès lors qu'ils sont passés sous la responsabilité du client, et validés par ce dernier. Les deux expressions recouvrent exactement la même chose du point de vue technique : la distinction peut être utile lorsqu'on sort du cadre strict d'XP, par exemple lors de la « reprise » d'un code existant qui n'a pas été développé selon cette démarche. Les programmeurs peuvent alors écrire de leur propre initiative des tests fonctionnels pour garantir la stabilité du système, sans pour autant que ces tests soient au sens strict des tests de recette.

fort différents : les tests de recette peuvent être écrits par le client lui-même si l'équipe dispose d'une infrastructure qui le permet ; par exemple, si l'application manipule des données financières, les tests seront écrits sous formes de feuilles Excel, et directement exploités sous cette forme. Ou bien, les tests peuvent être rédigés par le client et automatiquement retranscrits sous une forme exploitable. Ou encore, les tests peuvent être écrits par des testeurs professionnels. Enfin, il est également possible de les faire écrire par les développeurs de l'équipe.

Du point de vue du client XP, ces tests ont une grande valeur car ils garantissent – dans la mesure où ils ont été correctement formulés – que la livraison correspond exactement à ce qu'il en attend.

Il en va de même du point de vue du programmeur XP : il sait que le logiciel peut être remis au client dès lors qu'il est conforme à tous les tests.

Remarque

Dans le cas de configurations un peu complexes (validations d'applications 3-tiers à base de client léger par exemple), où le client souhaite des tests « réels » (pour reprendre notre exemple, des tests effectués à partir de navigateurs Web), l'équipe de développement pourra s'appuyer sur des frameworks de test, souvent disponibles en logiciel libre, qui facilitent la mise en œuvre de tests automatisés dans de telles conditions.

Nous allons à présent aborder le sujet des tests unitaires, qui concerne plus directement les développeurs.

Tests unitaires

Tous les tests autres que les tests fonctionnels qui garantissent la qualité d'un projet sont écrits par les programmeurs, sous leur entière responsabilité. Appelés *tests unitaires*, ils sont

1. Dans tous les cas de figure, il convient de noter que les tests de recette peuvent, au cours de l'itération, faire l'objet d'un dialogue entre les développeurs et le client – c'est même une excellente manière de garantir leur pertinence.

également écrits en amont du code correspondant ; tout comme les tests de recette, ils permettent, entre autres, d'éviter la régression par rapport à des fonctionnalités déjà implémentées.

Les tests unitaires jouent plusieurs rôles de tout premier plan dans un projet XP : ils rythment le travail du programmeur, le guident dans la conception du système et servent de documentation du code qui a été produit.

Exemple d'utilisation

Prenons pour exemple une tâche de programmation simple, correspondant à une fonctionnalité banale pour un logiciel de type traitement de texte – le décompte du nombre total de mots dans un document.

Énoncé à la première personne

Le mode de travail habituel d'un programmeur peut se représenter sous la forme d'un monologue intérieur – «Je veux compter le nombre de mots dans le document en cours... Bien ! Je dois commencer par obtenir un pointeur sur le document... je dispose d'une fonction qui me donne un pointeur sur la première ligne ; celle-ci me permet de passer à la ligne suivante... Je fais tourner une boucle sur toutes les lignes du document en passant à la ligne suivante tant que je ne suis pas à la fin du document... et maintenant je dois calculer le nombre de mots dans une ligne... Il faut que je définisse ce que j'entends par un «mot»... Est-ce que je compte les chiffres... oh ! et il faudra aussi que je fasse le total...»

Même pour une tâche de programmation aussi simple que celle-ci, le programmeur doit tenir compte de difficultés ponctuelles – qui sont autant d'embûches potentielles, d'occasions de faire une erreur. L'utilisation de tests unitaires permet de structurer la pensée du programmeur et d'éviter qu'il ne dérape vers des problèmes secondaires sans que le problème d'origine ne soit vraiment, et correctement, résolu.

Le programmeur XP commence par identifier une partie isolée du problème ; par exemple, dans notre cas, «définir ce que l'on entend par un "mot"». Il écrit ensuite un test qui lui indiquera si cette partie du problème a été résolue : il peut s'agir par exemple d'un test qui fait appel au code chargé de séparer les mots, en donnant comme paramètres des chaînes de caractères simples, contenant un ou plusieurs mots : «UN MOT», «1 MOT», «UN,MOT»... Le test vérifie que dans chacun de ces cas le nombre renvoyé est le bon.

À ce stade, le code qui sépare «réellement» les mots n'est toujours pas écrit.

Cela n'a pas d'importance encore : le programmeur exécute le test. Comme la fonction n'existe pas, le test doit normalement échouer et indiquer que les résultats obtenus ne sont pas corrects. Il est «évident» que le test va échouer, mais en exécutant les tests alors qu'il sait cela pertinemment, le programmeur se donne une première garantie : si le test «passe» – s'il indique un résultat correct –, c'est qu'il n'est pas efficace pour détecter une anomalie. En voyant le test échouer, il s'assure qu'il est pertinent.

C'est uniquement après cette première vérification qu'il peut véritablement écrire le code chargé de séparer les mots. La présence du test rend toute erreur improbable : lorsque le code sera écrit, il pourra exécuter le test pour vérifier qu'il fonctionne correctement. Dès que le test passe, le programmeur sait que cette partie du problème est résolue et qu'il peut passer à la suite.

Dans notre exemple, il pourra s'assurer que le programme sait compter le nombre de mots dans une ligne. Cela peut être testé rapidement en suivant la même procédure : en appelant une fonction de comptage, qui n'existe pas encore, avec des lignes d'un document contenant un nombre variable de mots. Le test vérifie qu'on obtient les bons résultats. Pour écrire le code de façon que le test passe, il suffit maintenant de transformer une ligne du document en une chaîne de caractères – ce qui est probablement très simple. De même, pour vérifier qu'on sait compter les mots d'un document entier, il suffit maintenant de vérifier qu'on sait effectuer le décompte pour plusieurs lignes au lieu d'une seule.

Les tests conditionnent le rythme de programmation

L'écriture des tests unitaires constitue, sur le plan de la programmation, le «battement de cœur» d'un projet XP : elle rythme de manière fondamentale la progression du projet vers un système complet. L'écriture des tests avant le code permet au programmeur de progresser à la façon d'un alpiniste : on grimpe de quelques pas – on pose un piton pour se prémunir contre la chute – puis on reprend l'ascension.

On écrit donc un test, puis le code correspondant, et ainsi de suite – et, c'est important, on conserve chaque test à mesure qu'il est écrit. Les nouveaux tests se rajoutent aux anciens. L'accumulation des tests permet de progresser plus rapidement : pour écrire la fonction la plus «complexe», celle qui concerne le total des mots dans un document, le programmeur n'a pas à se poser beaucoup de questions puisque ses tests précédents lui indiquent qu'il dispose d'une fonction de séparation de mots et d'une fonction de comptage pour une ligne qui sont l'une et l'autre parfaitement correctes.

Les tests pilotent la conception

L'exemple précédent nous montre également la façon dont les tests permettent au programmeur d'aborder les problèmes de conception : quel problème aborder avant tel autre, comment faire en sorte que le code soit modulaire, et comment s'assurer qu'il pourra revenir sur une décision technique donnée sans avoir à tout récrire.

S'il souhaite modifier la définition d'un «mot» au sens du programme, par exemple pour en exclure les nombres, il lui suffira de modifier les résultats attendus par le test quant à la fonction de séparation des mots. Après avoir constaté que le test ainsi modifié échoue, il devra modifier le code correspondant ; lorsque le test passera de nouveau, la modification sera terminée. En revanche, le programmeur n'aura pas à revenir sur le code qui effectue le

décompte pour tout un document : ses tests lui garantissent qu'il reste valable quelle que soit sa définition d'un «mot».

Notre XPérience

En programmation objet, par exemple en Java, l'écriture d'un test unitaire avant le code revient donc à définir l'interface d'une classe à partir du point de vue de ses «clients», c'est à dire des classes qui vont l'utiliser. Concrètement, on définit les méthodes avant même de réfléchir aux attributs. La conception s'en trouve d'autant plus orientée objet…

Nous reviendrons plus en détail au chapitre suivant sur les mécanismes par lesquels les tests peuvent, non seulement encadrer le travail de réflexion sur la conception, mais même le rendre plus efficace.

Les tests documentent la conception

Il peut arriver à un programmeur d'intervenir sur du code qu'il n'a pas écrit – ce qui est nécessairement le cas de temps à autre : peu de programmeurs travaillent toujours seuls et toujours sur leur propre code. Même dans ce cas, il est probable qu'en revenant, longtemps après, sur du code qu'il a lui-même écrit, il ne se souvienne plus de tous les détails. Tous les programmeurs doivent, tôt ou tard, travailler en suivant cette logique de «maintenance». C'est l'une des principales raisons qui rendent indispensable l'existence de documentations détaillées sur le code produit dans le cadre d'un projet.

L'écriture des tests conduit à décrire de façon formelle ce que fait chaque partie du code. En effet, un test est une façon d'exprimer des contraintes telles que «si cette fonction est appelée avec une chaîne contenant des nombres, ceux-ci ne seront pas comptés comme des mots». Plus formellement, on peut rapprocher ce rôle des tests unitaires des notions de «contrat» et de «conception par contrat», prônées notamment par Bertrand Meyer.

En ce sens, les tests unitaires produits par XP correspondent à une documentation détaillée des contrats remplis par le code.

Conception simple

Nous avons déjà évoqué, plus haut, la notion de «conception». Le mot lui-même, traduction de l'anglais «design», recouvre – tout comme «architecture», son cousin – un grand nombre de significations, sujettes à de nombreux débats.

Malheureusement – ou peut-être heureusement –, ce livre n'a pas pour vocation de fournir une réponse définitive à la question de savoir ce qu'est la conception... Quelques notions intuitives suffiront à notre discussion. Nous appellerons «conception» toute réflexion qui touche à la structure d'un système plutôt qu'à sa substance – sans chercher à définir rigoureusement ce que signifient «structure» et «substance». Nous nous appuierons, tout en la récusant, sur

l'idée qu'on peut séparer l'activité du programmeur en deux composantes, l'une nommée «conception» qui serait indépendante de l'écriture de code dans un langage particulier, l'autre «implémentation» qui consiste à exprimer dans le code les idées provenant de la conception.

L'un des objectifs d'XP, semble-t-il, est précisément d'évacuer la problématique de la conception, et avec elle la recherche de critères «mathématiques», ou «objectifs», pour juger une bonne conception, ou tout argument visant à imposer telle ou telle méthode, tel ou tel outil, telle ou telle notation, comme «la» solution permettant de concevoir des logiciels. En lieu et place, XP propose des règles et des activités concrètes ; on jugera de l'efficacité de ces règles ou activités sur les résultats.

La conception comme investissement

L'équipe chargée du projet dispose d'une idée générale de la structure du système – sous la forme d'une métaphore, que nous présenterons plus en détail au chapitre 5 ; on lui a par ailleurs fourni des tests fonctionnels qui précisent ce qui doit être implémenté.

Pour obtenir le meilleur système possible, il peut sembler nécessaire d'identifier toutes les classes ou tous les modules dont il sera constitué, examiner la façon dont chacun contribue aux fonctionnalités prévues, et s'assurer que tous les cas de figures envisageables sont bien pris en compte. On pourrait ainsi dégager les classes principales, les relations d'héritage..., en somme, tous les éléments qui permettraient à l'équipe d'anticiper des problèmes éventuels qui pourraient se poser par la suite, et donc de gagner du temps.

On pourrait aussi être tenté de considérer chacune des fonctionnalités à implémenter comme un cas particulier d'un problème plus général ; en commençant par implémenter un système générique capable de traiter n'importe lequel de ces cas particuliers, il suffirait ensuite de très peu de temps pour implémenter les fonctionnalités spécifiques. Et ce gain de temps serait démultiplié si, par la suite, de nouvelles fonctionnalités étaient demandées, proches de celles déjà traitées.

Ces deux démarches ont un point commun : elles représentent un investissement. Dans le premier cas, il s'agit d'un investissement en temps – le temps consacré à la conception avant le démarrage du projet. Dans le second, il s'agit également d'un investissement en temps – doublé d'un surplus de complexité : les programmeurs vont écrire plus de code que ce dont ils ont strictement besoin.

Pas d'investissements spéculatifs

Cet investissement peut avoir des effets bénéfiques : une conception initiale qui a correctement anticipé les changements, les évolutions de toutes sortes, ne demandera pas de travail supplémentaire par la suite. Mais il en va de la conception comme de la Bourse : si on a mal spéculé, on risque de perdre beaucoup. Il est parfaitement possible qu'au cours du projet, les

besoins et/ou le contexte technique évoluent ; si la conception se révèle sensible à de tels changements, il faudra reprendre une partie du travail... et le projet aura déjà pris du retard.

Malheureusement, l'une des rares certitudes en matière de projets informatiques, c'est que les besoins changent fréquemment – ou ne sont pas complètement exprimés au début du projet.

À la spéculation, XP préfère le rendement. Une équipe XP sait en permanence (nous verrons comment au chapitre 6) quelles fonctionnalités sont les plus importantes pour son client – et par conséquent lesquelles représentent le meilleur investissement de son temps, puisque la satisfaction du client représente sa «plus-value».

Notre XPérience

Il arrive très souvent aux programmeurs de «blinder» leur code ou leur conception, de tenir compte d'une foule de cas dont le client n'a jamais exigé qu'ils soient pris en compte; de mettre en place des fonctions qui n'étaient pas prévues, par exemple la gestion d'un fichier de configuration de l'application au format XML. Résister à cette tentation s'avère étrangement difficile, et demande peut-être plus de discipline que nombre des pratiques d'XP.

Les programmeurs vont donc travailler sur une et une seule fonctionnalité, et n'en implémenteront que le strict nécessaire. Mais elle sera implémentée complètement et correctement – les tests fonctionnels fournis (ou validés) par le client, ainsi que les tests unitaires écrits par les programmeurs eux-mêmes, permettent de le garantir. S'il est nécessaire de réfléchir à la meilleure conception pour cette fonctionnalité, l'équipe le fera – mais sans aller au-delà.

Les programmeurs XP aiment à résumer ces limites imposées au travail de conception par deux proverbes un rien provocateurs : «Do The Simplest Thing That Could Possibly Work» – choisir la solution la plus simple qui puisse résoudre le problème – et «You Aren't Gonna Need It[1]» – on n'en aura pas besoin (sous-entendu : du composant générique, ou du code très «futé» qui permet «presque pour rien» d'obtenir plus de fonctionnalités).

Simplicité ou facilité

Une distinction très importante doit être faite à ce stade – «le plus simple» ne veut pas toujours dire «le plus facile». En fait, il est souvent plus difficile de faire simple que de faire compliqué.

Supposons qu'après avoir écrit une méthode qui traite d'un cas particulier (par exemple, effectuer une opération sur certains des fichiers contenus dans un répertoire, et récursivement leurs sous-répertoires) le programmeur ait besoin d'une méthode similaire, mais qui opère dans un autre répertoire, et se base sur des critères différents pour la sélection des fichiers.

1. Il s'agit bien de l'orthographe consacrée – on ne l'écrit pas «Ain't». Le plus souvent d'ailleurs, ce sont les acronymes qui sont utilisés : DTSTTCPW et YAGNI. Seul le second peut raisonnablement être utilisé à l'oral.

La solution la plus facile qui puisse fonctionner consiste à copier le texte de la méthode et à le coller dans une autre. Et de recommencer, lorsqu'on devra à nouveau recourir à un parcours sélectif de répertoire.

Et si cette méthode n'est pas correcte ? (L'utilisation de tests automatisés n'évite pas forcément toutes les anomalies.) Le programmeur – ou pire, ses coéquipiers – devra rechercher toutes les variantes de la méthode et les corriger, une par une. À la perte de temps que représente ce travail répétitif s'ajoute la multiplication des risques d'erreur lors de la correction. Et si, par exemple pour optimiser les accès disques, il devient nécessaire de modifier l'algorithme de parcours ? Même motif, même punition...

Il peut sembler difficile, dans des cas pratiques qui seront sans aucun doute plus délicats que cet exemple, de faire la part des choses entre «simple» et «facile». On peut aussi nourrir quelque inquiétude : à se demander à chaque fois si la solution qu'on a choisie est la plus simple, ne va-t-on pas finir par être paralysé, incapable d'avancer ?

Remarque

Une bonne façon de déceler une solution facile, mais pas simple, est d'écouter parler le programmeur concerné. Une phrase commençant par «C'est facile…» est un bon indice. On sera également attentif à «Il n'y a qu'à…», plus subtil mais généralement tout aussi révélateur. La recherche d'une solution simple commence généralement par une question : «Quelles sont les différentes solutions possibles ?»

L'essentiel est de progresser – l'équipe a un projet à terminer, après tout ! Nous verrons plus loin qu'un outil remarquablement efficace est à notre disposition pour simplifier ce qui – par paresse ou par nécessité – est devenu complexe. Nous verrons aussi que la notion de simplicité peut s'exposer en quelques mots – de façon toute simple.

Une autre subtilité tient à ce que la règle veut qu'on implémente «la solution la plus simple qui puisse fonctionner» – mais pas «la plus simple», tout court. Si, pour un problème complexe, on a absolument besoin d'une solution complexe – tant pis pour la simplicité ; il faut disposer d'une solution.

Le travail de développement

Des différents points que nous avons abordés jusqu'à présent commence à émerger une vision presque complète du travail de développement tel qu'il est conçu dans XP :

- On ajoute un test au système.
- Le test échoue : il faut écrire la fonctionnalité correspondante.
- On écrit uniquement le code le plus simple permettant de faire passer le test.
- On vérifie que le test passe.
- On simplifie en appliquant les techniques dites de «remaniement».

Cette démarche est valable aussi bien pour les tests fonctionnels que pour les tests unitaires : lorsqu'un test fonctionnel est rajouté au système, il faudra peut-être, selon les cas, écrire plusieurs classes, plusieurs méthodes, ou plusieurs fonctions – et donc plusieurs tests unitaires – mais, dans tous les cas, le programmeur XP s'attachera à n'écrire que ce qui est strictement nécessaire pour faire passer chaque test, et toujours de la façon la plus simple possible.

Cette façon de procéder produit un effet presque «accidentel» : à n'écrire que le code strictement nécessaire pour satisfaire chaque test, on obtiendra au fil du temps une «couverture» totale des tests sur le code produit : c'est un terme technique («coverage» en anglais) qui indique la portion du code qui sera exécutée lorsque les tests sont exécutés, et qui bénéficie donc de la protection qu'apportent ces tests contre la régression ou l'introduction d'anomalies.

L'expérience est-elle utile ?

Une objection peut être légitimement soulevée : comment dans ces conditions tirer profit de l'expérience passée ? Sur des projets précédents, tel ou tel programmeur dans l'équipe aura déjà résolu des problèmes similaires, implémenté des fonctionnalités du même type. Il peut légitimement avoir un avis sur la structure qui permet d'obtenir une bonne solution ; sur une conception dont il sait déjà qu'elle est suffisamment simple, mais efficace. Il peut lui sembler impossible, ou simplement fastidieux, de la mettre en place en écrivant un test, puis un peu de code, puis un test… Est-on vraiment obligé de «redécouvrir» chaque fois ce qu'on sait déjà ?

L'expérience est toujours utile. Plus on en sait sur un problème donné, plus on est à même de le résoudre correctement. À cette occasion, il faut d'ailleurs mentionner que si, dans XP, les seuls outils considérés comme indispensables pour obtenir une conception robuste sont les pratiques de base – des techniques comme la notation UML ou les «Design Patterns» (modèles de conception), notamment, n'en font pas partie –, tout ce qui peut aider à mieux comprendre les questions techniques qui se posent au cours d'un projet a son utilité.

Si on est vraiment certain de la pertinence d'une solution, il sera généralement possible de l'implémenter par petites étapes ; à aucun moment, en écrivant ce qui est nécessaire pour satisfaire les tests déjà écrits, on ne devrait perdre de vue ce qu'on pense être l'objectif à atteindre en termes de conception. Si cela se produit malgré tout, c'est peut-être que la solution en question n'était pas si bien adaptée – en attendant, on aura toujours obtenu un système fonctionnel et sans anomalies.

Les règles de simplicité

Il ne suffit pas de partir d'une solution simple. Il faut aussi, nous l'avons vu, éviter de la rendre complexe. Toutefois, tout système finira par accumuler des lourdeurs, des redondances – ce qui peut faire obstacle à sa maintenance, mais aussi à son évolution au cours du projet.

Bien des solutions de facilité, qui peuvent apparaître «simples», se révèlent néfastes pour la qualité de ce qu'on produit. Entre toutes, la duplication du code existant est considérée, en XP, comme un péché cardinal ; la duplication sous toutes ses formes est le premier signe d'un code source dont la qualité se dégrade. L'un des commandements les plus importants est donc : «Once And Only Once», autrement dit n'écrire chaque chose qu'une et une seule fois.

Pour que notre code reste gérable tout au long du projet, et par la suite, l'autre facteur important est sa limpidité. L'information présente dans le code doit permettre d'en comprendre les moindres détails. Lorsqu'un programmeur écrit du code, il exprime et articule des idées – celles que nous avons décrites plus haut sous la forme d'un «monologue intérieur» du programmeur (et qui, en XP, peut d'ailleurs se transformer en dialogue – comme nous le verrons au chapitre 5). Le code doit identifier et exprimer clairement chacune de ces idées, de façon que tout autre programmeur qui le relise puisse comprendre ce qui s'y passe.

Les règles suivantes résument les notions essentielles de qualité du code :

- Tous les tests doivent être exécutés avec succès.
- Chaque idée doit être exprimée clairement et isolément.
- Tout doit être écrit une fois et une seule.
- Le nombre de classes, de méthodes et de lignes de code doit être minimal.

L'étape de «remaniement» – appliquée régulièrement, par exemple après avoir écrit du code qui a permis de satisfaire un nouveau test – permet de conserver en permanence un code qui répond à ces critères.

Notre XPérience

Ces règles de qualité du code sont très simples – presque triviales. Nous nous sommes aperçus en les mettant en pratique qu'elles donnent tout leur sens à des notions de conception objet qui nous semblaient jusque-là plutôt «académiques», telles que le partage des responsabilités entre les objets, l'intérêt d'utiliser la délégation plutôt que l'héritage, la distinction entre des catégories d'objets telles que objets valeurs ou objet références ; ou à des outils de conception comme UML ou les Design Patterns. Considérées sous l'angle XP, ces différentes notions sont autant de guides qui permettent d'orienter l'évolution du code vers une forme optimale, où les différentes règles de simplicité sont observées le plus complètement possible.

Remaniement (*refactoring*)

Martin Fowler, qui fait référence en la matière, décrit le remaniement comme un procédé consistant à modifier un logiciel de telle façon que, sans altérer son comportement vu de l'extérieur, on en ait amélioré la structure interne.

Parler de structure interne revient à parler de conception. À elle seule, la notion XP de conception simple n'est qu'une profession de foi : «Je peux systématiquement choisir la solution la plus simple parce que, si celle-ci se révèle plus tard insuffisante, j'aurai toujours les moyens de la faire évoluer vers une solution plus générale ou plus complexe.»

Le livre de Martin Fowler – ou le site Web associé – apporte sans conteste la preuve que ces moyens existent et qu'ils sont efficaces. Il répertorie plusieurs dizaines de transformations de code – pour la plupart accompagnées de diagrammes UML ou d'exemples de code «avant» et «après» – qui, appliquées séquentiellement, permettent de modifier la structure d'un programme sans altérer son fonctionnement.

Chacune de ces transformations est décrite à un niveau de détail suffisant pour laisser très peu de place à l'erreur dans leur application. La présentation adoptée emprunte quelque peu au format des Design Patterns : chaque mécanisme est présenté succinctement, puis on en expose l'intérêt – dans la plupart des cas, il s'agit de rendre plus maniable une partie du code que sa structure rend difficile à faire évoluer, pour des raisons variables. Vient ensuite le détail des étapes à suivre pour effectuer la transformation.

La description de ces étapes est très précise, à tel point que l'application de nombreux remaniements est purement mécanique. Certains environnements de développement commencent d'ailleurs à proposer des fonctions de remaniement automatique, capables d'assurer des transformations de manière autonome en s'appuyant sur une analyse très fine du code à modifier. Ces outils déchargent ainsi les développeurs des parties fastidieuses de ces transformations, ce qui rend les manipulations du code à la fois plus rapides et plus sûres.

Programmation darwinienne

L'utilisation de tests unitaires et le remaniement vont main dans la main. Sans tests unitaires, il ne serait pas possible d'être certain, après un remaniement donné, ou toute une série, que nous n'avons pas introduit d'erreurs. Dans ce cas, se hasarder à modifier – peut-être de façon significative – la structure du code serait périlleux.

En présence de tests unitaires suffisamment complets (qu'ils soient ou non écrits avant le code), on peut au contraire adopter une attitude plus flexible par rapport à ce qui est déjà écrit. Lorsqu'un programmeur relit le code source d'une classe et se dit, «ça serait plus pratique si ce truc-là héritait de ça», il a le loisir de pratiquer la transformation et d'observer, concrètement, quel en est le résultat. S'il semble meilleur que ne l'était l'ancienne version, il peut choisir de conserver ses modifications – à condition que tous les tests passent. Si les tests détectent une erreur – ou si la modification ne donne pas satisfaction –, il suffira de revenir à la version précédente.

Chaque modification effectuée au cours d'un remaniement est minime, et peut facilement être annulée. Un programmeur XP ne s'engage donc jamais dans un travail de longue haleine qui représenterait une perte de temps significative si jamais il s'avérait injustifié. Une modification qui ne serait pas judicieuse ne demande que quelques minutes à tester et éventuellement

annuler. Il est rare dans ce contexte de chercher longtemps l'origine d'un défaut introduit par accident : si cela marchait il y a cinq minutes, le programmeur sait exactement ce qu'il a fait depuis, et ses modifications sont en général très localisées.

On peut comparer ce style de programmation avec l'évolution biologique[1] – chaque remaniement est une mutation isolée, les tests unitaires et fonctionnels jouent le même rôle que la sélection naturelle : toute mutation nocive est éliminée ; on ne conserve que ce qui améliore réellement le code. De la même façon que, sur de longues périodes, la sélection naturelle est capable de produire des résultats étonnants par la seule accumulation de mutations minimes mais individuellement avantageuses, l'application répétée des étapes de remaniement, rigoureusement contrôlée par l'exécution des tests, peut produire des programmes des plus robustes.

Élimination du code dupliqué

L'un des principaux obstacles à la flexibilité du code est la présence de fragments de code identiques – ou très similaires – à plusieurs endroits. Ces doublons sont autant de freins potentiels à l'évolution du code pour de nouveaux besoins. Imaginons, par exemple, qu'à plusieurs endroits dans une application, on trouve du code qui ouvre un fichier, puis lit des données dans un format donné. Le client souhaite ouvrir l'application au Web – ce qui signifie que les données doivent être lues, non pas depuis un fichier, mais depuis une adresse URL sur un serveur distant.

Si le code dans son état actuel respecte la règle «une fois et une seule», les programmeurs ne devront modifier qu'une méthode pour modifier la façon dont l'application accède aux données ; et, pour un effort supplémentaire, minime, ils seront en mesure d'adapter leur programme pour qu'il puisse adopter l'une ou l'autre façon selon les cas.

Examinons une version simplifiée, mais représentative, de cet exemple[2] :

```
public void lireConfigurations() throws IOException {

   String ligneCourante;

   File configUtilisateur = new File("donnees/interface.cfg");
   Vector lignesUtilisateur = new Vector();
   BufferedReader lectureUtilisateur = new BufferedReader(new
➥FileReader(configUtilisateur));
   do {
      ligneCourante = lectureUtilisateur.readLine();
      if (ligneCourante != null) {
```

1. On parle également à ce sujet de «conception émergente».
2. Ce fragment de code a été compilé et exécuté sous Java 1.3. Un test unitaire très rudimentaire a été également écrit pour en vérifier la non-régression au cours des étapes de remaniement décrites ici. Un listing complet est donné en annexe.

```
            lignesUtilisateur.add(ligneCourante);
        }
    }
    while (ligneCourante != null);
    lectureUtilisateur.close();

    File configSysteme = new File("donnees/systeme.cfg");
    Vector lignesSysteme = new Vector();
    BufferedReader lectureSysteme = new BufferedReader(new FileReader(configSysteme));
    do {
       ligneCourante = lectureSysteme.readLine();
       if (ligneCourante != null) {
          lignesSysteme.add(ligneCourante);
       }
    }
    while (ligneCourante != null);
    lectureSysteme.close();

    this.configUtilisateur = lignesUtilisateur;
    this.configSysteme = lignesSysteme;

    }
```

Même sans connaître Java, la signification de ce code ne devrait pas poser de problème : il s'agit d'ouvrir un fichier, de lire son contenu ligne par ligne, en déposant chaque ligne dans une liste (`Vector`). Cette opération est répétée deux fois de suite.

La présence de code dupliqué est évidente dans l'exemple exposé ci-avant. Dans un cas aussi trivial, on peut être tenté de simplement récrire le code fautif de façon plus générique. Dans la pratique, il ne sera pas toujours évident de préserver le comportement actuel du code lors de ce genre de récriture. En revanche, nous allons voir que l'application méthodique de deux remaniements suffit à obtenir un résultat équivalent, quasiment sans effort.

Dans un premier temps, il convient d'appliquer «Extraire une méthode», deux fois de suite, pour isoler les parties dupliquées. Il s'agit de créer une nouvelle méthode, avec un nom approprié, dans laquelle on placera un fragment de code «atomique». (Le mécanisme de ce remaniement très courant peut être plus ou moins facile à appliquer, en particulier en présence de variables locales à la méthode. Pour tous les détails, consultez le livre de Martin Fowler.)

```
public void lireConfigurations() throws IOException {

   Vector lignesUtilisateur = lireConfigurationUtilisateur();
   Vector lignesSysteme = lireConfigurationSysteme();

   this.configUtilisateur = lignesUtilisateur;
   this.configSysteme = lignesSysteme;
```

```
}

public Vector lireConfigurationUtilisateur() throws IOException {
   File configUtilisateur = new File("donnees/interface.cfg");
   Vector lignesUtilisateur = new Vector();
   BufferedReader lecture = new BufferedReader(new FileReader(configUtilisateur));
   String ligneCourante;
   do {
      ligneCourante = lecture.readLine();
      if (ligneCourante != null) {
      lignesUtilisateur.add(ligneCourante);
      }
   }
   while (ligneCourante != null);
   lecture.close();
   return lignesUtilisateur;
}

public Vector lireConfigurationSysteme() throws IOException {
   File configSysteme = new File("donnees/systeme.cfg");
   Vector lignesSysteme = new Vector();
   BufferedReader lecture = new BufferedReader(new FileReader(configSysteme));
   String ligneCourante;
   do {
      ligneCourante = lecture.readLine();
      if (ligneCourante != null) {
      lignesSysteme.add(ligneCourante);
      }
   }
   while (ligneCourante != null);
   lecture.close();
   return lignesSysteme;
}
```

À ce stade, il est recommandé de recompiler cette portion de code et de repasser les tests unitaires correspondants. Ceux-ci devraient d'ailleurs continuer à passer sans problème : en effet, les remaniements changent uniquement la structure du code ; les services rendus, pour leur part, restent inchangés.

Dans un second temps, le remaniement « Paramétrer une méthode » permet de rendre l'une de ces deux méthodes plus génériques, en introduisant un paramètre. Il est ensuite possible de renommer la méthode et certaines de ses variables pour mieux correspondre à la nouvelle structure :

```
public void lireConfigurations() throws IOException {

   Vector lignesUtilisateur = lireConfiguration("donnees/interface.cfg");
```

```
   Vector lignesSysteme = lireConfigurationSysteme();

   this.configUtilisateur = lignesUtilisateur;
   this.configSysteme = lignesSysteme;
}

public Vector lireConfiguration(String nom) throws IOException {
   File fichier = new File(nom);
   Vector lignes = new Vector();
   BufferedReader lecture = new BufferedReader(new FileReader(fichier));
   String ligneCourante;
   do {
      ligneCourante = lecture.readLine();
      if (ligneCourante != null) {
      lignes.add(ligneCourante);
      }
   }
   while (ligneCourante != null);
   lecture.close();
   return lignes;
}

public Vector lireConfigurationSysteme() throws IOException {
   File configSysteme = new File("donnees/systeme.cfg");
   Vector lignesSysteme = new Vector();
   BufferedReader lecture = new BufferedReader(new FileReader(configSysteme));
   String ligneCourante;
   do {
      ligneCourante = lecture.readLine();
      if (ligneCourante != null) {
      lignesSysteme.add(ligneCourante);
      }
   }
   while (ligneCourante != null);
   lecture.close();
   return lignesSysteme;
}
```

Désormais, il est possible de remplacer la méthode spécifique par un appel à la méthode plus générique (si nous avions plus de deux cas de duplication, nous aurions à répéter cette étape autant de fois que de méthodes extraites).

```
public void lireConfigurations() throws IOException {

   Vector lignesUtilisateur = lireConfiguration("donnees/interface.cfg");
   Vector lignesSysteme = lireConfiguration("donnees/systeme.cfg");
```

```
   this.configUtilisateur = lignesUtilisateur;
   this.configSysteme = lignesSysteme;
}

public Vector lireConfiguration(String nom) throws IOException {
   File fichier = new File(nom);
   Vector lignes = new Vector();
   BufferedReader lecture = new BufferedReader(new FileReader(fichier));
   String ligneCourante;
   do {
      ligneCourante = lecture.readLine();
      if (ligneCourante != null) {
      lignes.add(ligneCourante);
      }
   }
   while (ligneCourante != null);
   lecture.close();
   return lignes;
}
```

Pour terminer, il est souhaitable de simplifier la méthode d'origine en appliquant «Supprimer une variable temporaire», deux fois :

```
public void lireConfigurations() throws IOException {

   this.configUtilisateur = lireConfiguration("donnees/interface.cfg");
   this.configSysteme = lireConfiguration("donnees/systeme.cfg");
}

public Vector lireConfiguration(String nom) throws IOException {
   File fichier = new File(nom);
   Vector lignes = new Vector();
   BufferedReader lecture = new BufferedReader(new FileReader(fichier));
   String ligneCourante;
   do {
      ligneCourante = lecture.readLine();
      if (ligneCourante != null) {
         lignes.add(ligneCourante);
         }
   }
   while (ligneCourante != null);
   lecture.close();
   return lignes;
}
```

Ce code est plus compact, plus lisible, et surtout plus flexible que le code d'origine – puisqu'il suffit maintenant de modifier ou de surcharger la méthode `lireConfiguration` pour être capable de traiter des adresses Web aussi bien que des fichiers.

Il faut noter par ailleurs qu'un tel remaniement constitue un progrès objectif vers un code qui répond aux règles de simplicité telles que nous les avons abordées précédemment. Il ne suffit pas – loin s'en faut – d'éliminer les redondances pour obtenir une conception parfaitement malléable, mais c'est l'une des techniques les plus utiles.

Séparation des idées

Les fonctions ou les méthodes à rallonge sont un autre obstacle majeur à la production d'un code de qualité, souple et évolutif, pour deux raisons. En premier lieu, de longues méthodes renferment une quantité importante de fonctionnalités ; elles font beaucoup de choses différentes, mais tout cela n'est disponible que dans le contexte de cette méthode. D'autres méthodes ou d'autres classes qui pourraient bénéficier de ces fonctionnalités en seront bêtement privées.

En second lieu, et plus important encore, elles peuvent constituer une entrave à la compréhension du code. Or, il faut se rappeler que tout code nécessitera de la maintenance, ou au mieux de la relecture. Tout doit être fait pour faciliter la communication avec les programmeurs qui hériteront de notre code.

Nous venons de voir que le fait d'identifier deux éléments distincts dans une même méthode et de les extraire à un autre endroit permettait d'isoler du code dupliqué pour obtenir un seul fragment de code plus générique. La même démarche permet également de favoriser la compréhension du code. En voici un exemple simple ; après quelques remaniements, nous avons obtenu :

```
   public void lireConfigurations() throws IOException {
      this.configUtilisateur = lireConfiguration("interface");
      this.configSysteme = lireConfiguration("systeme");
   }

   public String repertoireBase() {
      return "C:\Projets\Java\XP\Config";
   }

public Vector lireConfiguration(String zone) throws IOException {
   // Calcule le nom de fichier approprié
   String nom = repertoireBase();
   if (!repertoireBase().endsWith("/")) nom = nom + "/";
   nom = nom + "donnees/"+zone+".cfg";
```

```
   // Lit le fichier de configuration
   File fichier = new File(nom);
   Vector lignes = new Vector();
   BufferedReader lecture = new BufferedReader(new FileReader(fichier));
   String ligneCourante;
   do {
      ligneCourante = lecture.readLine();
      if (ligneCourante != null) {
      lignes.add(ligneCourante);
      }
   }
   while (ligneCourante != null);
   lecture.close();
   return lignes;
}
```

Dans le code exposé ci-avant, une même méthode effectue deux traitements bien distincts : d'une part, un calcul pour obtenir le nom d'un fichier à partir d'un nom de base et d'une règle de correspondance bien définie ; d'autre part, la lecture à proprement parler. La présence de plusieurs idées se manifeste par l'apparition de commentaires en ligne, et la séparation de la méthode en deux blocs distincts par des lignes vides.

La recherche d'un code plus expressif conduit à transformer ce code comme suit, par deux applications successives du remaniement «Extraire une méthode» :

```
public Vector lireConfiguration(String zone) throws IOException {
   String nom = nomFichierConfiguration(zone);
   return lireFichierConfiguration(nom);
}

private String nomFichierConfiguration(String zone) {
   String nom = repertoireBase();
   if (!repertoireBase().endsWith("/")) nom = nom + "/";
   nom = nom + "donnees/"+zone+".cfg";
   return nom;
}

private Vector lireFichierConfiguration(String nomFichier) throws IOException {
   File fichier = new File(nomFichier);
   Vector lignes = new Vector();
   BufferedReader lecture = new BufferedReader(new FileReader(fichier));
   String ligneCourante;
   do {
      ligneCourante = lecture.readLine();
```

```
        if (ligneCourante != null) {
        lignes.add(ligneCourante);
        }
    }
    while (ligneCourante != null);
    lecture.close();
    return lignes;
}
```

Vous noterez qu'au passage, on a éliminé les commentaires qui ne présentent plus d'intérêt, puisque les noms de méthodes que nous avons choisis sont équivalents pour ce qui est de l'information fournie au programmeur qui relira ce code. La présence de commentaires dans le code est d'ailleurs dans bien des cas l'indication de la nécessité d'un remaniement de ce type. Très fréquemment, il est possible de découvrir la meilleure façon d'exprimer une idée dans le code en se posant la question suivante : « Comment puis-je le récrire pour éliminer le commentaire ? »

Remarque

L'utilisation de commentaires au sein du code, notamment pour la description d'algorithmes complexes « par nécessité », reste bien entendu justifiée si ces commentaires se révèlent indispensables pour comprendre tel ou tel fonctionnement. La question que nous venons d'énoncer sert toujours de guide : une fois le code écrit, il faut se demander : « Est-ce totalement compréhensible ? » ; si la réponse est « non », on cherchera à clarifier le code lui-même ; si cet effort est infructueux, un commentaire s'impose. On n'hésitera pas, en présence d'un code difficile à suivre, tout d'abord à introduire des commentaires, puis à tenter de les éliminer.

Une autre observation importante est que l'écriture de tests unitaires pour la classe à laquelle appartiennent ces méthodes en est facilitée : il est possible de tester séparément, d'une part, le calcul du nom et, d'autre part, la lecture du fichier. (De fait, la nécessité de tester chaque fonctionnalité contribue à faire préférer les méthodes courtes qui font une chose et une seule. Et *vice versa* : lorsqu'un programmeur écrit du code uniquement après avoir écrit un test correspondant, il a tendance à n'inclure qu'une seule fonctionnalité dans une même méthode.)

Enfin, les deux méthodes nouvellement créées seront par la suite autant d'occasions d'implémenter de nouvelles fonctionnalités sans devoir « réinventer la roue » – par exemple, la méthode `nomFichierConfiguration()` pourrait être très utile si jamais il fallait détecter des changements dans les fichiers de configuration pour les relire dynamiquement.

Encore une fois, nous noterons que les techniques de remaniement permettent de se rapprocher, sans ambiguïté, d'un code qui répond aux règles de simplicité.

Élimination du code «mort»

On peut considérer le code «mort» – des parties du code qui ne sont jamais utilisées par une autre partie, et qui n'ont donc aucun effet sur le projet en termes de fonctionnalités – comme relevant de la duplication (au sens où il apparaît une fois de trop... puisqu'il n'a aucune fonction) ou comme relevant de la séparation des idées (puisqu'il n'exprime aucune idée qui ait besoin d'être exprimée).

Quoi qu'il en soit, l'élimination du code inutile est également une des applications fréquentes des techniques de remaniement. Combinée à l'élimination des redondances et à la séparation en éléments distincts de toutes les idées importantes, elle conduit à un code qui, dans la mesure où il passe tous les tests avec succès, sera considéré comme optimal au sens d'XP.

Exemples avancés

Déplacer ou fragmenter des méthodes ne suffit pas, à première vue, pour obtenir ce qu'on attend généralement d'une conception objet – des classes liées entre elles par héritage, agrégation, ou composition, selon des structures dont la modularité permet aisément de créer de nouvelles fonctionnalités.

On peut toujours objecter que, si une structure «plate» ou «sans intérêt» suffit aux besoins du moment – «suffit», en tout cas, pour que le système soit conforme à la fois aux demandes du client et aux critères de simplicité – il n'y a aucune nécessité à mettre en place une structure plus intéressante.

Pour être complet, pourtant, un dernier exemple relativement simple nous permettra de montrer que l'application méthodique de remaniements visant uniquement à produire du code qui passe tous ses tests et exprime toutes les idées peut aboutir à des structures qui ressemblent à des «patterns» connus.

Supposons deux méthodes rédigées comme suit :

```
public Vector lireFichierConfiguration(String nomFichier) throws IOException {
   File fichier = new File(nomFichier);
   Vector lignes = new Vector();
   BufferedReader lecture = new BufferedReader(new FileReader(fichier));
   String ligneCourante;
   do {
      ligneCourante = lecture.readLine();
      if (ligneCourante != null) {
      lignes.add(ligneCourante);
      }
   }
   while (ligneCourante != null);
   lecture.close();
   return lignes;
```

```
}

public Vector afficherFichierConfiguration(String nomFichier) throws IOException {
   File fichier = new File(nomFichier);
   Vector lignes = new Vector();
   BufferedReader lecture = new BufferedReader(new FileReader(fichier));
   String ligneCourante;
   do {
      ligneCourante = lecture.readLine();
      if (ligneCourante != null) {
      System.out.println(ligneCourante);
      }
   }
   while (ligneCourante != null);
   lecture.close();
   return lignes;
}
```

À première vue, il semble difficile de réduire ces deux méthodes à des variantes d'un tronc commun, uniquement par extraction de parties du code vers des méthodes de la même classe. Certes, les deux méthodes sont très similaires – mais la différence entre les deux est localisée précisément au milieu, et qui plus est au sein d'une boucle, ce qui empêche d'extraire les parties communes sous forme de méthodes indépendantes. Cependant, en se concentrant initialement sur une des deux méthodes, il est possible d'obtenir la structure suivante :

```
public class LecteurConfiguration {

   private Vector lignes = new Vector();

   public void traiterFichier(String nomFichier) throws IOException {
      File fichier = new File(nomFichier);
      BufferedReader lecture = new BufferedReader(new FileReader(fichier));
      String ligneCourante;
      do {
         ligneCourante = lecture.readLine();
         if (ligneCourante != null) {
         traiterLigne(ligneCourante);
         }
      }
      while (ligneCourante != null);
      lecture.close();
   }

   public void traiterLigne(String ligneCourante) {
```

```
   lignes.add(ligneCourante);
   }

   public Vector resultat() {
   return lignes;
   }
}
```

Il n'est pas difficile de voir que la méthode permettant l'affichage peut être traitée de manière similaire ; en utilisant «Extraire une classe de base» après «Extraire une méthode», on obtiendra :

```
public abstract class TraitementConfiguration {

   public void traiterFichier(String nomFichier) throws IOException {
      File fichier = new File(nomFichier);
      BufferedReader lecture = new BufferedReader(new FileReader(fichier));
      String ligneCourante;
      do {
         ligneCourante = lecture.readLine();
         if (ligneCourante != null) {
         traiterLigne(ligneCourante);
         }
      }
      while (ligneCourante != null);
      lecture.close();
   }

   public abstract void traiterLigne(String ligneCourante);

}

public class LecteurConfiguration extends TraitementConfiguration {

   Vector lignes = new Vector();

   public void traiterLigne(String ligneCourante) {
   lignes.add(ligneCourante);
   }

   public Vector resultat() {
   return lignes;
   }
```

```
}

public class AfficheurConfiguration extends TraitementConfiguration {

   public void traiterLigne(String ligneCourante) {
   System.out.println(ligneCourante);
   }

}
```

Cette structure est appelée «patron de méthode» et permet d'introduire plus aisément de nouveaux traitements qui fonctionnent sur un même schéma général, exprimé dans la classe de base. Dans l'exemple précédent, ce modèle de conception ne surgit que de la motivation du programmeur d'éliminer les redondances et de rendre le code plus simple ; il n'est pas le résultat d'une préférence de principe envers ce modèle de conception, ou d'une volonté de «faire objet».

Autour des pratiques de programmation

Après cette présentation des différentes pratiques concernant la programmation, nous allons répondre à quelques questions qui peuvent encore se poser concernant leur mise en œuvre ou les différences à retenir par rapport aux méthodes «classiques».

Que faire du code existant ?

Compte tenu de la place particulière accordée dans XP aux tests automatisés et à une philosophie de conception qui interdit l'anticipation, on peut se demander s'il est possible d'adopter la démarche dans un autre contexte que celui d'un projet entièrement nouveau, sans aucun travail de conception préalable et surtout sans qu'il soit question de travailler à partir d'un code existant.

Il semble difficile d'appliquer les pratiques XP à la maintenance d'un code pour lequel aucun test unitaire n'a été rédigé, dont la conception – bonne ou mauvaise, mais probablement pas «simple» au sens que nous avons prêté à ce mot – ne peut pas être modifiée graduellement par l'application du remaniement (puisque celui-ci nécessite la présence de tests unitaires).

Même avec la plus grande conviction dans la pertinence de la méthode, les difficultés qui peuvent se poser pour l'appliquer à un projet en cours ou à la maintenance d'un projet précédent semblent insurmontables. Et ce d'autant plus que ces systèmes sont instables, truffés de défauts ou bien fastidieux à faire évoluer. Ne vaut-il pas mieux proposer de récrire entièrement ces systèmes, en se promettant que – grâce à XP – les choses iront mieux cette fois-ci ?

Paradoxalement, cette dernière option n'est pas particulièrement recommandée par XP. La récriture d'un système relève d'une décision stratégique, pas d'une décision technique ; elle appartient nécessairement au client. Il sera plus difficile – peut-être franchement douloureux – d'appliquer les principes de programmation XP sur un système existant que sur un projet «tout neuf». Mais c'est parfaitement possible.

Comment s'y prendre ?

L'essentiel est de ne pas chercher à tout faire d'un coup. Notamment, il ne faut pas entreprendre d'écrire une batterie complète de tests unitaires et fonctionnels pour la totalité du système existant. Il serait également au plus haut point périlleux de ne faire que du remaniement jusqu'à avoir obtenu une conception «optimale». (Sans tests unitaires, même un remaniement trivial peut entraîner des régressions ou de nouvelles anomalies.)

Une démarche possible consiste à identifier dans un premier temps une partie du système relativement isolée, peu dépendante du reste. Les programmeurs écriront quelques tests unitaires pour cette partie. L'équipe peut ensuite pratiquer le remaniement dans cette partie sans courir de risque. Cela leur permettra très certainement d'identifier une partie voisine pour laquelle il sera possible d'écrire de nouveaux tests unitaires, et ainsi de suite. Petit à petit, on verra s'accumuler une batterie de tests suffisante pour garantir que les remaniements destinés à obtenir une conception simple peuvent se faire sans introduire d'anomalies dans le code.

Notre XPérience

Un bon moyen pour assurer la transition vers XP d'une application existante, prise en main dans un contexte de maintenance, consiste, pour chaque opération de maintenance corrective ou évolutive, à écrire le ou les tests automatisés associés.

Rythme et ponctuations

Les rythmes de travail proposés par XP – que nous avons brièvement abordés dans le passage relatif aux tests automatisés – constituent un élément particulièrement important de la méthode. Parmi les différents paramètres qui caractérisent un projet, l'un des plus sensibles est le temps – rien d'étonnant, quand on sait combien de projets prennent des retards qui se comptent parfois en mois ou en années. Le découpage du temps – le rythme – peut par conséquent avoir des effets significatifs sur le projet.

Les cadences régulières imposées par plusieurs des pratiques aiguisent la sensibilité de l'équipe au passage du temps. Le rythme le plus rapide est celui qui ponctue l'activité de programmation : «Je teste, je code, je remanie.» En fonction de la nature du projet, cela peut représenter quelques minutes à peine, ou quelques dizaines de minutes tout au plus. Le rythme d'intégration – que nous aborderons au sujet de l'intégration continue – est de quelques heures, de même que le rythme auquel se forment et se défont les binômes de programmeurs.

Il en va de même pour les cycles plus larges, que nous découvrirons au cours des chapitres suivants. Quotidiennement, ou au moins plusieurs fois par semaine, une équipe XP fait le point sur les progrès constatés, ou les difficultés éventuellement rencontrées. Au sein de chaque itération, son travail concerne un certain nombre de fonctionnalités dont on saura, au bout de quelques jours, qu'elles sont définitivement implémentées – les tests fonctionnels le garantissent. Enfin, au terme de chaque itération, une nouvelle version du système est remise au client, et un certain nombre d'itérations produisent régulièrement une version «majeure».

Sentiment de succès

Ces rythmes – du plus court au plus long – ont pour but d'alimenter un sentiment de succès permanent. Après avoir implémenté correctement le code correspondant à un nouveau test unitaire, les programmeurs sont récompensés par la fameuse «barre verte» de l'outil de test indiquant que tout est en ordre. C'est le moment de taper sur l'épaule de son partenaire – voilà une bonne chose de faite ! Après une intégration réussie, on peut s'accorder une pause avant de recommencer voire, à la fin d'une itération, organiser un pot si tout s'est bien passé…

C'est également en étant attentif à ce sentiment de confiance et à la régularité de tous ces rythmes qu'une équipe XP est en mesure d'identifier rapidement des dérives naissantes, des dangers potentiels. Lorsqu'on met plusieurs minutes à chercher comment écrire le test pour une nouvelle fonctionnalité, c'est qu'il y a un problème quelque part dans la conception existante. C'est sans doute le moment d'insister un peu plus sur le remaniement.

De même, si l'on rencontre des difficultés au moment de l'intégration, si un test fonctionnel se révèle plus retors que prévu, si on accumule un retard important au cours d'une itération, c'est que quelque chose est en train de ralentir l'équipe, que celle-ci doit rapidement identifier pour pouvoir repartir du bon pied. Tant que les programmeurs – et leurs clients, et leur encadrement – restent attentifs à ces signaux, ils seront capables d'éviter des difficultés majeures. Si, en revanche, ils se cachent la tête dans le sable – si, malgré tous les retours d'informations dont XP leur permet de disposer, ils choisissent de les ignorer –, rien ne saurait sauver le projet.

La documentation et XP

L'un des débats autour d'XP a trait aux documents produits au cours d'un projet. Cette question est souvent résumée sous une forme pour le moins caricaturale : «On n'écrit pas de documentation dans XP». S'il est vrai que, comme les autres méthodes dites agiles, XP cherche à éliminer toute documentation de type paperasse, cette position doit être nuancée.

Il faut distinguer plusieurs catégories de documents parmi ceux que peuvent être amenés à produire des programmeurs : d'une part, la documentation destinée aux utilisateurs ou aux administrateurs d'un système ; d'autre part, tout ce qui peut être regroupé sous l'expression «documentation technique» – documents de conception, scénarios de test ou d'utilisation,

diagrammes, etc. ; enfin, les documents liés au projet, tels que rapports, plannings et comptes rendus.

Aucune controverse réelle n'existe concernant les documents d'utilisation, d'administration ou d'installation. Rien ne saurait s'opposer, notamment, à ce qu'ils soient produits par l'équipe technique si les circonstances l'exigent. (Il faut néanmoins observer que la rédaction et la programmation ne réclament pas les mêmes compétences et que l'on ne peut en général pas s'attendre à un résultat parfait si ce sont les programmeurs qui se chargent de la rédaction des manuels.) Toutefois, XP recommande que cette tâche de rédaction – comme toute activité consommatrice de temps – soit planifiée au même titre que les tâches techniques, selon les mêmes principes.

Comme l'accent est mis dans XP sur la proximité physique des différents interlocuteurs d'un projet – en particulier, la présence du client dans l'équipe – et comme la conformité du produit aux besoins est, en principe, garantie par la présence de tests de recette, l'importance que peuvent avoir les documents de projet y est largement réduite.

Le code e(s)t la documentation

La documentation technique, et notamment les documents de conception ou d'architecture, sont en revanche au cœur d'une réelle divergence de points de vue. L'approche «classique» maintient que l'on ne peut programmer efficacement qu'après avoir entièrement analysé le problème sous une forme schématique, analyse dont le produit est un document écrit de spécification ou de conception. Cette analyse et ces documents jouent un rôle prépondérant par rapport au code proprement dit : s'ils ont été correctement réalisés, l'activité de programmation à proprement parler se résume à une traduction dans le langage de programmation utilisé de la conception préalable.

Le principal argument en faveur de ce point de vue a toujours été celui du «coût du changement» : il serait dix fois plus coûteux de revenir sur une décision au moment de la programmation qu'au moment de la conception ; plus généralement, corriger une erreur est d'autant plus coûteux qu'elle intervient tardivement dans le cycle spécification, conception, programmation, tests, mise en production.

A contrario, XP fait valoir, d'une part, que le remaniement permanent du code et que la conception la plus simple possible permettent en fait de revenir aisément sur des décisions de conception, même lorsque le projet est très avancé ; et, d'autre part, que le code source et les documents de conception finissent systématiquement par diverger, ce qui finit par rendre inutilisables ces documents. Par ailleurs, ce n'est bien souvent qu'une fois implémentée qu'on se rend compte des défauts d'une conception donnée.

Pour XP, par conséquent, la principale source de documentation technique est le code source lui-même. Les tests fonctionnels jouent le rôle de documents de spécification : ils précisent de façon formelle ce que le programme doit faire ; les tests unitaires servent à documenter la

conception. Les personnes auxquelles s'adressent ces documents sont des programmeurs – ils sont donc en mesure de comprendre le langage dans lequel ces documents sont rédigés.

La place du programmeur

Cette insistance à considérer le code source comme la principale production du projet est caractéristique de la méthode et s'accompagne d'une autre inversion des valeurs «traditionnelles» : le programmeur est considéré comme un acteur essentiel du projet. Il n'y occupe pas une place de simple exécutant, interchangeable, subordonnée aux rôles plus nobles de l'architecte ou du chef de projet dont la compétence serait seule déterminante pour le succès de l'entreprise.

Pour clore ce chapitre sur la question avec laquelle nous l'avons ouvert, nous noterons donc que, contrairement à d'autres méthodes qui semblent vouloir accorder une importance réduite à l'activité de programmation et aux programmeurs eux-mêmes, XP propose à ces derniers, non seulement des outils et une discipline permettant de pratiquer leur métier plus efficacement, mais également une certaine reconnaissance, pour ne pas dire, une dignité retrouvée.

4

Zoom sur les tests

Si on ne prenait garde d'y réfléchir attentivement, il pourrait nous sembler que programmer se résume à écrire des instructions dans un langage de programmation.

– Ward Cunningham

Le chapitre précédent nous a donné l'occasion d'aborder ce que l'on pourrait appeler le noyau d'XP : un style de programmation à part entière, organisé autour d'une utilisation très particulière des tests de régression automatisés, dénommée «développement piloté par les tests».

Nous allons aborder plus avant dans cet ouvrage les pratiques d'XP qui vont au-delà de l'activité du programmeur et qui structurent la vie du projet à proprement parler ; le lecteur qui souhaite obtenir une vue d'ensemble d'XP et de son application à des projets de développement peut d'ailleurs ignorer le présent chapitre en première lecture.

Il nous a cependant semblé important – notamment pour le lecteur qui souhaite passer rapidement à la pratique – de fournir rapidement quelques-unes des clés de cette démarche de programmation, d'en souligner quelques aspects très concrets : pas tant parce qu'ils sont plus importants que les autres pratiques d'XP, mais parce que, pour de nombreux programmeurs, ils constituent le «sésame» qui permet de faire leurs premiers pas avec la méthode, à temps perdu, sans qu'il soit nécessaire pour cela de remettre en cause aucun aspect de leurs projets.

Volontairement lapidaire, ce chapitre abordera dans un premier temps les outils habituels de mise en place des tests à la mode XP, puis décrira le mode d'utilisation de ces outils dans les tâches quotidiennes de programmation.

Les outils : la famille xUnit

Un ouvrage sur XP ne serait pas complet s'il ne présentait pas l'outil fétiche des programmeurs XP : un module minimaliste dédié aux tests, comprenant les objets de base qui permettent de définir des séries de tests unitaires, et plusieurs interfaces utilisateurs – lesquelles sont différentes les unes des autres, mais réunies par un lien de parenté évident.

Cet outil créé par Erich Gamma et Kent Beck a été adapté pour la quasi-totalité des langages de programmation pratiqués par plus d'une poignée de développeurs dans le monde. On en connaît ainsi des implémentations en Java (JUnit), en C++ (CppUnit), en Delphi (Dunit), en Visual Basic (VBUnit), en Smalltalk/Squeak (SUnit), en Ruby (runit), en Python (PyUnit)… La liste est longue et ne cesse de s'allonger ; c'est devenu un exercice courant pour les programmeurs XP d'un certain niveau que de découvrir un nouveau langage de programmation en écrivant la version adaptée à ce langage du « framework », version qui est traditionnellement nommée selon le schéma « *x*Unit », ou *x* est une lettre ou une syllabe rappelant le nom du langage.

Un rapide panorama de quelques-unes de ces versions permettra de découvrir rapidement ce qu'il est possible d'attendre d'un outil de tests unitaires.

Degré 0 : interface texte, avec runit

Deux premiers critères sont essentiels en matière de tests unitaires. Ceux-ci doivent absolument :

- pouvoir être exécutés rapidement ; s'il faut attendre plusieurs heures ou même de longues minutes avant de voir le résultat de ses tests, l'ennui l'emportera chez les programmeurs qui ne prendront plus la peine de les exécuter – et donc, rapidement, ne se donneront plus la peine de les écrire ;
- fournir un résultat déterministe et sans ambiguïté, de type binaire : OK ou KO ; s'il faut lire soigneusement le résultat d'un rapport de tests pour savoir si le programme fonctionne ou non, les tests seront bientôt à la merci d'une erreur humaine – or, c'est l'erreur humaine qui les a rendus nécessaires à l'origine.

La figure 4-1 présente un rapport de tests exemplaire à cet égard : il a été exécuté si rapidement que le temps total est arrondi à 0 seconde !

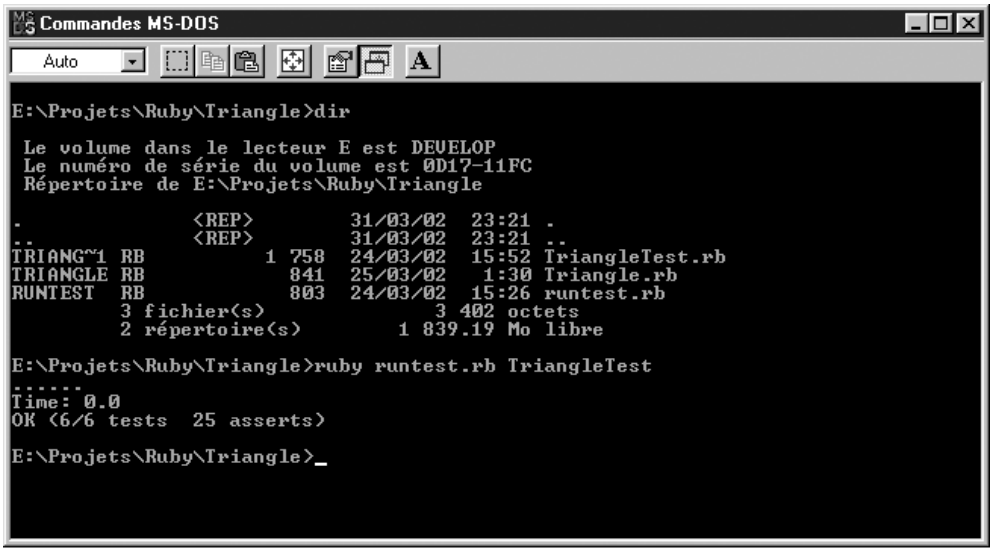

Figure 4-1. runit – tests passés avec succès

En l'occurrence, la mention «OK» nous signale que tous les tests pour le programme (une modeste classe décrivant un triangle, dotée de 6 tests) ont été exécutés avec succès.

Un autre critère important est la lisibilité des tests. Ces derniers ne comporteront idéalement que des syntaxes élémentaires, créations d'objets, appels de méthodes et appels à la méthode `assert` spécifique aux tests unitaires. Ainsi l'exemple suivant[1], en Ruby (langage pourtant peu connu), ne devrait laisser perplexe aucun programmeur, même si les détails de la syntaxe lui échappent :

```
class TriangleTest < RUNIT ::TestCase
   def testValidTriangle
      t = Triangle.new(3, 4, 5)
      assert(t.isTriangle)
      t = Triangle.new(0,0,0)
      assert(!t.isTriangle, "too small")
      t = Triangle.new(1,2,3)
      assert(!t.isTriangle, "not connected")
      t = Triangle.new(3,-4,-5)
      assert(t.isTriangle, "negative length")
   end
end
```

1. Emprunté à Ron Jeffries.

Le test se contente de créer des triangles représentés comme des triplets de longueurs entières, et donc de tester dans divers cas de figure le résultat affecté à `isTriangle`.

Lorsqu'un test échoue, même l'outil de test le plus fruste donnera quelques détails complémentaires qui fournissent au programmeur des informations de façon à isoler la raison de cet échec, comme l'illustre le rapport de test présenté en figure 4-2.

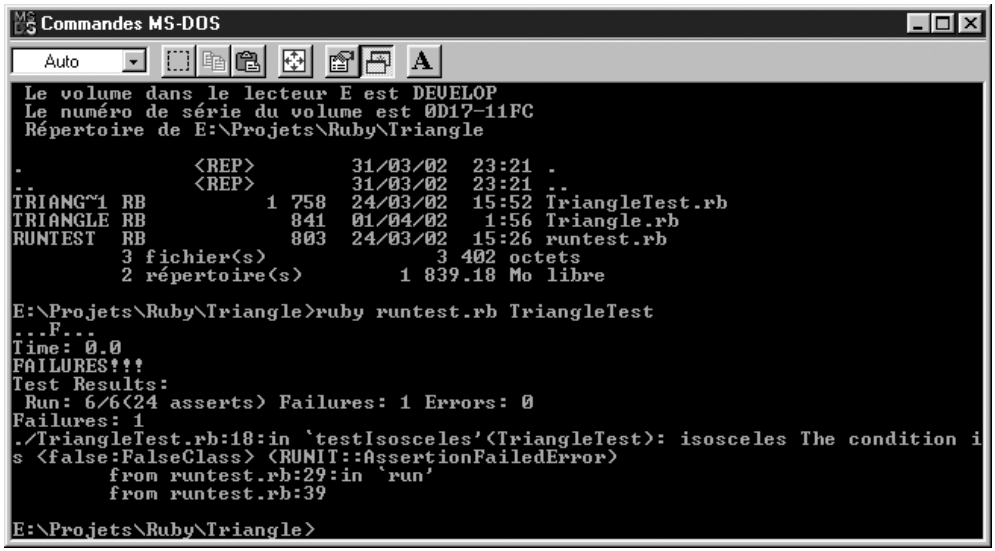

Figure 4-2. runit – tests avec un échec

Degré 1 : interface graphique, avec JUnit

L'essentiel du «framework» xUnit se compose de quelques classes, qui encapsulent les concepts fondamentaux : TestCase représente un «scénario de tests» ; TestSuite permet l'agrégation de ces scénarios en méta-scénarios qui peuvent à leur tour être ainsi agrégés. Ces deux interfaces sont les plus visibles pour les programmeurs qui utilisent xUnit ; s'y ajoutent également TestResult, TestRunner, et quelques autres qui facilitent notamment l'ajout à la version de base de xUnit d'extensions telles qu'une interface graphique.

La version canonique de cette interface est familière à tous les programmeurs XP qui lui vouent, sinon un culte (ce n'est pas leur genre), du moins plus de tendresse qu'il n'est sans doute de raison : c'est la fameuse «barre verte», ou «barre rouge», qui sanctionne de son jugement sans appel chaque exécution des tests.

Nous pouvons comparer à l'exemple donné plus haut en Ruby un test unitaire[1] écrit en Java, concernant cette fois l'embryon d'un tableur («*spreadsheet*») :

1. Cet exemple est inspiré du «Test-First Challenge» de Bill Wake.

```
public void testMultiply() {
   Sheet sheet = new Sheet();
   sheet.put("A1", "=2*3*4");
   assertEquals("Times", "24", sheet.get("A1"));
}
```

Cet exemple nous permet également d'illustrer un autre principe fondamental des tests unitaires – qu'il aurait été prématuré d'énoncer avant d'en avoir vu une illustration dans deux langages distincts : les tests unitaires doivent être écrits dans le même langage que celui qui est utilisé pour l'application qu'on est en train de programmer.

C'est cette exigence, en particulier, qui a conduit à l'émergence de xUnit comme un produit *open source*, facile à adapter vers d'autres langages. Un produit commercial, fonctionnant sur une gamme limitée de plates-formes matérielle ou système, ou nécessitant l'usage d'un langage spécifique, n'aurait jamais rencontré le même succès.

C'est également ce qui permet au programmeur utilisant xUnit de tester «très près» de son code, en manipulant directement les classes, objets ou fonctions qu'il implémente avec une «granularité» de manipulation idéale.

Voici la «barre verte» que nous évoquions plus haut et dont l'apparition est donc maintenant attendue, sous Junit (voir figure 4-3).

Figure 4-3. JUnit – tests passés avec succès

Outre cette interface rudimentaire, JUnit dispose également d'une interface plus évoluée, utilisant les bibliothèques Swing, qui permet notamment de présenter sous la forme d'une arborescence l'intégralité des tests unitaires constituant une série (voir figure 4-4).

Figure 4-4. Junit – tests avec un échec

Cette présentation en liste permet de «pointer» le (ou les) test(s) qui ont donné lieu à un échec ; nous en voyons ici un exemple avec une «barre rouge», montrant également comment JUnit exploite le mécanisme de signalement des erreurs propre à Java, les exceptions, pour indiquer dans quel contexte l'erreur qui a mis le test en échec s'est produite.

Degré 2 : intégration à l'environnement, avec SUnit

La communauté Smalltalk, traditionnellement acquise au style de programmation prôné par XP, bénéficie d'une avance confortable en termes d'intégration des outils de tests unitaires aux environnements de travail courant.

Une telle intégration n'est bien entendu pas nécessaire pour pratiquer efficacement les tests selon XP, mais elle laisse entrevoir l'intérêt qu'on peut y trouver. Ici, nous pouvons observer qu'après une exécution des tests pour un interpréteur XML, qui s'est soldée par la «barre rouge» tant redoutée, le programmeur Smalltalk utilisant SUnit peut se rendre directement dans la fenêtre de débogage correspondant à l'appel de la méthode de test qui a été mise en échec. Il pourra y effectuer directement les modifications nécessaires (voir figure 4-5).

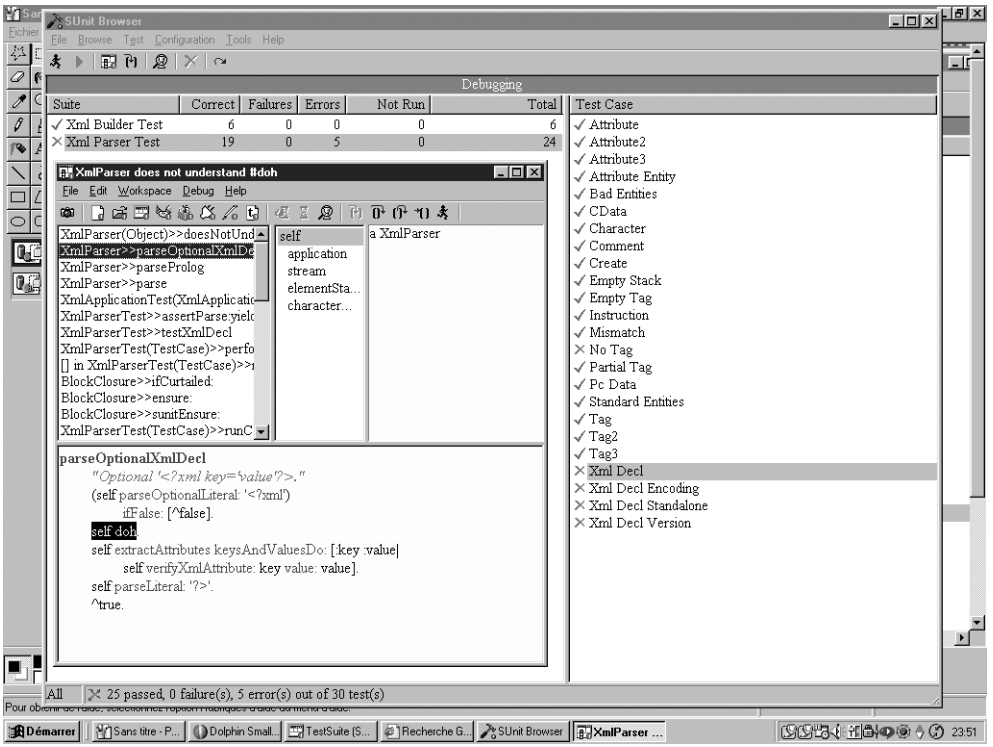

Figure 4-5. SUnit browser – tests avec un échec

Des environnements de développement conçus de A à Z pour favoriser le style de programmation prôné par XP sont également en cours de réalisation, tel le projet ENSU[1]. Plus terre à terre, les éditeurs d'environnements de développement – notamment Java – se font concurrence pour intégrer de façon plus performante l'outil JUnit à leurs différents produits phares.

Comment tester avant de coder

La règle recommandée par XP est la suivante : «Ne jamais écrire une seule ligne de code qui ne soit justifiée par un test qui échoue. » Comme nous l'avons indiqué au chapitre précédent, elle fonctionne de concert avec cette autre règle : «Exprimer chaque idée une fois et une seule. »

1. *http://www.ensu.net.*

De ces deux règles, la première est la plus difficile à suivre pour les programmeurs qui abordent XP pour la première fois. Certes, on peut se dire que d'autres développeurs ont réussi à la suivre et que ce n'est donc qu'une question de volonté, d'attitude. Pour autant, il apparaît souvent très difficile de se conformer à un mode de pensée entièrement nouveau, sans aucun exemple concret sur lequel on puisse s'appuyer. «Comment est-ce que je peux tester quelque chose avant même de l'avoir codé ? » Voir des exemples de code déjà écrit, accompagné de ses tests, ne sert bien sûr à rien, puisque rien ne permet de distinguer *a priori* des tests écrits avant le code et des tests écrits après.

Ce qu'il est donc crucial de se représenter, c'est la dynamique qui préside au développement piloté par les tests. Ce point pourrait en soi faire l'objet d'un livre à part entière, mais nous nous contenterons ici d'en donner les grandes lignes.

Décomposition des tâches : la «liste de tests»

Contrairement à ce que pourraient laisser supposer certaines idées reçues, XP ne recommande pas de «coder sans réfléchir». Bien au contraire, le développement piloté par les tests requiert que l'on ait tout d'abord une idée assez précise de ce qu'il s'agit de faire pendant l'implémentation. Par «assez précise», il faut comprendre qu'on saura la décomposer en tâches atomiques. L'expression «tâches atomiques» signifie que la tâche d'implémentation doit être *totalement* évidente.

Bien entendu, ce qui est évident pour une personne ne l'est pas nécessairement pour une autre, et *vice versa*. L'intérêt de la démarche, toutefois, tient en ceci que si une tâche donnée doit être décomposée en deux sous-tâches pour une personne, alors que pour une autre la tâche d'origine est évidente, les deux programmeurs en question arriveront quand même au résultat souhaité !

Cette méthode de décomposition conduit à dresser, pour n'importe quelle tâche d'implémentation, une liste de tests – que nous gérerons un peu comme une liste de courses, en rayant chaque article lorsque nous l'aurons implémenté. Comme c'est d'XP qu'il s'agit, cette liste sera bien sûr écrite sur… une fiche cartonnée.

Reprenons l'exemple donné au chapitre précédent : il s'agit de compter les mots contenus dans un document.

Ma liste de tests

Savoir compter le nombre de mots dans un document

Est-ce une tâche évidente ? *A priori*, non. On va donc chercher à la décomposer :

Ma liste de tests

~~Savoir compter le nombre de mots dans un document~~

Savoir effectuer une boucle sur chaque ligne du document

Savoir compter le nombre de mots dans une ligne

Savoir faire le total du décompte pour chaque ligne

La première tâche peut nous maintenant nous paraître évidente à tester – à supposer que nous disposions déjà des méthodes adéquates dans l'objet Document. La dernière peut nous sembler également triviale, alors que la deuxième nécessite d'être à nouveau décomposée :

Ma liste de tests

~~Savoir compter le nombre de mots dans un document~~
Savoir effectuer une boucle sur chaque ligne du document
~~Savoir compter le nombre de mots dans une ligne~~
Savoir faire le total du décompte pour chaque ligne
On compte comme un seul mot la chaîne 'toto'
On compte comme deux mots la chaîne 'to to'
On compte comme trois mots la chaîne 'to to to'

Ces derniers exemples peuvent sembler ridiculement simples, presque triviaux. L'idée est précisément de décomposer le problème en étapes suffisamment petites pour qu'elles n'appellent presque aucune réflexion pendant la mise en œuvre du cycle de rédaction des tests, qui seront écrits les uns après les autres à un rythme régulier.

Le cycle tester/coder/remanier par étapes

Pour illustrer ce rythme, ainsi que l'application détaillée de la «recette» qu'il représente, nous allons examiner un exemple[1] encore plus simple : une méthode calculant la série de Fibonacci. Nous en connaissons la définition mathématique – chaque terme de la série s'obtient par la somme des deux termes précédents. Nous en connaissons les premiers termes : 0, 1, 1, 2, 3, 5, 8…

L'intérêt de cet exemple n'est pas de présenter une implémentation innovante de la série ! Il s'agit simplement de donner rapidement un aperçu du rythme recherché. Il n'est pas utile de décomposer la liste de tests – nous allons traiter chaque terme de la série, dans l'ordre. Le cycle se décompose comme suit :

1. Écrire un nouveau test, compiler.
2. Lancer tous les tests ; seul le nouveau test doit échouer.
3. Écrire l'implémentation la plus simple qui puisse passer ce test.

1. Cet exemple a été compilé et exécuté sous Java 1.3. Une version plus complète du code définitif est donnée en annexe.

4. Lancer tous les tests ; tous doivent maintenant passer.
5. Généraliser le code en supprimant les redondances.

Nous traiterons cet exemple en Java. Voici notre tout premier test :

```
import junit.framework.*; // Ne sera pas répété par la suite

public class FibTest extends TestCase {
   // Junit a besoin de la ligne suivante
   public FibTest(String name) {super(name);} // Ne sera pas répété par la suite
   public void testFib() {
      assertEquals(0,fib(0));
   }
}
```

Pour l'instant, nous n'avons pas dépassé l'étape 1, puisque le test ne peut pas compiler. Nous nous rappelons que l'objectif est d'obtenir un test qui échoue, et nous transformons donc l'appel à une méthode inexistante en un appel à une méthode sur un nouvel objet :

```
public class FibTest extends TestCase {
   public void testFib0() {
      assertEquals(0,new Fib().fib(0));
   }
}
public class Fib {
   public int fib(int terme) {
      throw new RuntimeException("Pas implémenté");
   }
}
```

À ce stade, nous sommes en mesure de satisfaire à l'étape 2 – le test peut être exécuté, et échoue. Nous cherchons maintenant la solution la plus simple pour le faire passer :

```
public class FibTest extends TestCase {
   public void testFib0() {
      assertEquals(0,new Fib().fib(0));
   }
}
public class Fib {
   public int fib(int terme) {
      return 0;
   }
}
```

Miracle, ça marche ! Nous avons franchi l'étape 4. L'étape 5 est satisfaite sans aucun effort : avec une seule ligne de code, il n'y a rien à généraliser. Il est maintenant possible de recommencer à l'étape 1 avec un nouveau cas de test.

```
public class FibTest extends TestCase {
   public void testFib0() {
      assertEquals(0,new Fib().fib(0));
   }
   public void testFib1() {
      assertEquals(1,new Fib().fib(1));
   }
}
public class Fib {
   public int fib(int terme) {
      return 0;
   }
}
```

L'implémentation est déjà moins évidente. Voici l'idée la plus simple qui nous vienne à l'esprit :

```
   public int fib(int terme) {
      return terme;
   }
```

qui nous amène cependant à l'étape 5. Mais les choses se corsent avec le test suivant :

```
public class FibTest extends TestCase {
   public void testFib0() {
      assertEquals(0,new Fib().fib(0));
   }
   public void testFib1() {
      assertEquals(1,new Fib().fib(1));
   }
   public void testFib2() {
      assertEquals(1,new Fib().fib(2));
   }
}
public class Fib {
   public int fib(int terme) {
      return terme;
   }
}
```

Notre première solution ne fonctionne plus ; nous écrivons donc :

```
   public int fib(int terme) {
      if (terme == 0) return 0;
      return 1;
   }
```

Une structure commence à s'esquisser, mais nous passons au test suivant :

```
public class FibTest extends TestCase {
   public void testFib0() {
      assertEquals(0,new Fib().fib(0));
   }
   public void testFib1() {
      assertEquals(1,new Fib().fib(1));
   }
   public void testFib2() {
      assertEquals(1,new Fib().fib(2));
   }
   public void testFib3() {
      assertEquals(2,new Fib().fib(3));
   }
}
public class Fib {
   public int fib(int terme) {
      if (terme == 0) return 0;
      return 1;
   }
}
```

Le nouveau test échoue. Nous continuons sur notre lancée et notre implémentation devient alors :

```
public class Fib {
   public int fib(int terme) {
      if (terme == 0) return 0;
      if (terme <= 2) return 1;
      return 2;
   }
}
```

Tous nos tests passent ; nous sommes à l'étape 4. Pour une fois, nous remarquons que l'étape 5 donne un résultat intéressant. En effet, nous pouvons remanier cette méthode en :

```
   public int fib(int terme) {
      if (terme == 0) return 0;
      if (terme <= 2) return 1;
      return 1+1;
   }
```

Attention

Nous ne sommes pas ici en train de «découvrir», comme par hasard, l'implémentation récursive de la série de Fibonacci ou quoi que ce soit de nouveau sur cette série. Nous connaissons bien sûr la «solution», et c'est ce qui nous motive à effectuer cette transformation et la suivante. La démarche de développement piloté par les tests n'a pas pour objet la «découverte» d'une implémentation, en nous exonérant de toute réflexion préalable ; elle doit guider vers une implémentation correcte, en sécurisant le travail à chaque étape.

Cela peut sembler une modification triviale, mais nous remarquons alors que, lorsque fib() est appelée avec le paramètre 2, cette implémentation est équivalente à :

```
public int fib(int terme) {
   if (terme == 0) return 0;
   if (terme <= 2) return 1;
   return fib(terme-1)+fib(terme-2);
}
```

qui peut être encore simplifiée en

```
public int fib(int terme) {
   if (terme <= 1) return terme;
   return fib(terme-1)+fib(terme-2);
}
```

qui donne l'implémentation récursive classique. Cette implémentation est capable de passer non seulement les tests que nous avons écrits jusqu'à présent, mais aussi tous ceux que nous pourrions vouloir écrire ensuite !

Attention

On peut ne pas déceler si tôt le «déclic» que constitue ce remaniement (le mot anglais *refactoring* prend ici son sens le plus mathématique puisque c'est bien d'une factorisation qu'il s'agit). Dans ce cas, on continuera à aligner les tests les uns après les autres jusqu'à ce que la vraie structure de la solution devienne évidente et s'impose par la nécessité d'éliminer la duplication… y compris la duplication entre le test et le code : ainsi on peut percevoir le «**2**' »que nous remanions en «**1+1** » comme une duplication du «**2** » présent dans **testFib3**.

La structure d'un test : les 3 A

Nous avons vu comment aller d'une tâche de programmation aux tests qui la composent, puis nous avons énuméré les étapes à franchir pour écrire après chaque test le code correspondant. En descendant maintenant au niveau le plus bas, on peut se demander comment est écrit un seul test ?

Sous réserve des nombreuses variations qui sont bien entendu possibles autour de ce thème, les tests unitaires répondent en général à une structure que l'on retrouve presque systématiquement, appelée les «3 A» : Acteurs, Action, Assertion.

La première partie consiste à instancier un ou plusieurs objets, éventuellement à les initialiser d'une façon bien précise – un peu comme on place des acteurs sur une scène. Dans la partie suivante, on appelle en général une seule méthode d'un de ces objets – ce qui déclenche souvent une action (ou plusieurs) bien précise de leur part. Enfin, presque tous les tests se terminent par une assertion ou une série d'assertions, qui vérifie que, compte tenu des conditions initiales, l'action demandée aux objets a bien eu l'effet escompté.

Même l'exemple précédent, volontiers simpliste, de la série de Fibonacci répond à ce schéma, à condition d'écrire chaque test de la manière suivante :

```
public void testFib1() {
   FibTest test = new FibTest(); // Acteur
   Int result = test.fib(0); // Action
   assertEquals(0,result); // Assertion
}
```

Quelques chiffres

Lorsqu'on débute dans une discipline, il est toujours utile de pouvoir s'attribuer un «score», un indicateur numérique qui témoigne d'un progrès. Une fois passé le stade du débutant, bien entendu, la pratique elle-même devient plus importante que l'acquisition de cette information rudimentaire.

Il n'est pas du tout exceptionnel, pour une application non triviale, d'écrire par mois et par programmeur une quantité de tests unitaires de l'ordre de la centaine. Dans une application nouvelle, une centaine de tests unitaires représente déjà des fonctionnalités significatives dans un «noyau» fonctionnel, sans nécessairement prendre en compte une interface utilisateurs, la gestion de données persistantes, etc. Par exemple, cent tests unitaires, dans le domaine de la finance, couvrent le cœur d'une application destinée à simuler un marché actions, y compris le protocole d'accès à la diffusion des données *via* le réseau et une gestion simplifiée des carnets d'ordres et de l'exécution de ces derniers.

Le nombre de tests unitaires peut varier considérablement, selon qu'ils regroupent plus ou moins d'assertions. Une «densité» typique semble être de 4 ou 5 assertions par test, mais on peut atteindre jusqu'à 20 assertions par test.

Dans le cadre d'une application nouvelle, la stricte application des principes du développement piloté par les tests conduit assez naturellement à un volume de lignes de code équivalent entre code de tests et code principal.

Conseils pratiques

Les techniques que nous exposons ci-après ne sont pas une composante obligatoire de la démarche de développement piloté par les tests. Leur utilisation relève du style de chaque développeur.

Gestion des dépendances entre classes

L'un des «effets de bord» bénéfiques des tests unitaires est la mise à plat des dépendances entre les classes de l'application. En effet, dans la partie Acteurs d'un test (celle qui consiste à mettre en place son contexte d'exécution), les programmeurs doivent instancier tous les objets utilisés par le code testé. Lorsque cette partie devient trop longue ou trop complexe, cela constitue souvent un indice pour les programmeurs : les dépendances de la classe testée sont trop nombreuses ou trop profondes, le code commence à «sentir mauvais» (les auteurs d'XP parlent de *code smells*), et il est temps de procéder à un remaniement pour le simplifier.

En programmation objet, la rupture des dépendances passe le plus souvent par l'introduction de classes d'interface : si une classe A dépend d'une classe B, on peut inverser la dépendance entre A et B en introduisant une classe d'interface I dont hérite B. La classe I représente ainsi une sorte de contrat auquel doit se conformer B pour fonctionner avec A, et ce contrat fait partie intégrante de A. Dans ce nouveau schéma, on peut considérer que la classe B dépend du couple formé par A et I – la dépendance est inversée, et A peut être testé sans avoir à utiliser B, en ayant recours à des *bouchons de test*.

Utilisation de bouchons de test

Les opérations de factorisation et de rupture des dépendances menées pendant les phases de remaniement conduisent à des situations telles que la classe à tester dépende de classes d'interface plutôt que de classes concrètes. Les programmeurs peuvent en tirer parti et faciliter l'écriture des tests en substituant aux classes concrètes définies dans l'application des classes plus simples, dédiées aux tests, qui héritent des mêmes interfaces. Ces substituts sont communément appelés «bouchons de tests».

Une alternative s'offre alors aux programmeurs pour réaliser ces bouchons. En premier lieu, ils peuvent utiliser la classe de test elle-même, en lui faisant implémenter l'interface attendue par l'objet testé. Cette technique dite du s*elf-shunt* est la plus pratique, mais elle pâtit de deux limitations principales : d'une part, on ne dispose que d'une seule instance (celle du test lui-même) à fournir au code testé, et, d'autre part, le code spécifique écrit pour implémenter l'interface utilisée n'est aisément accessible que dans cette classe de test.

Ces limitations peuvent amener les développeurs à opter pour la seconde possibilité, qui consiste à créer des classes spécifiques pour jouer le rôle de bouchons – on parle alors de *mock objects*.

Quelle que soit la technique adoptée, l'intérêt des bouchons de tests ne se limite pas à ce qu'ils facilitent l'écriture du test en réduisant le nombre d'acteurs mis en jeu. En effet, ils autorisent également un contrôle beaucoup plus fin des conditions de test : les programmeurs peuvent choisir sans contraintes le comportement de leurs bouchons, et placer ainsi le code testé dans les situations voulues de manière bien plus aisée que s'ils utilisaient les classes concrètes disponibles dans l'application. Ils peuvent aussi contrôler plus facilement les opérations effectuées par le code testé sur les interfaces qu'il utilise.

Tester la mise en œuvre de l'héritage

L'un des grands principes de conception objet, le principe de substitution de Liskov, caractérise l'héritage en ceci que, dans toute fonction attendant un objet d'une classe donnée, on doit pouvoir substituer un objet d'une classe dérivée sans altérer le comportement du logiciel. Une autre façon d'exprimer ce principe consiste à énoncer que les classes dérivées doivent respecter le contrat établi par la classe de base.

Dans le langage Eiffel, qui supporte un mécanisme d'assertions (*Design by contract*), les contrats qui définissent les services d'une classe peuvent être spécifiés, et ils sont automatiquement hérités par les classes dérivées qui doivent donc s'y conformer. Dans les autres langages, on peut utiliser les tests unitaires pour obtenir le même résultat.

En effet, les tests unitaires, selon XP, définissent justement le comportement attendu, le contrat, d'une classe avec ses clients. Lorsque cette classe doit être dérivée, on peut vérifier l'application du principe de substitution de Liskov en soumettant les classes dérivées aux tests unitaires déjà écrits pour la classe de base.

Pour que cela fonctionne, il faut que le test unitaire de la classe de base soit conçu dans cette optique, ou adapté à cet effet. Les techniques sont diverses, et parfois spécifiques au langage de programmation. Pour l'essentiel, il faut s'assurer qu'il ne manipule que des références à des objets, et ne crée pas lui-même les objets à tester :

- Le cas le plus simple est celui des objets sans état (*stateless*), ou n'ayant qu'un constructeur sans arguments, ou qui ont une méthode de réinitialisation ou de remise à zéro (par exemple, `string.clear`). En effet, un seul et même objet peut être utilisé par toutes les méthodes de test. Dans ce cas, il suffit que la classe fournisse une méthode d'initialisation permettant d'affecter l'objet à tester, qui sera manipulé par tous les tests. Il est donc facile de réutiliser ce test avec des classes dérivées.
- Pour des objets plus complexes, et notamment lorsqu'il est nécessaire de tester différentes manières de construire les objets, la classe de test peut éviter de construire elle-même les objets en déléguant cette responsabilité à une classe (ou une fonction) *factory*, qui doit elle-même être paramétrable (passée en argument à la construction, ou *template* dans le cas du C++).

- Enfin, les relations d'héritage entre les classes testées peuvent se déduire des relations d'héritage entre les classes de test. Dans ce cas, le mieux est d'utiliser le *design pattern* «Template Method» pour imposer le déroulement des tests de base, tout en déléguant aux classes de tests dérivées la construction des objets.

Ces techniques plus avancées permettent d'éviter la duplication du code de test dans les cas d'héritage, tout en renforçant l'intégrité de la conception objet dans le respect du principe de substitution de Liskov.

Tester les classes génériques

Il s'agit ici de classes génériques au sens où elles sont conçues en faisant abstraction d'un ou plusieurs types d'objets manipulés, sachant qu'elles peuvent ensuite être spécialisées (il s'agit du mécanisme des templates en C++ et de celui des classes génériques en Eiffel).

Dans un contexte de programmation «classique», ces classes représentent parfois une source d'erreurs découvertes tardivement, lorsqu'elles sont spécialisées pour la première fois avec un type qui perturbe le fonctionnement prévu de la classe générique.

Avec les tests unitaires d'XP, on peut se prémunir contre ce genre de surprise. En effet, toute classe ayant sa propre classe de test unitaire, l'idéal pour tester les classes génériques est de rendre la classe de test également générique. Dès lors, chaque fois que le code applicatif a besoin de spécialiser la classe générique avec un type donné, on peut, sans effort et sans dupliquer de code, spécialiser la classe de test pour vérifier que la classe générique fonctionne toujours avec ce type donné.

La qualité par les tests

Nous allons clore ce chapitre sur une note plus théorique. Pour de nombreuses personnes, il est évident que les tests automatisés tels que les recommande XP ne sauraient suffire à garantir qu'un programme est correct.

Ce point nous paraît tout à fait juste. Pour autant, nous pensons, de façon pragmatique, qu'aucune autre des démarches couramment utilisées pour la vérification des programmes n'arrive à la hauteur des tests pratiqués par XP en ce domaine : les tests «manuels» pratiqués en fin de projet apportent un niveau de fiabilité notoirement insuffisant et les méthodes formelles sont rarement applicables à des projets mettant en œuvre des langages ou des technologies en usage courant.

À nos yeux, les tests unitaires, tout comme les tests de recette au niveau d'abstraction supérieur, n'ont pas tant pour objectif de garantir qu'un programme est en tout point conforme aux attentes que d'amener programmeurs et clients à mener un projet en se concentrant sur leur objectif, et d'éviter ce qu'on pourrait appeler la «programmation par coïncidence» : quand le

programme fonctionne, on ne sait pas pourquoi ni comment il fonctionne – on saura encore moins pourquoi il ne fonctionne pas !

Un programmeur XP sait toujours pourquoi et comment son code produit les résultats qu'on en attend ; il pratique la «programmation par intention». Cette attitude se manifeste en particulier dans la façon dont les programmeurs XP traitent les «bogues».

Du statut des défauts logiciels en XP

La critique du terme «bogue» tel qu'il est employé par la plupart des programmeurs, plus tard reprise par Bertrand Meyer ou encore par Ron Jeffries, remonte à E. W. Dijkstra : «La métaphore animiste de l'insecte [en anglais, *bug*], qui s'est malicieusement introduit pendant que le programmeur avait le dos tourné, est intellectuellement malhonnête puisqu'elle tend à cacher qu'il s'agit d'une erreur dont le programmeur est lui-même à l'origine. On appréciera ce que ce simple changement de vocabulaire peut avoir d'effets en profondeur : auparavant, un programme qui ne contenait qu'un seul bogue était 'presque correct' ; alors qu'après ce changement, un programme qui contient encore une erreur est tout simplement 'erroné'. »

La «malhonnêteté intellectuelle» dénoncée ici peut s'expliquer par les relations parfois malsaines qui s'établissent entre clients et techniciens, chacun cherchant à rejeter la faute sur l'autre lorsqu'un programme se révèle non conforme aux attentes.

Dans une équipe XP, on parlera rarement de «bogue». La qualité accrue apportée par les pratiques de programmation permet un certain détachement des intervenants techniques : les «bogues» n'étant plus un élément constant de leur quotidien mais des événements rares et isolés, ils font l'objet d'une analyse plus attentive.

Terminologie relative aux défauts logiciels

Dans ce contexte, on parlera plus volontiers d'un *incident*, qui se manifestera (par définition) par un «rapport d'incident». On n'emploiera pas les termes d'incident technique, qui laisseraient penser que les intervenants techniques sont nécessairement responsables : un «incident», dans ce sens, peut simplement résulter d'une incompréhension de la part du client.

Même lorsque c'est le cas, cette identification fait partie d'une tâche systématique qui est la *localisation* de l'origine de l'incident : insuffisance de la documentation utilisateur, ou une *défaillance*, qui peut être une défaillance des opérations d'installation ou d'exploitation, ou (plus souvent) une défaillance du programme.

Une défaillance peut généralement (mais pas nécessairement) s'expliquer par un *défaut* du programme, lequel est systématiquement lié à une *erreur* de la part d'un programmeur. La correction du défaut constitue une *résolution* de l'incident : elle peut avoir pour effet de dissiper une ou plusieurs causes de défaillance.

Cette terminologie peut paraître trop précise, voire précieuse. Toutes les équipes XP ne l'utiliseront pas nécessairement, mais elles feront nécessairement preuve d'une précision équivalente dans leurs actions correctives.

Lorsqu'un incident est rapporté, qui laisse penser qu'un défaut du programme en est l'origine, l'équipe XP accueillera cette information non comme un reproche, mais comme l'occasion d'améliorer les procédures de tests unitaires qu'il a mises en place. En effet, si un défaut lui a échappé, c'est que ses tests unitaires étaient à ce jour insuffisants, et qu'il convient de renforcer leur vigilance.

Ainsi, lorsqu'ils ont identifié un défaut dans leur programme, les programmeurs XP ont pour premier réflexe, *avant* la correction de ce défaut, d'écrire un nouveau test unitaire, écrit de telle sorte qu'il échoue *parce que* le défaut constaté est présent. (Si le test ainsi écrit n'échoue pas, c'est que l'hypothèse formulée par le programmeur sur le défaut n'est pas la bonne, et donc qu'il doit chercher encore.) Lorsque le défaut est corrigé, le test doit immédiatement passer.

Une équipe XP sait donc toujours, lorsqu'un programme ne fonctionne pas, pourquoi et comment il ne fonctionne pas ; et, à l'inverse, elle saura toujours exactement comment et pourquoi ses programmes fonctionnent.

5

Pratiques collaboratives

La stratégie consistant à « diviser pour régner » s'avère toujours redoutable contre l'ennemi. Pour le manager, la sagesse consiste à ne pas s'en servir volontairement contre ses propres équipes.

– Don Wells

Une approche fondée sur le travail d'équipe

L'Extreme Programming rompt avec l'organisation traditionnelle des équipes de développement, selon laquelle les développeurs travaillent seuls sur des tâches et des parties distinctes de l'application. En effet, le modèle XP est pour l'essentiel basé sur la coopération et la communication : c'est dans la création d'une réelle dynamique d'équipe qu'XP recherche les performances.

Dans la pratique, l'organisation d'une équipe XP est caractérisée par les deux éléments suivants :

- Les membres de l'équipe collaborent et interagissent en permanence ; ils ne travaillent seuls qu'exceptionnellement.
- L'application n'est pas découpée en zones réservées à tel ou tel développeur, mais forme au contraire un tout sur lequel chacun sera habilité à intervenir en fonction des besoins.

Comme ils travaillent toujours ensemble, les membres d'une équipe XP apprennent à se connaître pour opérer ensemble efficacement. L'information circule, les compétences s'échangent ; les problèmes rencontrés lors du développement sont compris puis résolus par l'ensemble de l'équipe.

Chacun étant amené à travailler sur toutes les parties qui la composent, la connaissance de l'application est mieux distribuée dans l'équipe. L'intervention de plusieurs personnes sur le code (relecture, corrections et optimisations) en améliore la qualité et la fiabilité. En outre, la gestion du projet est d'autant facilitée que les développeurs sont capables d'intervenir sur toutes les parties de l'application.

XP tend donc idéalement à constituer une équipe soudée, flexible, et dont la capacité de résolution de problèmes est largement supérieure à celles que peuvent cumuler tous ses membres, pris individuellement. Or, cet idéal est plus facile à décrire qu'à mettre en place, car l'entente entre membres d'une équipe ne se commande pas. XP fixe cependant cette cohésion comme objectif et propose un ensemble de pratiques concrètes destinées à favoriser son émergence dans le projet. Nous verrons que ces pratiques se déclinent comme suit :

- **La métaphore** (*metaphor*) : les développeurs n'hésitent pas à recourir aux métaphores pour décrire la structure interne du logiciel ou ses enjeux fonctionnels, de façon à faciliter la communication et à assurer une certaine homogénéité de style dans l'ensemble de la conception – l'idéal consistant à décrire le système dans son ensemble par une métaphore unique.
- **La programmation en binôme** (*pair programming*) : les développeurs travaillent par groupes de deux, en changeant fréquemment de binôme de sorte que chacun soit amené à travailler avec tous les autres.
- **La responsabilité collective du code** (*collective code ownership*) : tous les membres de l'équipe connaissent l'application dans son ensemble, et chacun est responsable de toutes les parties de l'application et peut y intervenir lorsqu'il le juge nécessaire.
- **Les règles de codage** (*coding standards*) : les développeurs se plient aux règles de codage définies par l'équipe, pour assurer l'homogénéité de l'application et ainsi faciliter le travail collectif.
- **L'intégration continue** (*continuous integration*) : les développeurs livrent leur travail au reste de l'équipe plusieurs fois par jour, de manière à limiter les problèmes d'intégration et à disposer à tout moment d'une version opérationnelle du logiciel qui intègre les derniers développements en cours.

Dans les sections suivantes, nous allons détailler chacune de ces pratiques.

Rôle de la métaphore

Au sein des méthodes «classiques», on cherche généralement à communiquer ce que l'on attend du projet, et ce qu'on en a effectivement réalisé, au moyen de documents écrits – et généralement répondant à certains formalismes. C'est ainsi qu'on rédigera un cahier des charges détaillé, des diagrammes d'architecture ou des documents de conception selon la notation UML.

Ces documents peuvent refermer une quantité considérable d'information et nécessiter un effort de rédaction non négligeable. Ils jouent plusieurs rôles distincts, mais l'un en particulier est crucial : ils permettent aux programmeurs, mais aussi bien aux clients ou aux gérants du projet, de se référer à une vision d'ensemble commune du système. Cette vue d'ensemble permet par exemple de savoir où il convient d'intervenir pour ajouter tel ou tel type de fonctionnalité, quelles parties du système sont critiques ou au contraire moins importantes...

Cette vision synthétique est malheureusement souvent noyée dans une documentation trop détaillée et pas toujours à jour, qui nuit finalement à la maintenabilité des applications réalisées. Certes, la présence d'un cahier des charges peut jouer un autre rôle – préciser les besoins ou conditionner l'acceptation des livraisons par le client ; de même, un diagramme d'architecture peut orienter les choix techniques, un diagramme de conception peut donner des garanties quant à la stabilité du système. Mais ces fonctions, nous l'avons vu dans les chapitres précédents, sont assurées dans XP par d'autres éléments.

Rappelons que, dans XP, on cherche à minimiser les activités qui ne contribuent pas efficacement à l'objectif premier – fournir un logiciel qui fonctionne. Si nous avons de bonnes raisons de considérer la plus grande partie des documents usuels comme redondante, nous ne rédigerons que les quelques pages dont il n'est pas possible de se passer – celles qui fournissent une représentation d'ensemble. Dans le meilleur des cas, cette description prendra la forme d'une métaphore.

En général, il s'agit de décrire le logiciel attendu – ou réalisé – en des termes plutôt imagés, évocateurs de sa structure ou de ses parties les plus importantes, et d'éviter un jargon technique stérile. À chaque occasion, lorsqu'on parle du programme ou de sa conception, les membres d'une équipe XP s'efforcent consciemment de trouver des images plutôt que de parler en termes techniques : «Cette partie du système fonctionne comme une autoroute, les messages qui circulent dans un sens n'ont pas d'effet sur le traitement des messages qui circulent dans l'autre sens. »

Cette approche est efficace pour se faire comprendre ponctuellement, même si ces métaphores restent éphémères et n'interviennent plus dans les conversations ultérieures. Mieux encore, l'utilisation de ces métaphores que nous appellerons «locales» conduit à une flexibilité mentale et à un degré d'abstraction lors des conversations qui sont largement bénéfiques pour toutes les activités de conception, qu'il s'agisse de l'écriture des tests unitaires, du remaniement ou de l'analyse des scénarios client. Concrètement, la conséquence en est que l'on passe moins de temps à discuter de «ce qu'il est possible de faire dans le système» et plus de temps à découvrir «ce que le système doit faire».

L'objectif ultime mais rarement atteint consiste en une seule métaphore «globale» dont la seule évocation permet de fédérer les idées autour de toutes les problématiques de conception qui peuvent survenir.

Métaphores célèbres…

Des exemples évidents illustrent l'intérêt et l'efficacité qu'on peut attendre d'une description métaphorique d'un système. Les logiciels les plus connus, ceux qui sont omniprésents dans notre vie courante ou dans le vocabulaire technique de la profession, ont fait l'objet à la base d'une expression métaphorique : la toile d'araignée, un bureau avec des documents, des dossiers ou des classeurs... Certaines métaphores sont tellement bien intégrées et fondamentales qu'elles n'apparaissent plus comme telles – arbre, client-serveur, etc. On ne les conçoit plus que comme termes techniques alors que la seule «réalité» technique qu'ils recouvrent celle d'électrons circulant dans des circuits.

Notre XPérience

Dans la pratique, nous n'avons jamais vu se produire ce «déclic» qu'évoquent les vétérans d'XP que sont Beck, Jeffries ou Cunningham, comme sur le projet C3 où la métaphore utilisée – celle d'une chaîne de montage dans une usine – a parfaitement collé, de bout en bout, aux contraintes techniques, et a servi de catalyseur au travail de l'équipe. En revanche, on constate sans peine les effets néfastes que peut avoir sur un projet l'absence de métaphores, même «locales» ; au bout d'un certain temps, il devient pénible d'expliquer – et pire encore, de s'expliquer – ce que fait telle ou telle partie du programme, autrement qu'en détaillant minutieusement son fonctionnement. Bientôt, même cette explication n'est plus possible – la seule manière de comprendre ce que fait le code est de le lire, voire de le déchiffrer.

La programmation en binôme

Dans une équipe XP, les développeurs travaillent systématiquement à deux devant une même machine. L'un des développeurs est aux commandes, il joue le rôle de *pilote* et prend à son compte d'écrire le code proprement dit. Il s'occupe donc de la manipulation des outils de développement, de la navigation dans le code, des contraintes du langage, bref des aspects «tactiques» de la tâche, et en cela son rôle ressemble à celui du développeur solo.

L'autre développeur joue le rôle de *copilote* ; il est chargé des aspects stratégiques du développement en cours. Il a pour mission d'effectuer une relecture immédiate et continue du code écrit, de proposer d'autres solutions d'implémentation ou de design, d'imaginer de nouveaux tests, de penser aux impacts des modifications sur le reste de l'application, etc. Même s'il n'a pas les commandes, le copilote est loin d'être passif : il est responsable du code écrit au même titre que le pilote et leur dialogue est donc permanent pour définir les solutions d'implémentation ou de design à mettre en œuvre.

Remarque

La « relecture » pratiquée en XP comme partie intégrante du travail en binôme apporte la plupart des avantages généralement attribués aux « relectures de code » formelles… sans en avoir les inconvénients : une revue des travaux finis est souvent ennuyeuse, et parfois humiliante, pour les développeurs.

Ces deux rôles ne sont cependant qu'indicatifs. Dans la pratique, pour un observateur extérieur, ces deux développeurs paraîtront simplement entretenir un dialogue permanent, s'affairant à réaliser ensemble la tâche en cours. Le seul élément qui différencie réellement ces deux rôles est le contrôle du clavier, mais celui-ci peut changer de mains à tout moment, les rôles étant alors inversés.

Règles de formation des binômes

Les binômes changent fréquemment – jusqu'à plusieurs fois par jour – de telle sorte que chaque membre de l'équipe soit amené à travailler avec tous les autres.

La sélection des binômes se fait de la propre initiative des développeurs. En général, celui qui commence à travailler sur une tâche demande simplement l'aide de quelqu'un d'autre ; ce dernier ne peut la lui refuser, ou seulement temporairement s'il est occupé sur une autre tâche.

Le choix d'un partenaire se fait sur des critères variés :

- Lorsque le développeur ne connaît pas bien certains aspects fonctionnels/techniques de la tâche, il peut s'associer avec quelqu'un qui les connaît pour profiter de son expérience.
- À l'inverse, si le développeur connaît bien certains aspects fonctionnels/techniques de la tâche, il peut s'associer avec quelqu'un qui ne les connaît pas pour les lui faire découvrir.
- Le développeur peut également s'associer avec un développeur intéressé par la tâche en cours.
- À défaut, le développeur peut simplement s'associer à un autre développeur libre, la règle étant bien sûr d'éviter que deux développeurs ne travaillent seuls.

Répartition individuelle des tâches

À la différence de la programmation qui s'effectue en binôme, la responsabilité du suivi des tâches n'est pas partagée : chaque développeur est seul responsable du suivi de ses propres tâches. Cela signifie qu'à un instant donné, un développeur est soit en train de réaliser l'une de ses tâches, soit en train d'aider un autre développeur à réaliser les siennes. C'est aux développeurs eux-mêmes de s'organiser et de faire en sorte qu'au final, chacun ait pu mener à bien la réalisation de l'ensemble de ses tâches. Ils disposent pour cela d'une grande latitude,

sachant que les tâches peuvent être réalisées en plusieurs fois et qu'un binôme ne reste pas nécessairement figé pendant toute la durée d'une tâche[1].

La gestion du projet doit tenir compte de cet élément et laisser à chaque développeur le temps à la fois d'aider les autres développeurs et d'assurer la réalisation de ses propres tâches. Cette organisation repose sur une identification et un suivi précis des tâches à réaliser, que nous aborderons en détail dans le chapitre 6 consacré à la gestion de projet.

Une pratique rentable

La programmation en binôme peut soulever au premier abord la question de la rentabilité. Après tout, chaque tâche ne risque-t-elle pas de coûter deux fois plus cher que si elle était réalisée par un développeur seul ?

L'expérience des projets XP et les résultats des premières études sur le sujet tendent à prouver que cette pratique peut au contraire se révéler plus rentable que le développement en solo. Il est cependant difficile d'en fournir une preuve quantifiée et irréfutable, et ce pour deux raisons principales. D'une part, il est difficile, voire impossible, d'établir une métrique de la productivité d'une équipe sur un projet complet. D'autre part, les bénéfices de cette pratique sont pour la plupart liés à des phénomènes difficilement mesurables : facilité de modification et d'extension du code, diminution du nombre de régressions provoquées par ces modifications, répartition des connaissances dans l'équipe, motivation des développeurs pour rester sur le projet, etc. Ces éléments ont tous des effets significatifs à moyen ou long terme, et s'ils n'interviennent pas directement (ou du moins pas visiblement) dans le coût du projet lui-même, ils peuvent avoir un impact très important sur le service ou l'entreprise concernés.

> **Remarque**
> La rentabilité économique du travail en binôme est d'autant plus difficile à évaluer qu'elle ne peut être aisément décorrélée de celle des autres pratiques XP. Nous vous renvoyons au chapitre 8 consacré aux aspects économiques d'XP pour plus d'informations sur ce sujet.

Nous avons pu cependant constater en pratique le bien-fondé de cette démarche, et surtout sa viabilité économique qui repose sur plusieurs facteurs :

- Un binôme est beaucoup plus rapide sur une tâche donnée qu'un programmeur seul – sans toutefois aller jusqu'à être deux fois plus rapide. Le surcoût induit par l'affectation d'une seconde personne sur une tâche est donc limité.

1. Il faut en revanche éviter de faire deux tâches en même temps, par exemple profiter de ce qu'on est en train de modifier un fichier pour faire une correction qui intéresse une autre tâche. On peut changer fréquemment de « casquette », mais il faut n'en porter qu'une seule à tout moment.

- La conception et le code sont meilleurs lorsqu'ils ont été produits et vérifiés par deux personnes. Cela implique des coûts de modification, d'évolutions et de maintenance réduits.
- La connaissance de l'application est mieux répartie dans l'équipe, ce qui accélère le développement et facilite l'allocation des tâches.
- L'équipe est plus cohésive et motivée.
- Les tâches les plus critiques peuvent être réalisées plus rapidement, offrant ainsi l'opportunité de livrer plus tôt une version réduite du logiciel.

Ces phénomènes sont détaillés dans les sections qui suivent.

Un surcoût immédiat limité

Comme nous l'avons vu, la programmation en binôme apporte en général des bénéfices à moyen ou long terme. Mais à quel prix ? Et qu'en est-il du surcoût sur le court terme ?

Dans leur article «The cost and benefits of pair programming»[1], Alistair Cockburn et Laurie Williams décrivent les résultats d'une série d'expériences dans laquelle des étudiants – seuls ou en binôme – menaient des tâches de développement identiques. En moyenne, les binômes complétaient leurs tâches dans un délai d'environ 40 % de temps plus court que les développeurs seuls. Autrement dit, la réalisation de deux tâches par un couple de développeurs ne nécessite qu'environ un cinquième de temps supplémentaire par rapport à la réalisation de chacune de ces tâches en solo – ce temps supplémentaire ne se répercutant que dans les coûts de main-d'œuvre (C1, voir chapitre 8, «Coûts et retour sur investissement»).

Ce chiffre, quoique issu d'une expérience contrôlée, est cohérent avec les retours d'expérience des équipes XP en milieu industriel, qui constatent en général que le gain de performance des binômes compense en grande partie le surcoût lié à la présence du copilote sur la tâche.

Le premier facteur de ce gain tient aux bénéfices d'une relecture de code immédiate et continue. Le copilote signale immédiatement des erreurs de frappe ou de syntaxe qui auraient été découvertes à la compilation, mais également des erreurs de logique qui n'auraient été découvertes que lors de l'exécution des tests. Le binôme économise ainsi des cycles de compilation et de test inutiles, et surtout le temps perdu à mener quelque investigation pour comprendre ces problèmes.

Par ailleurs, deux développeurs qui résolvent un problème ensemble explorent un champ de possibilités plus vaste que ne le ferait un développeur seul, et ils aboutissent en général à de meilleurs compromis. Cela se traduit par l'élaboration de solutions plus ingénieuses, qui en général requièrent moins de code et sont donc implémentées plus vite. Ce phénomène est

1. Voir *Extreme Programming Examined* de G. SUCCI et M. MARCHESI, Addison Wesley, 2001.

renforcé par le fait que le spectre de compétences du binôme est plus large. Le binôme a ainsi moins de chances d'être ralenti par un manque de connaissances techniques ou fonctionnelles.

Enfin, il y a un effet de stimulation réciproque des développeurs du binôme, qui fait que les deux peuvent rester concentrés plus longtemps sur leur tâche, sans être tentés par exemple de consulter leur messagerie personnelle ou de naviguer sur Internet.

Bien entendu, le gain peut varier largement d'un binôme à l'autre. Les deux développeurs dialoguent beaucoup, et leur performance dépend directement de leur capacité à aboutir rapidement à des compromis. Cette capacité s'améliore avec l'expérience, et les gains obtenus s'accroissent au fur et à mesure que les développeurs se familiarisent avec cette pratique et apprennent à mieux connaître leurs partenaires.

Mis bout à bout, ces éléments montrent que l'investissement direct consenti pour la mise en place de cette pratique est finalement assez peu élevé. Nous allons examiner dans les paragraphes suivants les aspects bénéfiques de cet investissement.

Partage des connaissances

Meilleure vue d'ensemble de l'application

Puisqu'ils terminent leurs tâches plus rapidement, les développeurs qui fonctionnent en binôme ont l'occasion de travailler sur davantage de tâches différentes et acquièrent ainsi une bonne vision d'ensemble de l'application. Leur travail est plus varié, et ils peuvent s'impliquer plus encore dans la vie générale du projet.

Cela représente également un atout important du point de vue de la gestion du projet : l'allocation des tâches est facilitée par le fait que plusieurs personnes ont les connaissances requises pour intervenir sur une tâche donnée, et il y a bien moins de risques que le projet se retrouve bloqué lorsqu'un développeur part en congés ou quitte l'équipe.

Par ailleurs, du point de vue de la réalisation proprement dite, la participation de plusieurs personnes dans la résolution des problèmes rencontrés favorise, par la diversité des points de vue récoltés, la recherche de solutions «propres» et économiques.

Notre XPérience

Nous avons pu constater en pratique que ce dernier point a une portée significative sur l'avancement du projet. Lorsqu'un binôme bute sur un problème ou se trouve confronté à un choix difficile à opérer, il s'adresse aux autres binômes, sachant qu'ils ont tous une connaissance minimale du contexte. Le problème est alors résolu de manière collective : chaque membre de l'équipe a ainsi eu l'occasion d'apporter ses idées et ses compétences, et a ensuite connaissance de la solution retenue et des raisons qui y ont conduit.

Échange de connaissances techniques et fonctionnelles

Outre la connaissance spécifique à l'application, les développeurs ont également l'opportunité de se transmettre des connaissances générales sur leur métier, que ce soit sur le plan du langage de programmation, des techniques de conception, de l'environnement de développement ou encore sur les aspects fonctionnels du projet. Ces éléments font de l'environnement XP un environnement particulièrement propice à la formation et au développement personnel : les développeurs débutants ou expérimentés y trouveront l'occasion de compléter leurs connaissances par la pratique, et les débats des séances collectives de résolution de problèmes permettront de faire avancer l'équipe entière.

Amélioration de la qualité du code

Nous avons mentionné plus haut les bénéfices des relectures de code continues et des compétences élargies du binôme par rapport au développeur solo, mais uniquement dans une perspective de performance immédiate. Or, la programmation en binôme joue également un rôle important dans la qualité du code produit, et il convient de rappeler ici qu'un projet XP fonde son ouverture au changement sur le fait que le code soit à même d'accueillir des extensions et des modifications.

Cette qualité se traduit en général par :

- une plus grande clarté : le copilote pousse le pilote à respecter les règles de codage de l'équipe et il l'aide à trouver de meilleurs noms pour exprimer le sens du code qu'il écrit ;
- une meilleure conception : puisque le binôme a une connaissance plus large de l'application, il peut penser à des factorisations plus nombreuses et mieux respecter ainsi le principe du *Once and Only Once*. Dans l'étude précédente, cela se traduisait par une diminution de 10 % à 25 % du nombre de lignes dans les programmes écrits par les binômes ;
- moins d'erreurs : le copilote peut déceler à la lecture du code des erreurs de logique qui n'auraient pas été révélées par des tests élémentaires. À titre indicatif, dans l'étude citée plus haut, le code des binômes comportait en moyenne 15 % de défauts de moins que celui des développeurs solo ;
- un meilleur respect des règles de codage : si un programmeur ne connaît pas ou a oublié telle ou telle règle de codage, son binôme la lui rappelle. Un meilleur respect des règles de codage facilite ultérieurement le partage du code et diminue le nombre de défauts dans le cas de règles qui sont fixées pour éviter les pièges du langage, par exemple.

Satisfaction et cohésion de l'équipe

Terminologie

Nous désignons ici par *cohésion* l'aptitude de l'équipe à s'organiser pour exploiter au mieux les capacités de chacun de ses membres, en exploitant leurs complémentarités et en résolvant les divergences d'intérêts.

Dans les équipes organisées sur un modèle de séparation des tâches, chaque développeur est souvent confronté seul à ses problèmes, car les autres membres de l'équipe ne connaissent pas suffisamment le contexte précis dans lequel il travaille pour pouvoir l'aider. Le problème est particulièrement manifeste à l'approche des livraisons ou lors de certaines phases de maintenance : au sein d'une même équipe, certains développeurs sont bien plus sollicités que d'autres.

La programmation en binôme permet de sortir de ce schéma, en créant une situation dans laquelle chaque développeur sait qu'il peut s'appuyer sur le reste de l'équipe.

Tout d'abord, le risque de se trouver confronté seul à une tâche difficile disparaît puisque les développeurs travaillent par deux. Mais même le binôme n'est pas isolé puisque potentiellement toute l'équipe est capable d'intervenir sur un problème donné. Ainsi, si un problème difficile est décelé dans l'application, l'équipe au complet peut se réunir et le solutionner rapidement plutôt que de laisser un développeur dans l'embarras.

De plus, l'interaction permanente d'une équipe XP induit un environnement nettement plus vivant que celui des développeurs en solo, réduits à une concentration silencieuse. Les développeurs apprennent à se connaître mutuellement sur le plan professionnel et peuvent se découvrir sur un plan plus personnel. La bonne entente des membres de l'équipe facilite nettement la résolution des problèmes humains propres au travail en groupe.

La description qui est faite ici peut sembler idyllique, et certaines équipes XP ne manqueront pas de rester sujettes à des tensions ou des antagonismes. La cohésion de l'équipe dépend bien sûr en premier lieu de la personnalité de chacun de ses membres, mais XP crée avec cette pratique des conditions particulièrement propices à l'épanouissement et à la motivation de l'équipe. Du point de vue du projet, cette motivation garantit un certain investissement dans le travail et la pérennité d'une équipe à moyen ou long terme ; il suffit de considérer un instant le coût du départ d'un développeur en termes de perte de connaissances, de recherche de candidats, puis de formation du nouveau venu, pour apprécier l'avantage que cela représente.

Opportunités en matière de gestion de projet

Comme nous l'avons vu, la répartition de la connaissance dans l'équipe facilite la distribution des tâches et permet ainsi de mieux répartir la charge de travail. Mais outre cette optimisation, le travail en binôme offre des avantages en termes de délais de livraison du produit.

Certes, les binômes d'une équipe XP ne peuvent travailler à un moment donné que sur la moitié des tâches que traiterait une équipe de développeurs solo. Mais si l'on reprend les chiffres de l'étude citée plus haut, on peut considérer que l'équipe XP terminera ces tâches au bout de 60 % du temps total de réalisation des tâches. Si ces dernières ont été correctement choisies en fonction de leur priorité, cela signifie que l'équipe XP peut livrer plus tôt une première version du logiciel qui inclut les tâches les plus importantes. Cet élément peut être déterminant dans des contextes où les délais de livraison sont primordiaux.

Apprendre à travailler en binôme

Les bénéfices du travail en binôme ne sont pas nécessairement perceptibles tout de suite. Il faut du temps – de quelques heures à quelques semaines – pour que les binômes apprennent à se connaître et à travailler efficacement ensemble, et cet apprentissage requiert des développeurs eux-mêmes et du coach une attention particulière.

Notre XPérience

L'évolution est tout à fait sensible au cours du déroulement du projet. Au départ, les binômes passent du temps à chercher un terrain d'entente sur des questions de design, d'idiomes d'implémentation ou de règles de codage. Chaque programmeur fait des efforts de diplomatie lorsqu'il relève les erreurs de son partenaire ou qu'il défend son point de vue dans un choix d'implémentation ou de conception. Cependant, avec le temps, les binômes trouvent des points d'accord qui leur permettent de converger plus rapidement vers des solutions communes. Ils se parlent aussi plus librement, ayant appris à mieux se connaître. C'est à partir de ce moment que les bénéfices de la programmation en binôme se font sentir et que l'équipe commence réellement à prendre de la vitesse.

Dans les paragraphes suivants, nous allons explorer quelques pistes visant à faciliter la mise en place de cette pratique et à gérer les difficultés qu'elle soulève. Nous allons voir en effet :

- comment introduire cette pratique dans l'équipe ;
- toute l'importance de l'effort d'ouverture consenti par chaque développeur ;
- comment maintenir le rythme ;
- comment gérer les différences de niveau ;
- comment exploiter les différences de points de vue ;
- les impacts de cette pratique sur l'embauche de nouveaux membres ;
- la façon d'introduire cette pratique dans des projets en cours.

Partir sur des explications claires

Lors de la mise en place de cette pratique, le coach doit fournir un certain nombre d'explications.

Il faut déjà bien faire comprendre que la programmation en binôme ne se réduit pas à une situation du type «un qui code, l'autre qui surveille», mais qu'il s'agit bien d'une collaboration active et continue entre les deux membres du binôme.

Il faut également expliquer en quoi cette pratique est souhaitable pour le projet, mais aussi en quoi elle l'est pour chaque membre de l'équipe. Rappelons ici les bénéfices que peuvent en attendre les développeurs eux-mêmes :

- l'opportunité d'apprendre les uns des autres, que ce soit au niveau fonctionnel ou technique ;
- l'opportunité d'entreprendre des tâches pour lesquelles ils n'ont pas toutes les compétences requises, sachant qu'ils pourront demander l'aide d'un développeur qui possède ces compétences ;
- l'opportunité de participer à de nombreuses tâches, puisqu'elles sont réalisées plus vite et que les binômes changent fréquemment ;
- une certaine sérénité lors de la réalisation des tâches, car quelqu'un d'autre aura relu tout le code et approuvé les choix de design avant que le code ne soit livré ;
- la garantie de ne pas se retrouver dans une situation où l'on travaille jour et nuit tandis que le reste de l'équipe se tourne les pouces.

Il y a bien sûr fort peu de chances pour que ces arguments convainquent d'emblée l'ensemble de l'équipe. Certains seront immédiatement enthousiastes, d'autres resteront sceptiques ou réfractaires pour diverses raisons.

Certains développeurs préfèrent par exemple avoir une autonomie complète sur une partie de l'application qui leur est réservée. Ils ont leurs propres habitudes de développement et ne souhaitent pas devoir négocier avec un partenaire la mise en place de telle ou telle solution, ou présenter leur code de telle ou telle façon.

On trouve également des cas d'incompatibilités d'humeur caractérisées entre certains membres de l'équipe ; il arrive aussi que des développeurs expérimentés refusent de travailler avec des débutants de peur d'être ralentis.

Ces cas réclament une certaine fermeté de la part du coach, qui doit alors se faire l'avocat du projet et rappeler les objectifs visés par la mise en place de cette pratique. Il doit faire comprendre aux développeurs qu'il n'est pas admissible de laisser certaines parties de l'application sous le contrôle absolu d'un seul développeur. En effet, que se passera-t-il lorsque celui-ci partira en vacances, voire quittera le projet ou tombera malade ?

Les blocages complets sont de toute manière assez rares, et la plupart des développeurs sceptiques deviennent enthousiastes après quelques semaines de pratique. Pour les cas

«désespérés», il peut être envisagé d'effectuer des permutations avec des membres d'autres équipes, sachant que l'individu réfractaire risque fort d'être laissé de côté à mesure que le reste de l'équipe va se souder.

Un effort personnel d'ouverture

Le succès de la programmation en binôme repose pour l'essentiel sur les qualités humaines des membres de l'équipe et son efficacité ne pourra être complète que lorsque les développeurs feront preuve d'esprit d'ouverture et d'honnêteté les uns envers les autres. Cela peut nécessiter un effort de la part de chacun pour que soit levé un certain nombre de freins :

- Le copilote doit oser signaler les erreurs commises par le pilote, et ce dernier doit naturellement en convenir. Cela se passe sans problème lorsque les développeurs se connaissent bien, mais exige davantage de diplomatie lorsque ce n'est pas le cas.
- Chacun doit accepter de discuter chaque décision d'implémentation ou de design, et donc parfois renoncer à des préférences ou des habitudes personnelles. Cela implique également d'apprendre à réfléchir à plusieurs, et en particulier d'oser proposer des solutions en acceptant qu'elles puissent être rejetées ou paraître ridicules.
- Chacun doit accepter qu'un autre développeur découvre ses propres limites ou lacunes. Ce dernier point est souvent le plus difficile à dépasser au début, puisque le développeur se met en position de «faiblesse» vis-à-vis de son partenaire. Cela ne peut réellement fonctionner que si la forme d'honnêteté que ce type de collaboration engage est réciproque. Une fois que ce blocage aura disparu, chacun pourra alors très vite compléter ses connaissances.

Maintenir le rythme

Tout l'art de la programmation en binôme consiste à s'assurer que les deux développeurs restent concentrés et participent activement à la tâche en cours. Le risque de «décrochage» concerne plus particulièrement le copilote, qui ne dispose pas des commandes et se trouve donc dans une situation par nature plus passive.

Le problème ne se présente pas lorsque les deux développeurs décident ensemble du design ou des choix d'implémentation, mais plutôt lors de certaines périodes d'écriture du code proprement dit, lorsque les deux développeurs se sont mis d'accord sur la marche à suivre et que le copilote se contente de suivre le travail de son partenaire.

Dans ce genre de situation, le pilote doit s'assurer que son partenaire suit tout ce qu'il est en train de faire. S'il n'entend plus d'acquiescements au bout de quelques minutes, il doit s'arrêter et aider le copilote à reprendre le fil de la tâche. Au besoin, le pilote peut «penser tout haut» pendant qu'il effectue les manipulations dans le code, pour aider le copilote à rester concentré. Si tout cela ne fonctionne pas et que le partenaire n'arrive pas à suivre, le mieux est encore d'inverser les rôles.

La programmation en binôme est une activité vivante, chacun doit se sentir concentré et avoir le sentiment d'avancer vite. Si ce n'est pas le cas, il est important que l'équipe en parle et trouve des solutions pour améliorer la mise en œuvre de cette pratique.

Gérer les différences de niveau

Le travail en binôme est parfois délicat lorsqu'il existe une forte différence de niveau entre les développeurs :

- Lorsque le débutant est aux commandes, le développeur expérimenté risque de s'impatienter ou de décrocher parce que le rythme n'est pas suffisamment élevé.
- Lorsque le développeur expérimenté est aux commandes, le débutant risque de décrocher parce qu'il n'arrive pas à suivre le travail de son partenaire.
- Si la situation n'est pas facile à gérer, cela ne signifie pas pour autant qu'il faille s'interdire certaines configurations de binômes. Une équipe XP investit dans la prise en charge de débutants et fait le nécessaire pour qu'ils deviennent des acteurs de l'équipe à part entière. Cela est surtout vrai lorsque le projet s'étale sur plusieurs mois et que les débutants ont réellement le temps de progresser suffisamment.

Pour avoir rencontré cette situation en pratique, voici l'approche que nous préconisons :

- Laisser le débutant aux commandes autant que possible, pour qu'il puisse se familiariser avec les outils et s'assurer aussi que le développeur expérimenté ne terminera pas seul la tâche. Il faut, surtout au début, lui montrer toutes les astuces et les raccourcis d'utilisation des outils, pour que les manipulations et la navigation dans le code ne soient plus un frein.
- Le développeur expérimenté doit guider le débutant en permanence, en lui expliquant les raisons qui motivent ses choix d'implémentation ou de design.

Le développeur débutant est ainsi fortement stimulé, constamment tiré en avant par son partenaire. Le rythme pourra certes lui paraître élevé, mais il l'acceptera volontiers s'il est conscient des efforts consentis par le développeur expérimenté et de l'investissement que cela représente du point de vue du projet. La contrepartie en termes de formation ne pourra de toute manière que lui être extrêmement bénéfique.

> **Remarque**
> Les qualités humaines des développeurs se révèlent ici encore primordiales, qu'ils soient débutants ou expérimentés. Ces conditions *sine qua non* de succès d'un projet XP seront traitées plus en détail dans le chapitre 7.

Exploiter les différences de points de vue

Nous avons pu constater qu'au début du projet de nombreuses discussions portent sur des choix techniques ou sur des styles de programmation. Si la plupart de ces discussions conver-

gent rapidement par la prise en compte d'arguments irréfutables, il reste toujours quelques points délicats où les préférences personnelles l'emportent sur les critères objectifs.

Un sujet célèbre de polémique insoluble est celui, pour les développeurs Unix, du choix entre Emacs et Vi ; mais différents sujets techniques peuvent s'y prêter, tels que «typage statique contre typage dynamique», des questions relatives aux règles de codage, des discussions sur la portée et la granularité des tests unitaires...

Dans ce genre de situation, le binôme peut s'en remettre à l'arbitrage du reste de l'équipe ; cette approche est souvent la plus efficace. Nous conseillons toutefois de laisser certaines de ces discussions se poursuivre – sans bien sûr en abuser – parce qu'elles sont l'occasion d'apprendre à se connaître et à opérer ensemble. L'efficacité de l'équipe sera en effet conditionnée par la capacité de chacun de ses membres à participer à un débat de manière constructive, c'est-à-dire en sachant équilibrer *discussion* et *dialogue*.

Une discussion se définit comme un débat contradictoire, dans lequel chacun cherche à faire valoir son point de vue en trouvant des failles dans le raisonnement des autres. Une discussion se termine avec un gagnant et un perdant, ou encore par un *ex aequo* si chacun campe sur ses positions. C'est davantage un processus de *sélection* d'idées.

Un dialogue se définit comme une conversation, un échange de vues entre deux personnes, ou encore comme une discussion visant à trouver un terrain d'entente. Chacun propose des idées et attend des autres qu'ils l'aident à les évaluer et les compléter ; c'est davantage un processus de *génération* d'idées. Ce type de débat n'est possible que lorsque les développeurs font preuve des qualités d'ouverture et d'honnêteté que nous avons évoquées plus haut.

Trouver un équilibre entre discussion et dialogue, c'est simplement savoir explorer de nombreuses pistes puis s'accorder à n'en choisir qu'une seule.

Dans la pratique, il n'est pas toujours aisé d'y parvenir. Si les développeurs ont tendance à favoriser la discussion, ils risquent de choisir une solution, ou de rester bloqués sur un désaccord, sans avoir exploré suffisamment de solutions. Inversement, s'ils ont tendance à favoriser le dialogue, ils risquent de partir dans un débat sans fin dont rien ne ressortira. Le coach peut apporter son aide en jouant un rôle de modérateur, en demandant aux développeurs expansifs de laisser les autres s'exprimer et en aidant les développeurs introvertis à participer au débat.

D'un point de vue purement pragmatique, tout cela n'a d'autre objet que d'améliorer la capacité de l'équipe à résoudre les problèmes d'une manière collective, que ce soit en binôme ou bien avec l'équipe tout entière. Et si les binômes trouvent des solutions plus ingénieuses aux problèmes qu'ils rencontrent que les développeurs solo, nous avons constaté en pratique que l'équipe complète peut disposer d'un pouvoir de résolution de problèmes sans commune mesure avec celle du développeur solo. Cet avantage peut devenir considérable lorsqu'il est question de résoudre des problèmes techniques difficiles dans un temps limité, par exemple lors de la découverte par le client d'un défaut majeur du logiciel, ou encore lorsqu'il s'agit de trouver une solution économique pour l'implémentation d'une fonctionnalité donnée.

Les séances de conception collective

Le binôme sait s'ouvrir sur le reste de l'équipe : on en trouve une illustration dans les séances de conception collectives, qui font intervenir tous les développeurs. Ces séances ne sont pas planifiées ou organisées de manière formelle : elles ont lieu dès lors qu'un binôme est confronté à un difficile problème de conception ou qui pourrait avoir une portée large dans l'application. De nombreuses équipes XP utilisent la technique des «cartes CRC» au cours de ces séances.

Les cartes CRC

Il s'agit de fiches cartonnées du même type que celles des scénarios client, mais utilisées cette fois-ci pour représenter les différentes classes du système. On les appelle ainsi parce que les développeurs y consignent le nom de la Classe, ses Responsabilités et ses Collaborations (les classes dont elle dépend). Les développeurs peuvent ainsi manipuler les cartes, se les échanger, les agencer sur une table pour illustrer certaines relations, parler de leur conception en termes qui restent généraux… Cette technique permet d'animer de manière vivante et efficace une séance de conception collective, et au terme de la séance les cartes représentent une documentation synthétique des principaux éléments de conception qui ont émergé.

L'embauche

À partir du moment où la cohésion de l'équipe devient un objectif pour le projet, le processus d'embauche doit être adapté de façon à évaluer la capacité d'intégration des candidats à l'équipe.

L'approche préconisée par XP est la suivante : après avoir évalué le candidat de manière «classique», le coach organise une courte entrevue avec l'ensemble de l'équipe. Le coach n'est pas nécessairement présent lors de cette entrevue et les sujets abordés ne sont pas nécessairement liés au travail. L'objectif consiste simplement à voir si «le courant passe» entre le candidat et l'équipe, afin de s'assurer qu'il ne sera pas rejeté par la suite.

100 % du temps de travail en binôme ?

En pratique, il est difficile de réaliser la totalité du travail en binôme, ne serait-ce que parce que l'équipe peut compter un nombre impair de développeurs. Il y a aussi les cas plus rares de tâches simplistes pour lesquelles il semble superflu de travailler à deux, par exemple certaines tâches de saisie.

Le pourcentage exact de temps passé en binôme n'a pas une importance décisive. Certaines équipes fonctionneront parfaitement en appliquant cette pratique pour moitié, d'autres y consacreront peut-être 95 % de leur temps. Comme pour chaque pratique XP, c'est à l'équipe de trouver l'équilibre qui lui convient en fonction de son contexte précis, sachant bien entendu que les bénéfices que l'on peut en attendre seront en rapport avec l'efficacité selon laquelle elle est appliquée.

Dans le cas où un développeur devrait travailler seul, il est conseillé de s'assurer que d'autres développeurs puissent rapidement prendre connaissance de son travail, et il faut également s'efforcer de renouveler fréquemment les binômes pour qu'il ne reste pas seul trop longtemps.

Introduire cette pratique sur des projets en cours

L'introduction de cette pratique dans un projet existant est un peu plus délicate dans la mesure où les développeurs doivent échanger davantage d'informations avant d'être réellement productifs. La manœuvre est cependant tout à fait réalisable sans que soit compromis le déroulement du projet, pour peu qu'elle soit abordée de manière progressive.

Dans un premier temps, les développeurs peuvent simplement s'associer occasionnellement pour résoudre des problèmes donnés. Ce sera l'occasion pour eux de se familiariser avec cette pratique et de découvrir progressivement les autres parties de l'application.

Le chef de projet peut également affecter deux développeurs à la réalisation d'une tâche précise, et ensuite reproduire cette expérience de plus en plus fréquemment.

Quelle que soit l'approche choisie, l'important est de s'assurer que la mise en place est faite de manière suffisamment progressive pour que les développeurs aient le temps de devenir compétents sur des zones plus larges de l'application. Cette phase transitoire devrait déjà apporter les bénéfices des relectures de code effectuées, et ce sera également l'occasion pour l'équipe de s'accorder sur des règles et des idiomes de programmation communs.

Comme la plupart des pratiques XP, la programmation en binôme n'est réellement efficace que dans un contexte réellement XP, les autres pratiques venant pallier les problèmes chroniques qu'elle pose. Par exemple, à partir du moment où chaque développeur doit pouvoir travailler avec tous les autres sans trop de contraintes, il devient souhaitable que chacun connaisse l'ensemble de l'application et que l'équipe adopte un modèle de responsabilité collective du code. Si chaque binôme peut travailler sur n'importe quelle partie de l'application, mieux vaut disposer d'une batterie de tests unitaires lorsqu'un binôme effectue des modifications dans des zones de l'application qu'il ne connaît pas parfaitement.

Ces éléments font que l'introduction du travail en binôme dans une équipe existante est difficilement dissociable d'une adoption plus large des pratiques XP. Ce thème est traité plus en détail au chapitre 7, consacré à la transition vers une démarche XP.

Aménagement de l'environnement de travail

La programmation en binôme a un impact sur l'environnement de travail de l'équipe, qu'il s'agisse de l'environnement logique (l'ensemble des outils de développement) ou bien de l'environnement physique (la disposition des bureaux).

Les outils de développement

Comme le binôme a une plus grande capacité de réflexion et de réaction qu'un développeur seul, il est aussi plus sensible aux temps de latence imposés par les outils de développement. En particulier, le copilote risquera davantage de décrocher si les manipulations du pilote sont trop lentes. Cela induit d'assez fortes contraintes sur l'environnement de développement.

Tout d'abord, la modification et l'exploration du code doivent être aussi rapides que possible. Si le pilote passe son temps à parcourir les répertoires du projet pour trouver telle ou telle portion de code, il y a peu de chances que le copilote reste concentré bien longtemps. L'équipe doit se doter d'outils adaptés, et surtout apprendre à les configurer et à les exploiter convenablement.

Les cycles de compilation et d'exécution des tests unitaires doivent également être aussi courts que possible. Une attention particulière doit être portée sur les temps de compilation, quitte à changer de compilateur ou à renforcer les machines. Les dépendances entre les modules de l'application doivent être réduites au strict minimum, grâce à une conception adéquate, pour limiter la quantité de code à compiler après chaque modification.

L'espace de travail

D'après Kent Beck, «la mise en place d'XP commence avec un tournevis». En effet, si le travail doit réellement être effectué en équipe, l'organisation des bureaux basée sur le travail individuel risque d'être rapidement inadaptée, et l'approche XP consiste dans ce cas à adapter l'environnement à l'équipe plutôt que le contraire.

Avec le développement des bureaux type «open space», la disposition la plus courante ressemble à celle présentée à la figure 5-1.

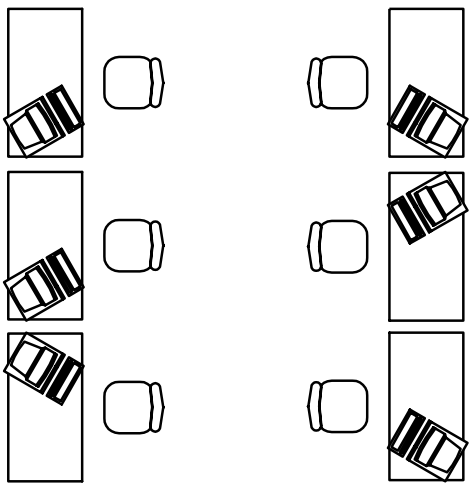

Figure 5-1. Agencement de bureau adapté au travail individuel

Dans ce genre de disposition, les développeurs se tournent tous le dos et font face individuellement à leur machine, l'objectif étant de favoriser leur concentration en les isolant de l'activité ambiante. Cette logique n'est bien entendu pas celle visée par XP, qui cherche au contraire à amener les développeurs à travailler ensemble et à participer en permanence à la vie de l'équipe.

Un environnement adapté à XP ressemble davantage à celui présenté à la figure 5-2 :

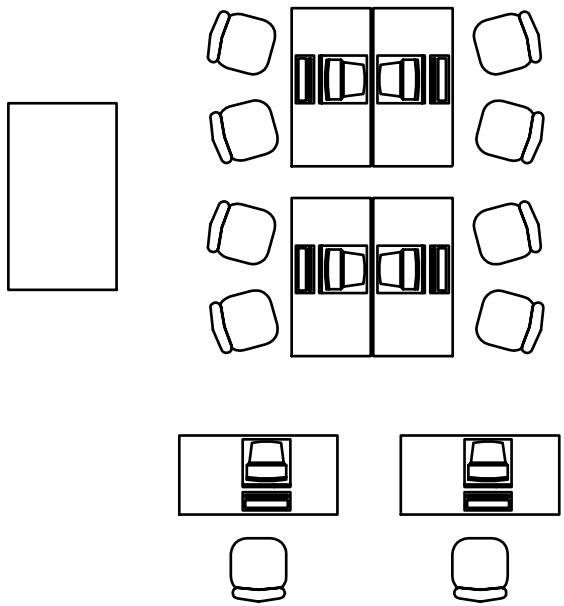

Figure 5-2. Agencement de bureau adapté au travail d'équipe

Cet environnement comprend divers éléments clés :

- Un espace central dédié au développement. Chaque bureau comporte deux chaises ; l'écran est posé au milieu pour que les deux membres du binôme puissent lire et se passer les commandes sans difficulté. Les bureaux se font face afin que tous les développeurs puissent communiquer sans être obligés de se déplacer.
- Un bureau pour les réunions, situé à proximité de l'espace de développement pour que les développeurs situés sur l'espace central puissent écouter et intervenir de temps à autre.
- Des bureaux isolés où les développeurs peuvent laisser leurs affaires, consulter leur messagerie ou téléphoner librement.
- Tout autour de cet espace de travail sont disposés un certain nombre d'affichages liés au projet : tableau de suivi des tâches en cours, statistiques des tests de recette, etc.

Ces éléments restent purement indicatifs, c'est à chaque équipe de définir puis d'améliorer continuellement son espace de travail en fonction des besoins et des contraintes qui lui sont propres.

Cette réorganisation des bureaux risque de se heurter à différents obstacles dans l'entreprise, notamment lorsque cette tâche est réservée à des services spécifiques plus ou moins indépendants du projet. Le coach et le manager doivent cependant œuvrer pour obtenir un maximum de souplesse, autant pour les aspects purement pratiques de ces adaptations que pour leur dimension symbolique. En effet, il s'agit là de transmettre à l'équipe le message selon lequel elle peut désormais prendre le contrôle de son environnement et de son processus de travail pour atteindre les objectifs qui lui sont fixés.

Responsabilité collective du code

Les modèles actuels

Le modèle de responsabilité du code le plus répandu aujourd'hui est celui de la responsabilité individuelle, dans lequel chaque développeur évolue dans une partie de l'application qui lui est réservée. Ce modèle suppose que les développeurs se mettent d'accord sur les interfaces exposées par leurs modules, et chacun peut ainsi travailler à ses propres tâches de manière autonome.

Cette approche se révèle pratique du point de vue de la gestion de configuration du logiciel, puisqu'elle limite en théorie les interférences entre développeurs aux seules interfaces de leurs modules. Elle peut également être appréciée des développeurs puisqu'elle leur garantit une certaine autonomie dans les décisions de conception et d'implémentation, chacun ayant le loisir d'organiser son code comme il l'entend.

La responsabilité individuelle est à double tranchant : d'un côté, elle a tendance à garantir qu'un travail est fait, car la personne s'en sent redevable. D'un autre côté, la notion de «responsable» au sens noble est rapidement entachée de «faute» en cas de problème, or cette attitude n'est pas constructive. Pourtant, la responsabilité individuelle est souvent exigée par des responsables qui veulent plus ou moins consciemment pouvoir se retourner contre quelqu'un.

Nous avons vu également que ce principe de séparation des tâches entraîne une segmentation des connaissances qui est nuisible tant du point de vue du projet en général que de celui de chaque développeur. Cette séparation pose également des problèmes sur le plan de l'application elle-même puisqu'elle augmente les risques de duplication de code et qu'en l'absence de relectures croisées, certains développeurs peuvent continuer à utiliser des techniques de codage maladroites qui entraînent des risques en termes de robustesse, de performances, de maintenance ou d'évolutions.

Ce mode de responsabilité du code n'est cependant pas le seul en vigueur aujourd'hui, certains projets pratiquant plutôt l'*absence* de responsabilité du code. Dans ce modèle, le code appartient à toute l'équipe et à personne à la fois, chacun pouvant intervenir sur l'application au gré des besoins sans que personne ne soit réellement garant de son intégrité globale.

Cette approche apporte une souplesse indéniable, mais elle tient souvent à une démarche de développement «opportuniste», dans laquelle les successions de modifications locales aboutissent progressivement à un design évoquant la fameuse métaphore du «plat de spaghettis», où toute nouvelle correction ou évolution se révèle à la fois coûteuse et risquée.

La responsabilité collective du code

Par rapport aux modèles que l'on vient d'évoquer, XP propose une solution originale sous la forme d'une *responsabilité collective* du code. Chaque binôme peut intervenir sur n'importe quelle partie de l'application dès lors qu'il le juge nécessaire, mais chacun est responsable de *l'ensemble* de l'application. En particulier, lorsqu'un binôme découvre qu'une partie de l'application n'est pas aussi claire ou simple qu'elle le devrait, son rôle va consister à améliorer le code en question pour garantir que l'application converge vers un tout cohérent.

> **Remarque**
> Afin de rendre cette notion de responsabilité collective acceptable aux yeux de certains responsables, préoccupés de ce qu'en cas de problème chacun pourrait s'en laver les mains, il est indispensable que l'équipe montre toujours qu'au contraire, on pourra s'adresser à n'importe qui en cas de problème, tous les développeurs étant compétents et disposés pour le traiter.

Ce mode de responsabilité est parfaitement adapté à la programmation en binôme. Il en reprend tous les avantages relatifs au meilleur partage des connaissances dans l'équipe, et renforce ceux qui ont trait à la relecture du code puisque celui-ci est relu non seulement par le copilote mais aussi par les autres binômes.

Pris isolément, ce modèle de responsabilité du code peut cependant souffrir de certaines limitations :

- Bien que tout le monde puisse intervenir sur l'ensemble de l'application, chaque développeur ne connaît parfaitement que le code qu'il a écrit. Un binôme qui travaille sur une partie qu'il n'a pas écrite risque de passer beaucoup de temps à la comprendre et d'y introduire des régressions.
- Plusieurs binômes peuvent travailler simultanément sur une même partie de l'application, et la fusion des modifications peut entraîner un surcoût en temps de gestion de configuration du logiciel.

Ce modèle se montre néanmoins viable dans un environnement XP où ces problèmes sont résolus par l'adoption de pratiques complémentaires.

D'une part, l'application est plus facile à comprendre grâce aux effets conjoints des pratiques de conception simple et de remaniement présentées au chapitre 3, et également par l'adoption d'un style de codage commun (ce dernier point est présenté en détail plus loin dans ce chapitre). La présence d'une batterie de tests unitaires permet par ailleurs de modifier avec confiance une partie de code que l'on n'a pas écrite, et dans tous les cas le binôme peut s'adresser au reste de l'équipe pour obtenir des informations ou de l'aide.

D'autre part, les problèmes de modifications simultanées de code sont réduits en raison de l'adoption d'une démarche d'*intégration continue.* Cette pratique spécifique est décrite plus en détail à la fin de ce chapitre.

Remarque

On notera toutefois que des projets dits «d'intégration», consolidant des technologies ou des progiciels hétérogènes, doivent être appréhendés avec précaution. En effet, le niveau de connaissance exigé pour intervenir sur les briques progicielles ou spécifiques peut impliquer une segmentation de cette responsabilité collective : il devient en effet difficile pour les membres de l'équipe d'avoir une vision globale et pertinente de l'ensemble du système.

Dans ce cas, en fonction du niveau de compétence de chacun, on pourra définir des sous-équipes susceptibles d'intervenir en modification sur chacune des parties. La responsabilité collective sera donc mise en œuvre, mais segmentée en fonction des compétences. Bien sûr, il faudra s'efforcer d'établir des «passerelles» entre ces sous-équipes, afin que la communication reste importante entre elles mais également pour donner l'occasion aux programmeurs de se former aux autres techniques et favoriser progressivement leur polyvalence.

Que deviennent les spécialistes ?

En général, les projets se dotent de spécialistes pour travailler sur des parties de l'application qui concernent des technologies spécifiques, par exemple la gestion de bases de données, les interfaces homme-machine, ou encore la communication inter-processus. Que deviennent ces spécialistes dans une équipe XP où chacun peut travailler sur toutes les parties de l'application ?

Tout dépend en fait de la façon dont cette spécialisation est exploitée. XP s'oppose au fait que les spécialistes s'attachent exclusivement aux parties de l'application relatives à leurs domaines de prédilection. Ils doivent s'intégrer au projet et participer à toutes sortes de tâches, au même titre que chaque développeur de l'équipe.

Cela ne signifie pas pour autant que leurs compétences spécifiques soient ignorées. Ces compétences sont précieuses lorsqu'il s'agit de réaliser des choix techniques spécifiques ou de résoudre des problèmes difficiles, et l'équipe sait en général vers qui se tourner lorsque ces situations se présentent. Si les spécialistes d'une équipe XP retrouvent une activité de développeurs «polyvalents», ils jouent donc néanmoins le rôle de consultants internes, chargés de

transmettre leurs connaissances aux autres membres de l'équipe et d'intervenir sur les questions pointues.

La responsabilité collective du code ne suppose donc pas que l'équipe est constituée de développeurs parfaitement uniformes. Elle suppose en revanche que les développeurs soient suffisamment polyvalents et qu'ils sachent mettre leurs connaissances en commun pour faire avancer ensemble le projet.

Remarque

Certains développeurs risquent de se montrer réfractaires à cette pratique, leur rôle de spécialiste leur réservant habituellement des tâches «nobles» de conception ou d'études en regard de leur spécialité. XP préconise une certaine fermeté à leur égard, et le coach aura là encore le devoir de faire comprendre au développeur l'intérêt de la démarche ou bien d'envisager son remplacement.

Règles de codage

Le modèle de responsabilité collective du code requiert une homogénéisation des styles de codage dans l'équipe, de sorte que chaque développeur puisse se sentir rapidement à son aise dans des parties de code qu'il n'a pas écrites. Pour cela, l'équipe définit au début du projet un ensemble de règles de codage qui sont ensuite scrupuleusement respectées par chaque développeur.

Ces règles portent sur des aspects divers de la programmation tels que la présentation du code, l'organisation des commentaires ou les règles de nommage.

L'introduction de cette pratique risque d'être mal perçue au premier abord car chaque développeur est attaché à son style personnel. L'expérience prouve cependant que l'adaptation à de nouvelles règles de codage est assez rapide, et cette pratique fait rapidement prendre conscience à chacun qu'il travaille réellement en équipe.

Remarque

Dans des entreprises qui intègrent plusieurs dizaines voire plusieurs centaines de développeurs, cette élaboration collective des règles de codage devient difficile. Si l'élaboration initiale peut se faire en ayant recours à ce mode de fonctionnement, il est souvent indispensable de figer ces règles (après quelques retours d'expérience) pour assurer leur diffusion auprès de l'ensemble des équipes.

Cette approche, qui s'inscrit dans une démarche qualité, permet alors aux équipes d'intervenir plus aisément sur plusieurs projets (notamment lors des phases de maintenance) en retrouvant des règles familières. Même dans ce cas, on constate que chaque équipe XP adoptera, *outre* les règles standards de l'entreprise, des règles supplémentaires spécifiques.

Un choix d'équipe

Il est important que chaque développeur participe à la définition des règles de codage, ce travail représentant en soi une première étape dans l'intégration de l'équipe.

Les règles doivent donc faire l'objet d'un consensus ; si, dans la pratique, on y parvient assez aisément pour la plupart d'entre elles, il en reste toujours certaines qui soulèvent des différences de point de vue plus délicates à résoudre. Dans tous les cas de blocage, l'équipe peut décider soit de trancher en votant ou en demandant l'intervention du coach ou d'un tiers, soit de laisser simplement le point ouvert. Après tout, l'objectif de cette pratique n'est pas de tout réglementer mais simplement de faciliter la lecture du code ; il est donc naturel que l'équipe détermine elle-même la portée du système de règles qu'elle adopte.

Mise en œuvre

Il n'est pas nécessaire de consigner ces règles par écrit, puisque chaque membre de l'équipe a participé à leur définition et qu'elles se retrouvent de toute manière dans toutes les parties du code produit. Les nouveaux venus pourront les découvrir en prenant pour exemple l'application entière, et leurs binômes leur transmettront l'information au fur et à mesure des développements. Dans le cas où un binôme s'écarterait du style prévu, cela serait noté tôt ou tard par un autre binôme qui pourrait alors le corriger immédiatement. La conservation de ces règles peut donc parfaitement être assurée dans le code lui-même sans qu'elles ne soient écrites explicitement.

Au début du projet, la phase de définition des règles ne consiste donc pas à élaborer un document, mais plutôt à écrire ensemble les premiers fichiers de l'application qui serviront de modèles par la suite. Il n'est pas utile à ce stade de chercher à définir l'ensemble des règles, seules les plus importantes le sont pour les premiers fichiers sachant que d'autres règles seront définies par la suite lorsque les cas concrets se présenteront.

Éléments de choix des règles

XP préconise de « voyager léger », aussi le système de règles doit-il être le plus réduit et le plus simple possible dès lors qu'il assure une homogénéité suffisante du code. Le style de codage induit par ces règles doit rester clair et sobre, et respecter le principe du *Once and only once* en évitant des duplications d'information inutiles.

Les règles portent sur les éléments suivants :

- **La présentation du code** : l'indentation, le positionnement des espaces doivent favoriser la lisibilité.
- **Les commentaires :** ils sont aussi courts et peu nombreux que possible ; on préfère remanier le code que commenter un code obscur. Ce remaniement peut consister à renommer certains éléments du programme, ou bien à déplacer certains blocs de code dans de

nouvelles méthodes dont le nom évoque plus clairement l'action effectuée ou le résultat recherché. En matière de commentaires, on évite en particulier les cartouches imposants devant les méthodes ou les classes, ceux-ci étant rarement exploités intelligemment et alourdissant le plus souvent inutilement le code.

- **Les règles de nommage** : elles doivent permettre de distinguer rapidement les différents éléments du langage en jouant sur la présentation des noms, par exemple NomDeClasse, nomDeMethode, NOM_DE_CONSTANTE, _nomDeDonneeMembre, etc.
- **Un système de noms** : l'équipe emploie un vocabulaire commun, généralement issu du domaine fonctionnel du projet ou de la métaphore choisie pour l'application. Les noms employés doivent être clairs pour tous ou bien être modifiés lorsque de meilleures idées se présentent.
- **Des idiomes de codage** : il s'agit de structures de code récurrentes, par exemple les parcours de liste, l'implémentation des singletons, etc.

La plupart des outils de développement proposent des fonctionnalités de formatage automatique du code (indentation, génération de squelettes de code). Ils doivent être configurés pour tenir compte des conventions adoptées par l'équipe. Un problème classique en la matière porte sur la largeur des tabulations ; certains outils utilisent en effet des caractères de tabulation associés à une largeur arbitraire tandis que d'autres insèrent un nombre fixe d'espaces. Sans une configuration spécifique de ces outils, leur utilisation conjointe peut aboutir à un code qui sera parfaitement bien présenté sous un éditeur et illisible sous un autre.

Règles de codage pour Java ou C++

Les équipes qui utilisent Java ou C++ peuvent s'inspirer du livre *The Elements of Java Style* de Scott W. Ambler[1]. Ce guide très complet propose des conventions de codage Java largement inspirées de celles de Sun. Il constitue un bon point de départ pour établir les règles de codage d'une équipe XP et peut être assez facilement adapté à d'autres langages.

Intégration continue

La pratique de l'intégration continue vise à ce que l'équipe XP soit en permanence en mesure de livrer une version opérationnelle du système. Les avantages d'une telle situation sont nombreux : d'une part, on évite le problème de la «période d'intégration» pendant laquelle des efforts disparates sur le système doivent être mis en commun ; d'autre part, les efforts de chaque développeur ou de chaque binôme pour améliorer l'application – en simplifier la conception ou y ajouter des fonctionnalités – profitent immédiatement, ainsi que les tests unitaires qui y sont associés, au reste de l'équipe.

1. Scott W. AMBLER & al, *The Elements of Java Style*, Cambridge University Press, 2000.

Lorsque plusieurs programmeurs travaillent sur une même application, il est indispensable que les travaux qu'ils réalisent séparément – individuellement, ou en binôme – soient à un moment ou à un autre répercutés sur une unique version «officielle» de l'application : celle qui sera compilée pour être livrée au client, puis aux utilisateurs finaux. La nécessité de cette version qui «intègre» toutes les modifications est évidente : il est déjà difficile de suivre les évolutions et les dysfonctionnements éventuels d'une seule version d'un programme écrit par un seul développeur ; s'il fallait suivre *n* versions différentes, autant que l'on compte de développeurs ou de binômes dans l'équipe, la tâche deviendrait tout à fait irréalisable.

Cette tâche d'intégration peut être entièrement manuelle, ou bien – tous les experts le recommandent, même si la pratique n'est pas universelle au sein des équipes de programmation, loin s'en faut – il peut y être procédé au moyen d'un outil informatique de gestion des versions, comme nous allons le voir. Toutefois, dans ces deux cas de figure, le problème qui se pose est le même : il s'agit de prendre en compte ou non des modifications dans la version officielle.

Les problématiques d'intégration

Il est parfaitement possible, par exemple, qu'une modification faite de bonne foi par un programmeur X fonctionne sans problème sur sa propre version, mais qu'en présence d'une autre modification, également faite de bonne foi et tout aussi parfaitement fonctionnelle, faite par un programmeur Y, un défaut apparaisse qui soit perçu comme un «bug». Il faut alors décider «quel» travail, de X ou de Y, va être officialisé, et même lorsque cette décision parfois délicate est prise, la séparation de la modification fautive d'autres réalisées par le même programmeur dans le même temps peut se révéler complexe.

Sur certains projets, la tâche d'intégration est tellement difficile qu'elle mobilise une personne à plein temps. Souvent d'ailleurs, effectuer une intégration complète est une «mission» à part entière, programmée à intervalles réguliers (toutes les deux semaines par exemple), voire à temps plein.

Le constat qui s'impose – même pour les partisans de ce mode de gestion – est le suivant : l'intégration est une tâche dont le coût et la complexité augmentent très rapidement dès lors que l'on espace dans le temps les intégrations successives. Pour XP, la solution la plus simple consiste donc évidemment à intégrer très, très souvent ! Ainsi, si la fréquence de synchronisation des développements d'une équipe «classique» est souvent de l'ordre de la semaine, celle d'une équipe XP est plutôt de l'ordre de la journée et peut même aller jusqu'à plusieurs fois par jour ou par heure – on parle alors d'*intégration continue* des modifications individuelles. Les détracteurs arguent de cette pratique qu'elle ne peut qu'engendrer le chaos – mais, nous allons le voir, les autres pratiques d'XP permettent de mettre en place des intégrations fréquentes sans que le projet n'en pâtisse.

Rôle des tests unitaires

Chaque intégration de code fait habituellement l'objet de tests de non-régression, puisque les modifications séparées des développeurs peuvent avoir des effets inattendus une fois intégrées.

> **Rappel**
> Les tests de non-régression sont destinés à vérifier que des fonctionnalités déjà validées de l'application sont toujours opérationnelles après modification de cette dernière.

Dans les équipes où ces tests sont réalisés ou interprétés manuellement il n'est pas concevable d'intégrer plusieurs fois par jour puisque cela consommerait trop de temps ou ne permettrait pas une couverture de test suffisante.

Les tests unitaires automatiques préconisés par XP se révèlent donc indispensables pour donner un *feedback* rapide sur le processus d'intégration. Les développeurs peuvent ainsi s'assurer que les tests unitaires passent à 100 % sur leur version locale du code avant intégration, et qu'ils passent toujours à 100 % après.

La viabilité de cette pratique dépend de deux facteurs :

- La compilation de l'application et l'exécution de l'ensemble des tests unitaires doivent être très rapides. Cet aspect est primordial dans XP, et l'équipe doit faire le nécessaire sur le plan du matériel, des outils et de l'organisation de son application. Elle doit en particulier se doter d'un outil de gestion de dépendances efficace qui limite la quantité de code à recompiler après chaque modification.
- Les tests unitaires disponibles doivent être suffisamment couverts pour que l'équipe n'ait pas à se reposer sur les tests de recette, dont l'exécution est souvent bien plus longue.

Organisation des copies de travail

L'utilisation d'un outil de gestion de versions s'impose dès lors que plusieurs développeurs travaillent sur une même application. Ce type d'outils leur permet de travailler sur des versions isolées du logiciel et de «synchroniser» leur travail avec celui du reste de l'équipe lorsqu'ils le désirent.

Voici comment se présente l'organisation habituelle d'une équipe XP : chaque binôme dispose de son propre espace de travail, correspondant à une version privée du logiciel, et l'équipe dispose d'une version commune du logiciel qui lui sert de point de synchronisation. Cette version commune constitue en quelque sorte un tampon entre les développeurs, évitant que chaque livraison d'un binôme ne perturbe les autres binômes. Puisque chaque intégration garantit que les tests passent à 100 %, cette version commune représente un logiciel toujours prêt à livrer intégrant au moins les développements de la veille.

Remarque

Les espaces privés de développement doivent être aussi indépendants que possible. Nous avons constaté en pratique que les outils de gestion de versions qui limitent les accès concurrents aux fichiers par un système de verrouillage peuvent peser sur le travail de l'équipe. En effet, cette politique de gestion des accès concurrents est adaptée lorsque les développeurs travaillent sur des parties distinctes de l'application, mais ce n'est pas le cas au sein d'une équipe XP : mieux vaut adopter un outil qui autorise les modifications concurrentes sans contraintes et propose ensuite des fonctions efficaces pour fusionner les modifications.

Le processus de modification et d'intégration

Le schéma de la figure 5-3 présente dans ses grandes lignes le cycle de modification du code et d'intégration à la version commune :

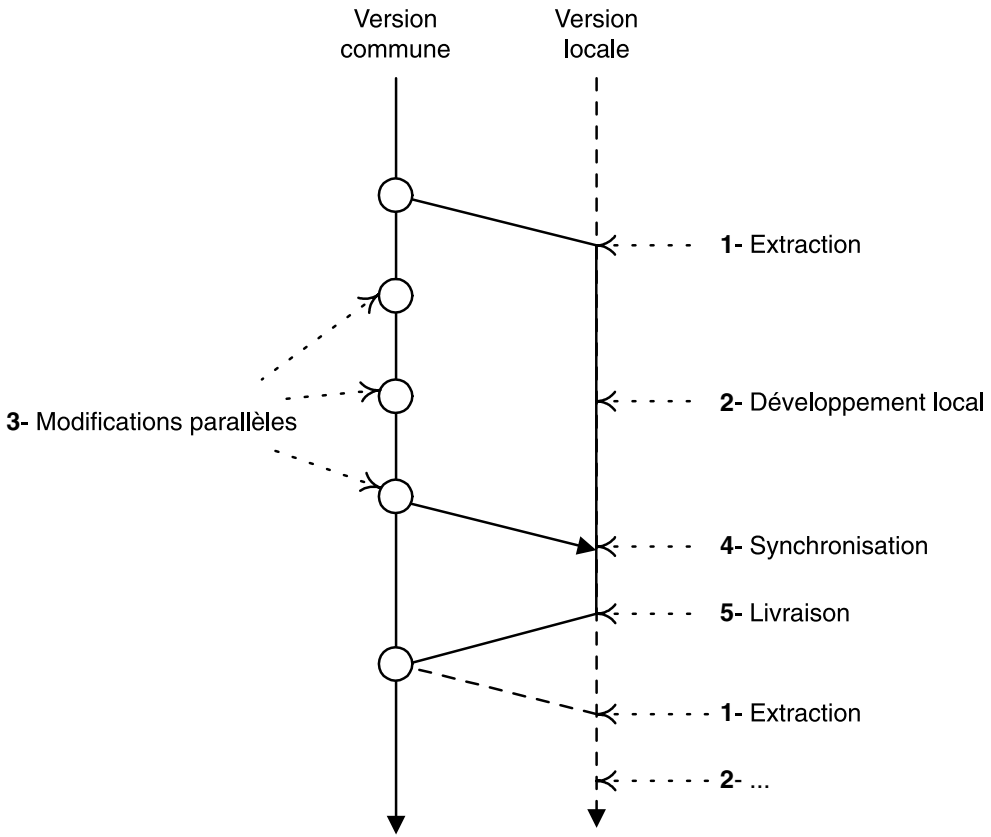

Figure 5-3. Cycle de modification et d'intégration du code

Les étapes de ce cycle sont les suivantes :

1. Le binôme extrait une version locale du logiciel à partir de la version commune de l'équipe.
2. Le binôme travaille sur sa version locale, selon le cycle habituel d'écriture des tests, d'écriture du code, de compilation, puis d'exécution des tests.
3. Pendant ce temps, d'autres binômes peuvent intégrer leur travail à la version commune.
4. Lorsque le binôme parvient à une version stable du logiciel dans laquelle les tests passent à 100 %, il met à jour sa version locale par rapport aux évolutions de la version commune. Il faut noter ici que cela peut se produire à tout moment et non pas uniquement à la fin d'une tâche, la seule contrainte étant que tous les tests unitaires existants passent à 100 %.
5. Le binôme corrige les éventuels problèmes survenus lors de l'intégration, en faisant appel si besoin au binôme qui a intégré juste avant lui. Lorsque les tests passent de nouveau à 100 %, il rapatrie son code dans la version commune du logiciel. Dans le cas exceptionnel où il ne parviendrait pas à stabiliser la version intégrée, le binôme peut décider d'abandonner ses modifications et de repartir d'une version plus récente du logiciel.

Dans ce schéma, les étapes 4 et 5 ne doivent être réalisées que par un binôme à la fois. Ainsi, si les tests échouent, il est plus facile de savoir d'où vient le problème, et surtout qui doit le corriger. L'idéal est de pouvoir disposer d'une machine dédiée à l'intégration, sur laquelle réside la version commune du logiciel. Le binôme qui souhaite livrer son travail se rend sur cette machine d'intégration, les autres binômes savent donc qu'ils ne peuvent intégrer pendant ce temps. En l'absence de cette machine d'intégration, il faut signifier oralement aux autres binômes que l'on est en train d'intégrer. Pour éviter tout accident, il peut être utile et amusant de disposer d'un «jeton» (un objet physique comme un chapeau ou un drapeau, unique et réservé à cet usage) que l'on pose sur le bureau du binôme en train d'intégrer, ou simplement d'inscrire le nom du binôme au tableau dans un coin réservé à cet effet.

6

Gestion de projet

Je sais pourquoi tant de gens aiment couper du bois. C'est une activité où l'on voit tout de suite le résultat.

– Albert Einstein

Les pratiques que nous avons présentées jusqu'ici étaient centrées sur le développement, qu'il s'agisse de la programmation ou de l'organisation des développeurs. Dans ce chapitre, nous allons décrire les pratiques complémentaires qui touchent à la planification, au suivi du projet, à l'identification des besoins, bref les pratiques qui lient une équipe XP à son client.

Rappel : client ou client XP ?

Nous utilisons ici le terme «client» en référence au rôle XP présenté au chapitre 2. Il est important de garder à l'esprit qu'au sens d'XP, le client est un membre à part entière de l'équipe de développement. Le terme «client» sera utilisé de façon plus informelle. Compte tenu du contexte, il désigne donc ici la personne qui finance le projet, le maître d'ouvrage ou encore l'utilisateur final de l'application.

Là encore, XP adopte une démarche originale, orientée par ses quatre valeurs fondatrices : une *communication* directe avec le client plutôt qu'au travers de documents ; divers mécanismes de *feedback* pour améliorer continuellement le pilotage du projet ; la *simplicité* du processus lui-même ; le *courage* d'afficher une transparence complète vis-à-vis du client et d'avancer dans le projet sans avoir tout planifié dans le détail.

Avant de passer en revue les pratiques concrètes qui soutiennent ces valeurs, nous allons aborder dans les sections suivantes les principes de la gestion de projet en XP.

Principes de la démarche

Rechercher le rythme optimal

Lorsqu'on interroge les développeurs de projets qui sont en difficulté, l'une des plaintes rapportées le plus fréquemment concerne le manque de temps : pas le temps d'écrire un code propre, pas le temps de faire des tests suffisants, pas le temps de creuser les spécifications, en somme pas le temps de réaliser un travail de qualité. Or, les modifications d'un logiciel de mauvaise qualité exigent plus de travail encore et engendrent souvent des régressions qui consomment également plus de temps, de sorte que le phénomène ne cesse de s'auto-entretenir. Pour gâter le tout, les projets soumis à ce type de cercle vicieux sont extrêmement difficiles à gérer, car les estimations de coût fournies par les développeurs deviennent de moins en moins fiables.

Comment échapper à ce phénomène ? Les projets informatiques ont très souvent une importance stratégique pour les entreprises qui souhaitent obtenir un maximum de fonctionnalités en un temps minimal pour faire face à la concurrence. Ces contraintes poussent les équipes de développement dans leurs retranchements, et ces dernières sont souvent amenées à accepter le défi de peur de perdre leur crédibilité, ou pire leur travail. Comment garantir à l'équipe un rythme de fonctionnement optimal, tout en répondant aux contraintes de l'entreprise ?

Selon les auteurs d'XP, un projet est dimensionné par quatre variables interdépendantes : le *coût*[1], le *délai* de livraison, la *qualité* du produit et son *contenu*. Nous reviendrons plus en détail sur ces variables dans le chapitre 8 consacré aux coûts d'un projet XP, mais pour le moment il nous suffit de considérer qu'il est possible de doser le rythme de l'équipe en jouant sur chacune d'entre elles. Or, parmi ces quatre variables, les trois premières présentent des effets secondaires gênants :

- Jouer sur les coûts en intégrant de nouveaux membres à l'équipe ou en changeant l'environnement de travail, c'est risquer de ralentir l'équipe sur le court terme.
- Jouer sur les délais en repoussant les livraisons, c'est générer des coûts indirects de synchronisation, et c'est aussi éroder progressivement la confiance du client dans l'équipe et saper la confiance de l'équipe en elle-même.
- Jouer sur la qualité, c'est prendre le risque d'être confronté au cercle vicieux évoqué plus haut, mais c'est également mettre en péril la motivation de l'équipe et la confiance du client.

Pour ces raisons, une équipe XP régule son rythme de travail en jouant uniquement sur la dernière variable : l'enveloppe fonctionnelle du produit. La figure 6-1 illustre ce phénomène

1. Conditionné notamment (mais pas uniquement) par la taille de l'équipe : on utilise fréquemment le «mois-homme» comme unité d'estimation du coût des projets… bien que cette association soit largement discréditée depuis la parution de l'ouvrage de Fred Brooks, *The Mythical Man-Month* («Le mythe du mois-homme»).

sous la forme d'un tableau de bord qui permet d'agir sur la pression subie par l'équipe, à la recherche de son rythme de fonctionnement optimal.

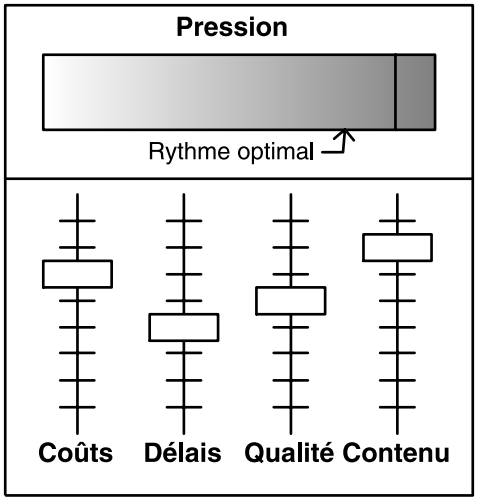

Figure 6-1. Quatre variables pour doser la pression du projet

Mais si l'équipe doit jouer sur l'enveloppe fonctionnelle du produit, cela implique-t-il nécessairement que le client devra se contenter d'un produit incomplet ?

En réalité, cette variable offre une flexibilité rarement soupçonnée. Les équipes de développement parviennent souvent mal à cerner les besoins réels du client et tentent de compenser cela en implémentant un surplus de fonctionnalités. Outre le temps perdu à développer des choses inutiles, l'application s'en trouve surchargée, ce qui affecte la compilation, les tests, la gestion de configuration, les développements suivants, etc.

Le tri des fonctionnalités les plus utiles se présente donc comme un moyen efficace pour alléger la charge de l'équipe et pour accroître à la fois la vitesse du développement et la qualité du produit. En cela, la valeur XP de simplicité trouve ici tout son sens : l'équipe va plus vite en se concentrant sur l'essentiel.

Nous allons expliciter dans les sections suivantes l'approche adoptée par XP pour mieux cerner les besoins du client :

- une démarche itérative pour améliorer en permanence la compréhension des besoins et s'adapter aux changements souhaités par le client ;
- un partage des responsabilités clair entre l'équipe de développement et le client, ainsi qu'une plus grande implication de ce dernier dans le projet.

(Re)définir régulièrement le projet

Limites de l'approche linéaire traditionnelle

« Visez… Feu ! Raté… »

L'approche « traditionnelle » du développement logiciel, fondée sur une construction linéaire de l'application à partir de spécifications initiales, présente des limites qui sont aujourd'hui bien connues :

- Il est très difficile d'établir *a priori* la spécification complète d'une application, et les quelques détails oubliés peuvent remettre en question des pans entiers de la solution proposée.
- Les besoins du client évoluent fréquemment au cours du projet, en particulier dès qu'il utilise les premières versions du logiciel. Là encore, une partie des efforts initiaux de spécification et de conception est perdue.

En d'autres termes, il est vain de passer trop de temps à viser la cible au début du projet : non seulement elle est trop loin, mais de plus elle se dérobe pendant le vol du projectile !

Adapter le périmètre fonctionnel en fonction des priorités

« Feu ! Visez, visez, visez… »

Un projet XP commence également par une phase initiale d'exploration du problème, mais l'ambition de cette phase est nettement plus modeste que dans une démarche « linéaire ». Il s'agit juste pour l'équipe de recenser les besoins du client, avec une granularité assez large qui permette simplement de réaliser les choix techniques initiaux et d'estimer de manière suffisamment fiable les coûts des fonctionnalités demandées.

À partir de ces estimations, le client et les programmeurs conviennent d'un plan de livraisons, celles-ci commençant très tôt dans le projet et s'enchaînant ensuite à un rythme assez soutenu. Il revient au client de gérer finement les priorités des fonctionnalités demandées, de sorte que les plus importantes soient implémentées en premier et disponibles rapidement. Mais cela reste, somme toute, assez classique.

Ce qui distingue XP sur ce point, c'est que le client peut changer ce plan à loisir en cours de projet. Il peut avancer des fonctionnalités qui lui paraissent prendre de l'importance, retarder des fonctionnalités dont il doute de la pertinence, en supprimer d'autres qui sont devenues inutiles, ou encore en ajouter de nouvelles. En outre, il s'appuie pour cela sur des estimations de plus en plus fiables, puisque à chaque livraison l'équipe réévalue le coût des fonctionnalités restantes en fonction du coût de celles qui ont été réalisées.

C'est à travers ces divers mécanismes de *feedback* que les fonctionnalités les moins utiles sont éliminées, et que l'équipe joue sur la variable « contenu » du projet. Ces mécanismes sont détaillés plus bas à la section « Planification itérative » de ce chapitre.

Notre XPérience

La démarche XP offre des avantages indéniables en termes d'ouverture au changement, mais cette souplesse même interdit à l'équipe de s'engager sur un résultat précis en début de projet. L'approche XP se fonde davantage sur une obligation de moyens, qui suppose une relation de confiance entre l'équipe de développement et son client – pris ici au sens de celui qui finance le projet. Or, l'adoption d'une telle relation peut se heurter à un problème de culture : un client habitué à une démarche plus classique aura certainement du mal à se lancer dans le projet sans disposer d'engagements forts de la part de l'équipe. Cette barrière peut être levée par un effort particulier d'explication au début du projet, mais une équipe qui démarre un projet XP devra s'attacher à produire très rapidement des résultats concrets pour maintenir et renforcer la confiance de son client.

Maintenir l'équilibre entre développeurs et client

L'ouverture au changement proposée par XP n'est envisageable que si le client et les développeurs jouent le jeu et assument leurs responsabilités respectives, c'est-à-dire que le client prend les décisions fonctionnelles et/ou économiques, tandis que l'équipe prend les décisions techniques.

Des deux côtés de cet équilibre, le projet est en péril. Si le client a trop de pouvoir, il risque de demander l'impossible à l'équipe – sous sa forme la plus maligne, ce phénomène correspond au stéréotype du projet « marche à mort » (*deathmarch project*). Si l'équipe a trop de pouvoir, elle risque de s'enliser dans la création du tout dernier « framework intégré de nouvelle génération », et ne jamais apporter une quelconque valeur à son client.

L'équilibre proposé par XP repose sur une séparation stricte des responsabilités :

- Le client définit les fonctionnalités à intégrer au logiciel, et il décide également de l'ordre selon lequel elles sont implémentées.
- Les développeurs fournissent des estimations de coût pour les fonctionnalités proposées par le client et se chargent de la réalisation des fonctionnalités choisies.

Ces responsabilités se traduisent de manière dynamique par les interactions présentées en figure 6-2.

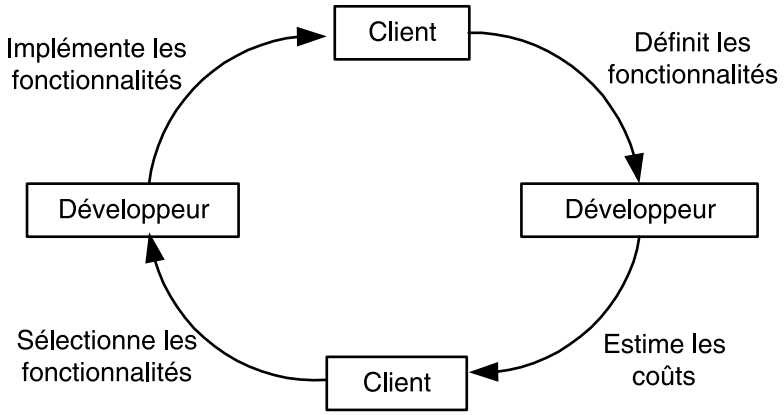

Figure 6-2. Cycle des relations client/ développeurs

Définir les spécifications tout au long du projet

XP fonctionne sans l'habituel document d'analyse et de spécifications détaillées. Les activités correspondantes sont conservées, mais l'élaboration du document est abandonnée au profit d'un contact direct et permanent avec le client.

Cela ne signifie pas que les spécifications ne sont consignées nulle part : elles se retrouvent sous la forme d'un ensemble de tests de recette automatiques, chargés de déterminer de manière objective si le logiciel répond ou non aux besoins exprimés par le client. Ces tests sont toujours en phase avec le produit, ce qui n'est pas toujours le cas des documentations papier.

Par ailleurs, les activités d'analyse et de spécification ne sont plus limitées au début du projet, mais se trouvent échelonnées tout au long de son déroulement. Elles deviennent des activités quotidiennes des développeurs, au même titre que la programmation et le test. Comme nous l'avons évoqué au chapitre 2, le rôle du développeur XP ne se réduit pas à l'écriture du code : il intègre également l'analyse des besoins du client et l'élaboration des solutions correspondantes.

Par rapport à l'approche « linéaire » du développement, cette démarche pose le problème de la conception : en l'absence de spécifications, comment définir l'organisation de l'application ? C'est ici que les techniques de programmation présentées au chapitre 3, basées sur une conception simple, le remaniement et les tests unitaires, prennent toute leur importance. Ces pratiques permettent de faire émerger l'architecture de l'application au fur et à mesure des besoins, et surtout l'application est ainsi toujours prête à suivre les changements souhaités par le client.

Notre XPérience

Comme il en va de la plupart des autres principes XP, ce principe de *just in time specification* est optimisé pour une ouverture maximale au changement. Au début, cependant, il risque de donner l'impression d'un manque de visibilité en début de projet, ce qui peut notamment entraîner quelques difficultés lorsque le projet doit être synchronisé avec d'autres, qui ne disposent pas d'une telle souplesse. Nous avons pu par exemple rencontrer cette situation dans des contextes industriels où le projet global comportait une part de développement *hardware*, et où des spécifications initiales complètes étaient indispensables. Dans ce type de cas, il est tout à fait possible d'intégrer le contexte général en partant de ces spécifications complètes, et en appliquant la démarche itérative d'XP dans le cadre de ces spécifications. Nous reviendrons plus en détail sur ce point au chapitre 12, consacré au retour d'expérience d'un projet industriel mené avec XP.

Les pratiques XP de gestion du projet

L'approche que nous venons de décrire est soutenue par les quatre pratiques suivantes :

- **Client sur site** (*on-site customer* ou *Whole Team*) : le client est intégré à l'équipe de développement, il apporte ses compétences métier et définit les tests de recette du produit.
- **Rythme durable** (*sustainable pace*) : l'équipe adopte un rythme de travail raisonnable, qui lui permet de produire un travail de qualité sur une longue durée.
- **Livraisons fréquentes** (*frequent releases*) : l'équipe adopte un rythme soutenu de livraisons, et ce, très tôt dans le projet.
- **Planification itérative** (*planning game*) : la gestion du projet suit une démarche itérative, basée sur des séances de planification régulières qui réunissent le client et l'équipe de développement.

Ces pratiques vont être décrites une à une dans les sections suivantes.

Client sur site

Nous verrons – notamment avec la pratique de «planification itérative» – que le client est fortement sollicité pour la définition des besoins et l'arbitrage des priorités tout au long du projet. Son intervention ne s'arrête pas là : dans un projet XP, il est entièrement intégré à l'équipe de développement et participe quotidiennement au projet. Nous évitons en général (et en particulier dans ce chapitre) de parler de «l'équipe» comme s'il s'agissait uniquement des développeurs : pour XP, il n'y a qu'une équipe (*One Team* ou *Whole Team*) qui réunit le client et les programmeurs.

Le client apporte ses compétences métier

XP fonctionne sans document de spécifications mais s'appuie en contrepartie sur une communication permanente entre les développeurs et le client.

En cours d'implémentation, il est très fréquent que les développeurs tombent sur des questions épineuses d'ordre fonctionnel. Traditionnellement, le document de spécifications laisse beaucoup de zones d'ombre de ce type, et c'est finalement aux développeurs eux-mêmes de trouver des solutions. Or, les développeurs sont souvent davantage des spécialistes de l'informatique que du domaine fonctionnel dans lequel ils évoluent, et ils perdent ainsi du temps à traiter des problèmes qui ne correspondent pas à leurs attributions. Avec cette pratique, XP s'assure que le client garde complètement la maîtrise d'ouvrage et que les développeurs se focalisent sur la programmation.

Le client définit les tests de recette

Nous avons vu aux chapitres 3 et 4 que les développeurs écrivaient des tests de non-régression automatiques en même temps que le code proprement dit. Ces tests valident le fonctionnement des composants élémentaires du système, mais ils ne concernent que les développeurs et ne prouvent pas nécessairement que le logiciel répond aux attentes du client.

À cet effet, XP emploie un second type de tests, appelés tests de recette (*acceptance tests*), tests fonctionnels ou tests de recette. Ces tests sont également automatiques, dans le sens où ils déterminent de manière binaire et objective si la fonctionnalité est acceptée ou non. Dans la mesure où ces tests sont l'équivalent XP des spécifications du produit, leur définition est l'affaire du Client lui-même.

Dans un cadre XP idéal, le client implémente lui-même ces tests, avec l'aide éventuelle de testeurs. De leur côté, les développeurs doivent fournir au client et aux testeurs des outils pour faciliter la réalisation de ces tests.

Notre XPérience

En réalisant un projet XP dans un contexte industriel friand de documentations, nous avons pris l'initiative d'enrichir nos outils de tests de recette afin qu'ils génèrent automatiquement leur propre documentation sous la forme de pages Web mises à disposition de tous. Au moyen de ces pages, nous pouvions « reboucler » avec l'équipe d'architectes du projet, lesquels ne souhaitaient pas nécessairement devoir relire les tests eux-mêmes pour retrouver la description des fonctionnalités du système. Ces pages nous permettaient également de montrer l'avancement concret du projet à la direction.

La plus inaccessible des pratiques ?

Bien des équipes éprouvent des difficultés à mettre en place cette pratique, parce qu'il est difficile de trouver une seule personne qui puisse assurer les différents rôles du client et être suffisamment disponible pour cela.

La solution consiste à partager le rôle entre plusieurs personnes et nous renvoyons au chapitre 2 pour une discussion sur les différentes possibilités envisageables. Néanmoins, il faut garder à l'esprit le fait que l'efficacité de la méthode sera d'autant plus grande que le projet se rapprochera de son cadre idéal. Aussi, il vaut mieux limiter autant que possible la dispersion des attributions de ce rôle et, si c'est inévitable, il faut s'assurer que les différentes personnes concernées y participent pleinement.

Rythme durable

La plupart des pratiques présentées dans ce chapitre ont pour objet de maintenir le rythme de travail à un niveau optimal. Cela signifie que les développeurs disposent de suffisamment de temps pour réaliser un travail de qualité mais aussi pour se reposer. En effet, une équipe fatiguée ou perturbée par des problèmes familiaux dus au travail ne peut avoir la démarche positive visée par XP : il faut beaucoup d'énergie pour trouver des solutions de conception simples, écrire du code propre, penser à des tests unitaires adaptés et explorer tous les problèmes rencontrés au fil des discussions avec son binôme.

Notre XPérience

Lorsque nous avons commencé à mettre en place sérieusement les pratiques de programmation XP sur l'un de nos projets, en particulier le travail en binôme, nous arrivions difficilement à garder le rythme plus de six heures par jour. Cela peut paraître faible de prime abord, mais c'était réellement la durée maximale durant laquelle nous pouvions maintenir le rythme et la qualité d'un développement XP.

Il semble opportun de rappeler que les contrats de travail régissant le statut « cadre » – les plus courants pour les ingénieurs de développement et chefs de projet – instaurent le plus souvent une certaine flexibilité horaire. Dans le cadre d'un travail de développement logiciel, on a affaire à une organisation optimale – le « temps de présence » d'un développeur n'a aucune valeur en soi, c'est sa productivité intellectuelle qui est capitalisée par l'employeur.

Cela ne signifie pas pour autant que les heures supplémentaires n'existent pas en XP. L'équipe peut être amenée à travailler beaucoup sur une courte période de temps, mais il faut être lucide : toute période de surcharge est généralement suivie d'une période de relâchement qui compense – au mieux – le gain obtenu. En outre, c'est justement pendant les périodes de surcharge que l'équipe perd ses bonnes habitudes (tests unitaires, remaniements) et que les développeurs introduisent des défauts dans le logiciel… et tout le monde le paiera tôt ou tard.

Pour ces raisons, une équipe XP adopte la règle suivante : pas d'heures supplémentaires deux semaines de suite.

De toute manière, si l'équipe ne parvient pas à respecter cette règle, c'est qu'il y a un problème dans la façon dont le projet est géré. Mieux vaut alors chercher la source réelle de ce problème, afin de le résoudre, que d'essayer de le masquer par un surplus de travail. Cela demande certes du courage de la part du coach et des divers intervenants du projet, mais rappelons ici qu'une équipe XP joue pour gagner, et surtout pas pour ne pas perdre.

Livraisons fréquentes

Un projet XP est rythmé par des livraisons régulières, qui permettent à chacun des acteurs du projet de se synchroniser autour de résultats concrets. Les dates de ces livraisons sont fixées par le client.

La figure 6-3 donne un exemple de plan de livraisons pour un projet de six ou sept mois, dans lequel les livraisons sont espacées d'environ un mois et demi.

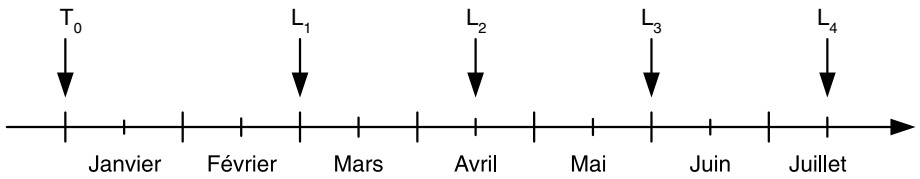

Figure 6-3. Un plan de livraisons type

Il ne s'agit cependant là que d'un exemple : les contraintes d'une application intranet réalisée en trois mois ne sont pas celles d'un logiciel de contrôle aérien qui prend plusieurs années. Toutefois, les deux idées clés à retenir sont les suivantes :

- La première livraison doit arriver très tôt, pour dissiper d'éventuels malentendus et donner consistance au projet.
- Les livraisons suivantes doivent être aussi rapprochées que possible, pour permettre un pilotage précis du projet et donner des preuves fréquentes de son avancement.

Notre XPérience

Le client peut parfois douter de l'intérêt d'une première livraison trop précoce, le faible nombre de fonctionnalités proposées rendant l'outil inexploitable en conditions réelles. Il faut cependant s'efforcer de livrer le plus tôt possible, quitte à ce que l'application ne soit utilisée que quelques heures par un ou deux utilisateurs finaux. L'important à ce stade est de réunir les programmeurs et le client autour du premier résultat concret du projet, et cette étape est souvent riche en enseignements pour tous.

Poussée à l'extrême, cette pratique peut se traduire par des livraisons quotidiennes de l'équipe à ses utilisateurs, l'équipe s'appuyant en cela sur les pratiques d'intégration continue et de tests automatiques présentées aux chapitres précédents. Cela peut être assez facilement mis en œuvre lorsque les techniques de déploiement s'y prêtent, par exemple dans des applications de type Web, ou bien lorsque les utilisateurs sont en nombre limité et que l'équipe peut mettre à leur disposition des versions récentes du produit sur un réseau local ou un site intranet.

Dans le cas d'une équipe XP bien rodée, c'est surtout la lourdeur du processus de livraison qui va en limiter la fréquence. Par exemple, les livraisons seront certainement assez espacées si le produit doit passer par une phase de validation externe avant d'arriver entre les mains des utilisateurs finaux ; ou encore si c'est un logiciel «embarqué», destiné à être placé dans la ROM d'un appareil tel qu'un téléphone portable.

La pratique des livraisons fréquentes est l'expression directe de la valeur de *feedback* adoptée par XP ; nous allons voir dans les sections qui suivent qu'elle est destinée aussi bien au client qu'aux programmeurs.

Feedback pour le client

Chaque livraison permet au client d'être rassuré quant à l'avancement du projet et également de mieux apprécier le travail des programmeurs. Mais surtout, elle lui offre l'occasion d'être au contact avec le produit, et ainsi de mieux cerner ses besoins pour les livraisons suivantes.

L'un des reproches adressés à XP est qu'il nécessite de la part du client de très fortes compétences pour définir ce qui a ou ce qui n'a pas de valeur pour le projet. Il est clair qu'en resserrant la «boucle de feedback» relative aux choix de fonctionnalités que fait le client, XP limite ou compense complètement ce risque : lorsque le client peut voir, tous les mois, toutes les deux semaines, voire plusieurs fois par semaine, ce que «donnent» les scénarios client qu'il a spécifiés, il ne peut que devenir plus efficace dans sa mission.

Ce n'est pas le cas pour le client non-XP qui constate au bout de six ou neuf mois, si ce n'est plus, l'interprétation qu'une équipe a donnée à sa formulation des besoins.

Feedback pour l'équipe

Les livraisons procurent à l'équipe un sentiment régulier de «travail fini» (*sense of completion*), qui entretient la motivation et lisse la pression sur l'ensemble du projet.

Les livraisons fréquentes présentent également des bénéfices pratiques, en offrant l'occasion de sortir le produit de son environnement de développement pour le confronter à un environnement réel. Par exemple, un site Web sera publié et visité à distance, ce qui permettra de mettre en évidence des problèmes de temps de chargement qui ne peuvent être perçus lorsque le site est consulté à l'intérieur d'un réseau local.

Remarque

Ces livraisons régulières donnent au client une grande visibilité sur l'avancement *réel* du projet, et cela peut gêner certaines équipes habituées à rester «tranquilles» pendant de longues périodes. D'un autre côté, la transparence prônée par XP peut se révéler inadéquate lorsque le client n'est pas disposé à jouer le jeu et cherche à pousser l'équipe dans la «zone rouge». Dans les deux cas, il faudra obtenir un réel changement de mentalités avant qu'une relation de type «gagnant-gagnant» telle que celle proposée par XP soit envisageable et puisse porter ses fruits. Mais cette remarque n'est pas propre à XP, et on peut légitimement se demander comment un projet peut être rentable dans ce genre de situation.

Planification itérative

La pratique des livraisons fréquentes d'XP donne des indications sur la *répartition* des livraisons au cours du projet. Quant à elle, la pratique de planification itérative définit les mécanismes de contrôle de leur *contenu*.

Les séances de planification collectives (planning game)

Pour faciliter la compréhension de leur mécanisme de planification, mais aussi pour alléger les tensions qui accompagnent habituellement ce genre d'activités, les auteurs d'XP présentent généralement le processus qu'ils préconisent sous la forme d'un jeu de société dans lequel le client et les programmeurs collaborent pour tirer le meilleur produit du projet.

Ce jeu décompose le projet en une suite de tours appelés «itérations», dont la durée est fixée de 1 à 3 semaines selon les projets. Comme le montre la figure 6-4, les livraisons sont calées sur les itérations au rythme d'une livraison toutes les deux ou trois itérations en général. Les dates de ces livraisons sont déterminées par le client.

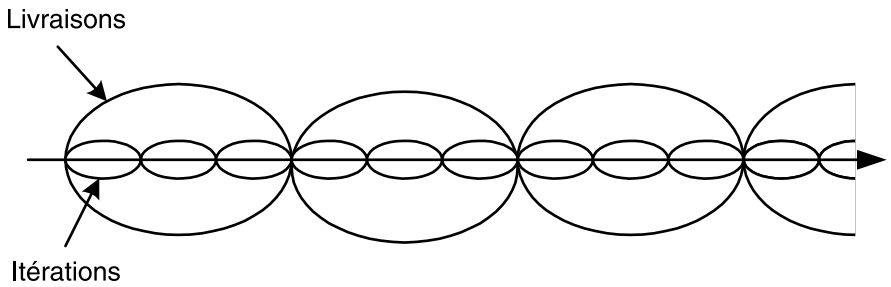

Figure 6-4. Décomposition du projet en livraisons et en itérations

Notre XPérience

Pour un projet qui s'étale sur plusieurs mois, des itérations de deux semaines paraissent les plus appropriées. Cela permet d'avancer suffisamment à chaque itération tout en apportant une bonne visibilité au client. Par ailleurs, les itérations ne doivent pas nécessairement commencer en début de semaine. Sur certains projets, nous avons choisi de commencer nos itérations le mardi, de sorte que l'équipe reprenne plus rapidement son activité le lundi matin.

Les livraisons et les itérations représentent donc deux cycles imbriqués d'évolution du projet, ou deux niveaux de granularité distincts pour son pilotage. Le cycle des livraisons porte sur les fonctionnalités visibles par le client, celui des itérations porte sur les tâches réalisées par les développeurs. Comme l'illustre la figure 6-5, ces deux cycles partagent un même découpage en trois phases :

- une phase assez courte d'*exploration* (*exploration*), qui consiste à identifier le travail à réaliser et à estimer son coût ;
- une phase très courte d'*engagement* (*commitment*), qui consiste à sélectionner le travail à réaliser pour le cycle en cours ;
- la phase la plus longue, celle du *pilotage* (*steering*), qui consiste à contrôler la réalisation des fonctionnalités demandées.

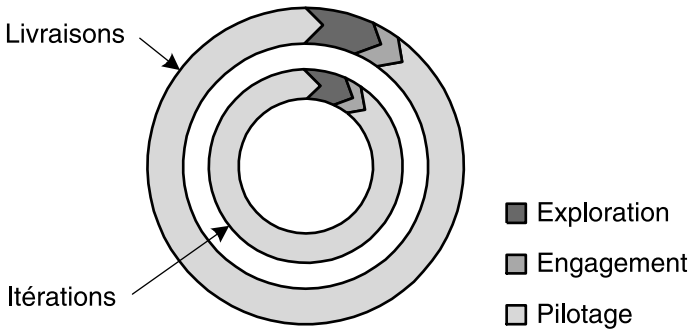

Figure 6-5. Les trois phases des cycles de livraisons et d'itérations

Les sections qui suivent décrivent les techniques de planification et de suivi propres à ces deux cycles.

Remarque

Il est tout à fait possible de confondre les deux cycles (livraison et itération) en un seul, si le rythme des itérations convient aussi au client pour les livraisons.

Par ailleurs, ce découpage des cycles imbriqués en «phase» peut sembler très formel, voire contraire aux principes mêmes d'XP. Il ne s'agit là que d'une vision théorique, qui permet d'adapter la démarche en fonction des situations rencontrées. Dans la pratique, par exemple, la «phase d'exploration» et la «phase d'engagement» du plan d'itération peuvent faire l'objet d'une seule réunion, pas nécessairement très formelle, en début d'itération.

La planification des livraisons

La phase préliminaire d'exploration

Au début du projet, puis au début de chaque cycle de livraison, le client et les programmeurs se réunissent pour définir les fonctionnalités à implémenter et pour estimer leur coût. La figure 6-6 reprend les diverses activités menées durant cette phase.

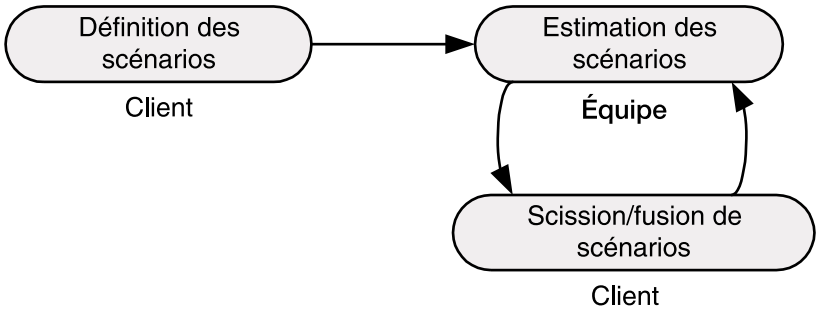

Figure 6-6. La phase d'exploration du cycle des livraisons

Définition des scénarios client

Les fonctionnalités sont décrites sous la forme de scénarios client (*user stories*), rédigés par le client lui-même, le plus souvent avec l'aide du coach. Le client y décrit, dans son propre langage, les interactions principales entre l'utilisateur et le système pour une fonctionnalité donnée. Mais cette description reste très sommaire : à la différence des *Use Cases* popularisés par UML, les scénarios client d'XP n'ont pas pour vocation de définir précisément une fonctionnalité. Ils l'identifient uniquement à des fins de planification, sachant que les détails seront échangés directement entre le client et les développeurs le moment venu. De ce point de vue, un scénario client est la promesse d'une discussion à venir entre le client et les programmeurs.

En tant qu'élément de planification, un scénario client doit pouvoir être facilement estimé et suivi. À cette fin :

- Il doit avoir une granularité suffisamment fine, de sorte que les développeurs puissent en avoir une vision assez complète et qu'ils puissent en implémenter plusieurs en une itération. Pour un projet qui utilise des itérations de deux semaines, chaque scénario représente au maximum une semaine de travail pour un binôme.
- Il doit être testable, c'est-à-dire qu'il doit être possible de concevoir un ou plusieurs tests de recette *automatiques* qui prouvent que la fonctionnalité est complètement implémentée.

Exemples

Il est difficile de présenter des scénarios client issus de nos projets, car ils ne sont souvent compréhensibles que dans leur contexte. Voici cependant quelques exemples de ce que pourraient être les scénarios client d'un navigateur Web :

« L'utilisateur peut conserver les adresses de ses sites préférés et les organiser sous forme arborescente. »

« Au démarrage, afficher automatiquement la dernière page visitée par cet utilisateur. »

« Proposer un mode de visualisation en plein écran. »

Les scénarios client peuvent également définir des contraintes de performance ou de robustesse imposées au produit :

« Le temps de lancement de l'application doit être inférieur à 10 secondes sur la machine de référence. »

« Si le serveur distant rompt la connexion en cours de chargement, la page doit être affichée en partie si la quantité d'information récupérée le permet. »

Les scénarios sont notés sur de petites fiches cartonnées (le format A5 convient parfaitement), qui se prêtent facilement aux manipulations. Un scénario n'est plus utile ? On déchire la fiche correspondante. Il manque un scénario ? On sort une fiche vierge. Les fiches peuvent ainsi être triées, partagées, échangées plus aisément qu'avec un document électronique. Cela illustre parfaitement la valeur de simplicité prônée par XP.

Outre la description proprement dite du scénario, chaque fiche est amenée à recevoir des indications associées à la planification des activités correspondantes (figure 6-7). Nous décrivons ces indications supplémentaires dans les sections suivantes.

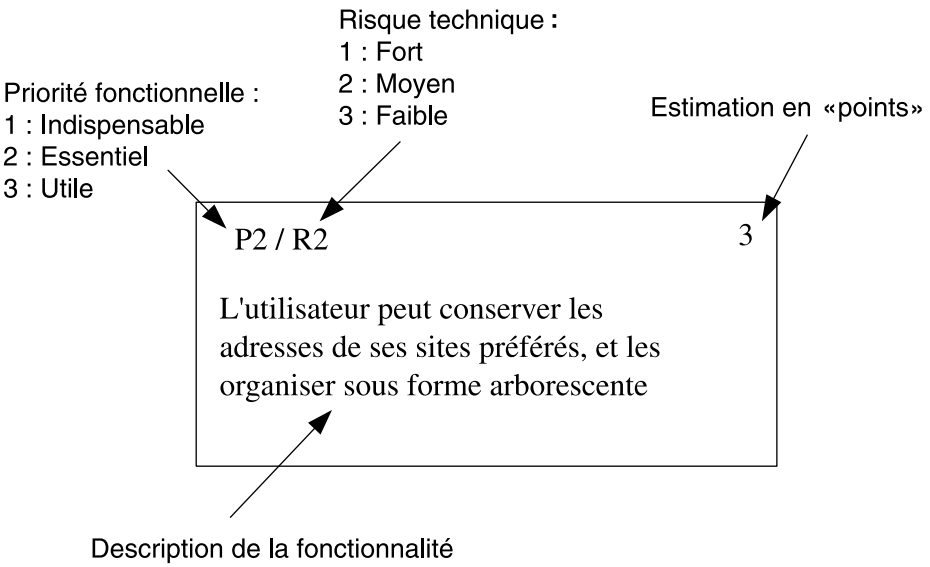

Figure 6-7. Exemple de scénario client.

Estimation des scénarios

Une fois que le client a défini les scénarios, il se joint aux programmeurs pour une séance d'estimation de leurs coûts. Au cours de cette séance, le client décrit les scénarios un par un et répond aux questions de l'équipe qui s'efforce de cerner au mieux leur envergure et leur difficulté pour en établir une estimation fiable. Cette estimation ne se réduit pas à la programmation seule, elle doit impérativement intégrer l'écriture des tests unitaires et la validation à l'aide des tests de recette. Ce travail se fait en groupe ; chaque développeur a ainsi l'occasion de prendre connaissance des scénarios et d'apporter sa propre expérience.

Les estimations ne se basent pas sur des durées calendaires, sujettes à trop d'inconnues – l'exercice est déjà suffisamment difficile en soi, il est inutile de le rendre plus difficile encore en y intégrant le temps passé en réunions ou en congés. L'équipe utilise donc une unité de temps plus abstraite, que nous appellerons ici «point», dont la définition exacte n'a finalement pas grande importance. En effet, comme nous le verrons plus loin, l'essentiel à ce stade de planification est de pouvoir déterminer combien de scénarios peuvent être traités en une itération.

Tout ce que l'on souhaite donc savoir à ce moment, c'est si tel ou tel scénario «coûtera» 1 point, 2 points ou plus. Pour cela, les développeurs tentent de rapprocher le scénario d'autres scénarios similaires qu'ils ont réalisés par le passé. Par exemple, si un scénario décrit un nouveau panneau IHM de définition des préférences utilisateur, et que ce panneau ressemble *grosso modo* à un panneau existant qui avait coûté 2 points, l'équipe peut consi-

dérer que ce nouveau scénario coûtera également 2 points. Une fois que l'estimation est faite, elle est notée sur la fiche du scénario. (voir figure 6-7)

Remarque

Lorsque deux scénarios sont similaires, on peut être tenté de penser que la réalisation du second sera plus rapide du fait de l'expérience acquise par les développeurs et de l'existence d'un exemple concret de réalisation. Les auteurs d'XP déconseillent cependant de se ranger à ce genre de suppositions : si le phénomène devait se révéler exact en pratique, il serait quoi qu'il en soit pris en compte au début du cycle suivant, lorsque les développeurs comptabilisent les estimations des scénarios qu'ils ont implémentés pour améliorer leur base de référence. Nous reviendrons plus loin sur ce point.

Cette démarche pose bien sûr la question de la première séance de planification, lorsque l'équipe ne dispose pas encore de scénarios de référence sur lesquels elle pourrait caler ses estimations. La solution dans ce cas consiste à baser les «points» de ces estimations sur un équivalent calendaire, en ce que XP appelle un «temps idéal». Ce critère de temps correspond à un état où les développeurs peuvent se consacrer pleinement à la programmation, sans interruptions extérieures d'aucune sorte. Il est souvent plus aisé pour les développeurs de se projeter dans ce temps idéal que dans un temps réel où les inconnues sont plus nombreuses. Les développeurs peuvent alors prendre pour «point» une semaine de ce temps idéal[1] et n'adopter les points abstraits que plus tard dans le projet.

Lever les inconnues techniques

Il peut arriver que certains scénarios soient difficiles à estimer en raison du manque d'expérience des développeurs sur des points techniques précis, en particulier en début de projet. Il est alors conseillé d'interrompre la séance d'estimation et de laisser les développeurs explorer le sujet à travers des prototypages rapides (*spikes*), que l'on peut assimiler à des «carottages» de la technique en question.

Le cas se présente par exemple lorsque le scénario fait intervenir un composant IHM complexe, et que la facilité de personnalisation offerte par les outils employés a un impact majeur sur le temps de mise au point. Dans ce cas, les développeurs peuvent essayer de créer rapidement un prototype qui mette en œuvre ce composant, pour se faire une idée plus précise de la difficulté de la tâche.

La phase d'exploration se prête également à la définition des axes structurants en matière d'architecture technique. Par exemple, dans le cas d'une application Web, si l'on doit faire face à une problématique de montée en charge, on s'orientera vers du développement à base

1. Ou une journée pour des projets durant moins de quelques mois ; cela dépend de la durée totale du projet et de la taille moyenne des scénarios client.

de composants dits «sans état». Ces choix d'architecture seront définis par l'équipe, qui pourra se faire assister d'un consultant selon les conditions décrites au chapitre 2.

Les prototypes créés durant cette phase ont rarement le niveau de qualité attendu par l'équipe pour son application. Pour cette raison, ils sont abandonnés aussitôt que l'équipe en a tiré profit.

Scission/fusion de scénarios

Au cours de l'évaluation, certains scénarios peuvent présenter une granularité trop grosse, correspondant à plusieurs semaines de travail. Or, plus la granularité d'un scénario est grosse, plus son estimation est incertaine, et plus sa planification est contraignante. Pour y remédier, les programmeurs peuvent demander au client de scinder les scénarios les plus gros en scénarios de taille inférieure, plus faciles à estimer.

De même, le client peut souhaiter que des scénarios à la granularité trop fine (c'est-à-dire nettement inférieure à un point) soient fusionnés, de façon qu'il ne lui incombe pas une «pénalité» suite au manque de précision de ses estimations. Trois scénarios clients, chacun inférieur à un point, «valent» trois points sur le plan de livraison ; à moins, précisément, de les réunir en un seul, qui peut alors ne représenter qu'un ou deux points.

Durée de la phase d'exploration

La phase d'exploration se poursuit jusqu'à ce que le client considère avoir couvert l'ensemble de ses besoins, et que tous ses scénarios sont estimés. Cela peut durer plusieurs jours au tout début du projet, et quelques heures seulement plus avant dans le projet, lorsqu'il ne s'agit plus que d'ajouter ou retirer quelques scénarios à l'ensemble existant.

La phase d'exploration n'a pas besoin d'être «complète» au sens où le client a entièrement exprimé ses besoins ; il convient cependant qu'elle soit la plus complète possible si on souhaite avoir une maîtrise de la planification sur le long terme. Inversement, pour un projet plus «exploratoire» par nature, tel un projet de Recherche & Développement, la phase d'exploration peut être confondue avec la première itération ! C'est le résultat de cette dernière qui déterminera la direction qui sera suivie sur le reste du projet.

La phase d'engagement

Au terme de la phase d'exploration, l'équipe dispose d'une série de scénarios client identifiés et estimés. On procède alors à une séance d'engagement de quelques heures seulement, dont le but est de répartir ces scénarios parmi les livraisons à venir pour établir le plan de livraisons.

La figure 6-8 présente les différentes activités relatives à cette séance.

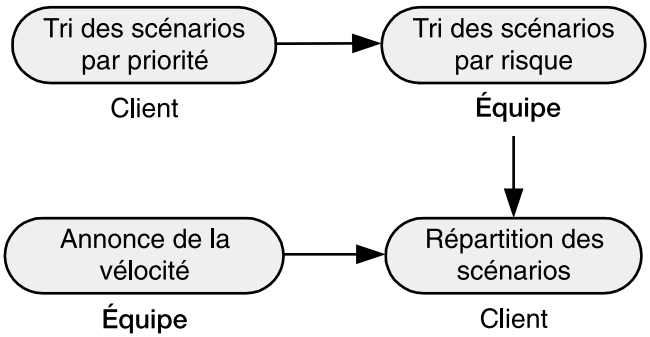

Figure 6-8. La phase d'engagement du cycle des livraisons

Tri des scénarios

Au début de la séance, le client trie les fiches de scénarios en trois tas distincts, correspondant à trois degrés différents de priorité du point de vue fonctionnel. Ensuite, les développeurs trient à leur tour chacun de ces tas en trois nouveaux tas en fonction du risque technique qu'ils représentent. La figure 6-9 illustre le résultat de ce double tri.

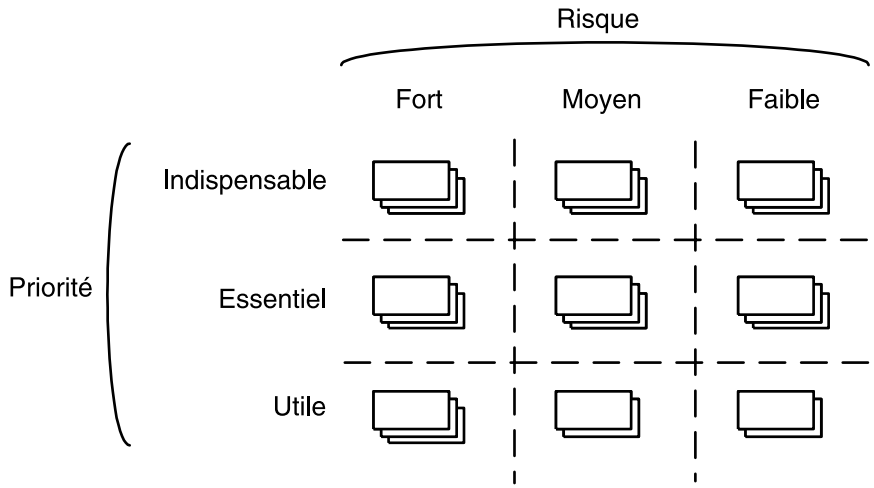

Figure 6-9. Tri des scénarios client par priorité et par risque

En adoptant une telle répartition, le client est armé pour choisir ses compromis «intérêt fonctionnel/intérêt technique» en connaissance de cause.

Remarque

Une interprétation plus légère d'XP tend à considérer le facteur «risque» comme redondant avec les estimations de coûts fournies par les développeurs. Si un programmeur considère un scénario client donné comme particulièrement risqué, son réflexe sera de «se couvrir» en fournissant une estimation élevée. Dans ce cas, seules deux choses peuvent réduire l'appréciation du risque : ou bien la réalisation d'un prototype pour lever les incertitudes, ou bien la scission du scénario. Dans la pratique, le prototype suggèrera presque certainement une scission. On peut donc se contenter de trier les scénarios par ordre de valeur.

Annonce de la vélocité de l'équipe

Avant de procéder à la répartition des scénarios, il ne reste au client qu'à savoir combien de scénarios il pourra placer dans chaque itération, ou plus exactement le nombre de points qu'il pourra «dépenser» à chaque itération pour «acheter» des scénarios.

À cette fin, XP définit la notion de vélocité.

Définition

La vélocité de l'équipe correspond au total des estimations en points des scénarios qui ont été totalement et correctement implémentés au cours de l'itération précédente.

Corollaire

La vélocité de l'équipe correspond donc au nombre de points de scénarios client que l'équipe peut traiter en une itération.

Aucun ajustement subjectif n'est effectué lors du calcul de la vélocité. On pourrait prendre en compte par exemple l'absence de certains membres de l'équipe ou un changement dans la durée des itérations (l'équipe peut souhaiter expérimenter une durée différente), mais ce n'est pas recommandé : même lors de cas où une variation de la vélocité est «prévisible», il est conseillé de la calculer en respectant strictement la définition usuelle, pour s'en tenir à un mécanisme aussi simple que possible.

En revanche, cette technique d'évaluation pose encore le problème de la première itération, au tout début du projet. En l'absence d'antécédents, le coach n'a d'autre choix que de fournir sa propre estimation de la vélocité de l'équipe.

Il peut pour ce faire s'appuyer sur quelques calculs. Supposons par exemple que l'équipe ait opté pour des itérations de 2 semaines, et des points tels qu'une semaine de temps idéal pour un binôme représente 2 points de scénarios client. En faisant la supposition plutôt pessimiste que les développeurs ne passent que 50 % de leur temps à programmer, le reste étant

consommé en réunions ou activités annexes, on peut considérer que chaque binôme ne vit qu'une semaine de temps idéal par itération, et donc qu'il pourra traiter 2 points de scénarios clients par itération. Si l'équipe comporte 6 développeurs, soit 3 binômes, elle pourra traiter 6 points par itération et affichera donc une vélocité initiale de 6.

Mise en garde

Il faut veiller à ne pas considérer cette vélocité comme un indicateur de productivité de l'équipe ; il s'agit plutôt d'un indicateur de sa granularité d'estimation. Une même fonctionnalité peut être implémentée dans le même temps par deux équipes, l'une annonçant une vélocité de 6 et ayant estimé le scénario à 3 points, et l'autre annonçant une vélocité de 12 et ayant estimé le scénario à 6 points. Néanmoins, si la vélocité n'est en aucun cas un indicateur *absolu* de performances, elle est utile en tant qu'indicateur relatif. Lorsque la vélocité chute soudainement lors d'une itération, l'équipe qui n'utiliserait cette information que pour planifier la suite, sans chercher à expliquer ce changement et à y remédier, ferait preuve de négligence.

Répartition des scénarios

Connaissant la vélocité de l'équipe, le client n'a donc plus qu'à choisir, parmi les 9 groupes de scénarios, un nombre de scénarios dont la somme des coûts en points est égale à ce total, et ce, itération après itération. À lui donc de choisir ses scénarios en s'assurant que ceux qui lui semblent les plus importants sont bien traités au plus tôt.

Notre XPérience

Cette phase d'engagement représente toujours un moment clé dans les relations entre l'équipe et le client, puisque c'est lors de cette phase que ce dernier devra peut-être accepter que les livraisons contiennent moins de fonctionnalités qu'il ne l'espérait. Le rôle du coach à ce stade est primordial ; c'est à lui de faire respecter l'équilibre client/développeurs que nous avons évoqué au début de ce chapitre.

Remarque

Selon ses besoins, le client peut procéder à la répartition de tous les scénarios sur l'ensemble des livraisons à venir, et obtenir ainsi un plan complet du projet, ou bien se concentrer uniquement sur la livraison suivante. Lorsqu'on dispose de suffisamment de scénarios pour planifier l'ensemble du projet, il est préférable de le faire – à condition toutefois de garder à l'esprit qu'un tel plan d'ensemble n'est établi qu'à titre provisoire.

La phase de pilotage

La répartition des scénarios établie lors de la phase précédente constitue le plan de livraisons du projet, ou plus exactement son plan de livraisons *provisoire* – il y a bien peu de chances pour qu'il reste inchangé jusqu'au bout. L'équipe est alors prête à se lancer dans la réalisation

des scénarios, à travers une phase de pilotage gérée par le mécanisme de planification d'itérations détaillé plus bas.

Cependant, deux mécanismes particuliers de suivi et de contrôle des itérations sont intéressants à ce niveau de granularité : le suivi des tests de recette, et la gestion des défauts.

Suivi des tests de recette

L'évolution du nombre de tests de recette réalisés par le client et les testeurs, ainsi que la proportion de tests validés, fournissent un indicateur concret de l'avancement du projet. La figure 6-10 montre le type de graphe qu'il est possible d'établir pour visualiser cet indicateur.

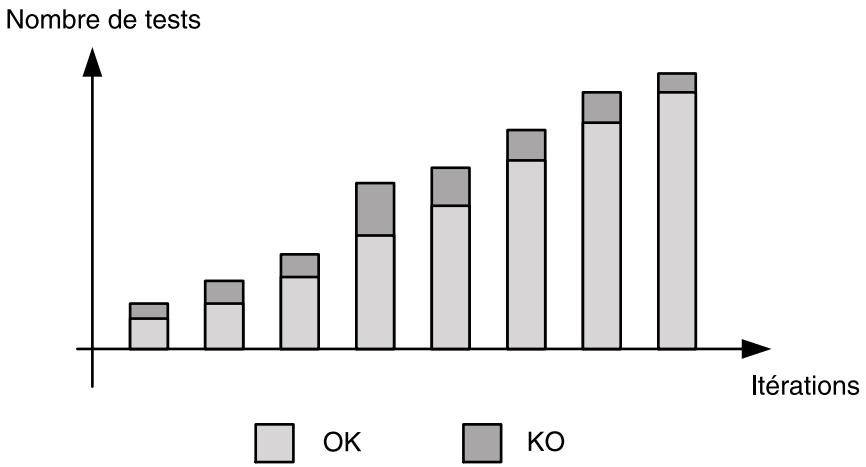

Figure 6-10. Graphe d'évolution des tests de recette

Ce type de graphe peut être mis à jour régulièrement par le coach, et être tenu à disposition du manager et du client afin qu'ils puissent avoir à tout moment une idée de l'avancement du projet.

Gestion des défauts et de leurs corrections

En dépit des efforts prodigués par l'équipe pour garantir la qualité du produit, celui-ci présentera tôt ou tard des défauts – rappelons ici que le mot «bogue» est exclu du vocabulaire XP. Comment intégrer leur correction dans la planification du projet ?

La solution proposée par XP pour parer à ce problème est «la plus simple qui puisse marcher» : transformer les corrections en nouveaux scénarios client ou corriger un scénario existant, de façon à intégrer les corrections au mécanisme général de planification. Ces éléments seront parfois difficiles à estimer, mais les développeurs peuvent passer un peu de leur temps à explorer le problème pour tenter d'en fournir une estimation grossière.

Une autre solution – plus stressante mais préférable par bien des aspects – consiste à considérer que la résolution d'un défaut, d'une anomalie ou d'un dysfonctionnement représente une priorité absolue sur toutes les autres modifications. Cette « garantie de qualité » pourra être mise en œuvre par des équipes rodées à XP, pour qui l'apparition d'un défaut non décelé par les tests unitaires ou de recette sera un événement rare.

Notre XPérience

Quel que soit le mode de gestion adopté, il est une pratique *vitale* qui doit être mise en place par toute équipe XP : lorsqu'un défaut est décelé, on écrira un test unitaire *avant* de le corriger. Le test échouera parce que le défaut est présent ; par conséquent, on aura la certitude d'avoir résolu le problème lorsque le test passera. À défaut d'adopter cette démarche, on constatera presque à coup sûr une dégradation de la qualité : les défauts s'accumulent là où les tests automatiques ne sont pas passés. Un défaut signale que les tests existants sont insuffisants : si on ignore cette information, c'est à ses risques et périls.

Fin du cycle, début d'un nouveau cycle

La phase de pilotage s'arrête au moment de la livraison proprement dite, à la fin de la dernière itération prévue pour ce cycle de livraison. Ainsi que cela a été expliqué dans la première partie de ce chapitre, la livraison est effectuée même si les scénarios prévus n'ont pu être implémentés dans leur totalité, puisqu'une équipe XP évite de jouer sur la variable « délais » du projet. De toute manière, le système de planification des scénarios assure que, dans le pire des cas, les scénarios éventuellement reportés seront les moins importants de la livraison.

Une fois la livraison effectuée, l'équipe se doit de célébrer l'événement, par exemple autour d'un bon repas hors du lieu de travail. Ce sera une bonne occasion pour se détendre et prendre un peu de recul vis-à-vis du projet.

Ensuite, le projet repart sur un nouveau cycle de livraison. Mais cette fois, l'équipe se trouve beaucoup mieux armée pour le piloter :

- D'une part, elle dispose d'une base de scénarios déjà implémentés et d'un logiciel en état de marche. Elle peut donc alors mettre à jour les fiches de tous les scénarios implémentés pour y noter leur durée effective de réalisation, et réévaluer tous les scénarios restants.
- D'autre part, elle dispose d'un certain nombre d'itérations de référence, qui lui donnent une mesure plus réaliste de sa vélocité.

Livraison après livraison, l'équipe améliore ainsi la fiabilité de ses estimations. En outre, elle gère ses séances de planification de plus en plus rapidement, puisqu'elle acquiert la maîtrise du processus et de son produit… et qu'il y a moins de scénarios.

Fin du projet

Les cycles de livraisons peuvent ainsi se poursuivre longtemps. En définitive, c'est au client de décider du moment où le projet doit se terminer, le plus souvent lorsqu'il juge que les scénarios restants ne méritent pas un tel investissement.

Planification d'itération

Si le cycle des livraisons porte sur la planification des fonctionnalités, celui des itérations se rapproche de la réalisation proprement dite et porte sur la planification des activités menées par les développeurs.

Le plan d'itération porte donc sur des *tâches*, définies par les développeurs selon des considérations techniques, et dont la granularité est telle qu'un binôme peut réaliser une tâche en une ou deux journées. Les tâches sont soumises à des traitements similaires à ceux des scénarios client au cours des trois phases d'une itération :

- Elles sont *définies* au cours de la phase d'exploration.
- Elles sont *affectées* au cours de la phase d'engagement.
- Elles sont *réalisées* au cours de la phase de pilotage.

La phase d'exploration

Au début de chaque itération, le client et les programmeurs se réunissent pour analyser les scénarios client de l'itération et en déduire les tâches à réaliser. Les programmeurs posent toutes les questions qu'ils souhaitent au client et discutent en sa présence de la façon d'établir une décomposition du scénario en tâches. Cette activité est souvent indissociable d'une activité de conception, même superficielle, qui permet aux programmeurs de se faire une idée plus précise du travail correspondant. Pour cela, ils peuvent si nécessaire utiliser la technique des cartes CRC présentée au chapitre 5.

À la différence de la phase d'exploration du cycle des livraisons, celle des itérations n'intègre pas l'estimation des tâches qui y sont définies. Et ce, parce que l'estimation d'une tâche est confiée au développeur qui se l'approprie, et parce que cela n'est décidé que lors de la phase d'engagement.

Du scénario à la tâche

Une illustration par l'exemple s'impose : on peut être en présence du scénario « L'utilisateur peut conserver les adresses de ses sites préférés » ; les tâches correspondantes dépendront de ce qui est déjà implémenté, par exemple « Créer une classe SitePréféré », « Ajouter une option au menu Adresses », etc. Individuellement, les tâches n'ont pas de valeur pour le client XP et ne contribuent pas à l'élaboration de ses besoins : si c'était le cas, ce seraient des scénarios client.

En revanche, la décomposition en tâches permet au client de constater que les programmeurs ont bien compris ce dont il s'agit, et éventuellement de les reprendre sur tel ou tel point qu'ils ont mal appréhendé.

La phase d'engagement

La phase d'engagement a pour objet de définir le plan d'itération, sur lequel figurent les tâches, les développeurs qui en sont responsables et les estimations qu'ils en ont donné.

Au cours de cette séance, les développeurs se répartissent les tâches une à une, comme ils le souhaitent, en se conformant toutefois au processus schématisé figure 6-11 et présenté dans les paragraphes suivants.

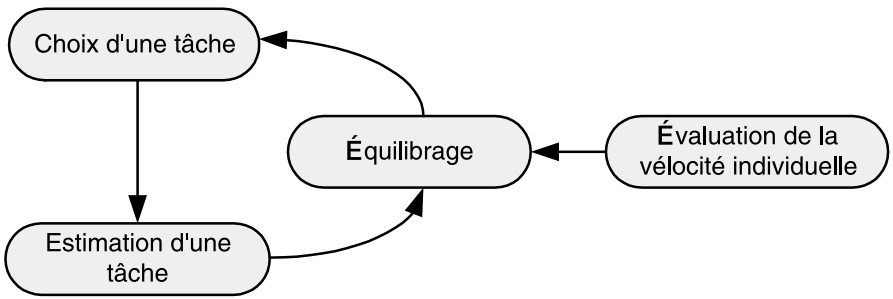

Figure 6-11. La phase d'engagement du cycle des itérations

Choix et estimation d'une tâche

Chaque développeur se porte volontaire pour les tâches qu'il souhaite réaliser. Il sera ensuite seul responsable de l'exécution de la tâche, même si la programmation est toujours menée en binôme.

Remarque

Le fait de se porter volontaire pour *n* tâches est la seule partie «officielle» et indispensable de la phase d'engagement. L'équilibrage peut être réalisé de façon informelle et se fera quoi qu'il arrive. Une équipe XP peut donc ne pas tenir compte des conseils qui suivent et qui décrivent des aspects «pointus» de la phase d'engagement.

Lorsqu'un développeur choisit une tâche, il commence par en donner une estimation. L'unité utilisée ici n'est plus un point abstrait, mais une journée du «temps idéal» que nous avons défini plus haut.

La technique d'estimation est voisine de celle des scénarios client : le développeur essaye de rapprocher la tâche en question d'une autre qu'il a précédemment réalisée. Au tout début du projet, il ne peut compter que sur son expérience et son bon sens, mais après quelques itérations il disposera d'un référentiel lui permettant d'établir plus rapidement des estimations fiables.

Lors de cette séance d'engagement, l'idéal est que les développeurs disposent d'un grand tableau sur lequel ils notent, pour chaque tâche, les initiales du développeur qui en est responsable et l'estimation qu'il en a donné, comme cela est illustré à la figure 6-12.

Tâche	Qui	Prévu
☐ Sauvegarde/chargement des préférences utilisateur	AB	2
☐ Impression des raccourcis clavier (+ commande dans menu Options)	CD	2
☐ Operation IDL d'accès au nombre de clients connectés	AB	1
...	...	...

Figure 6-12. Exemple de plan d'itération

Ainsi présenté le plan d'itération est visible de tous ; les développeurs peuvent s'y référer et le mettre à jour librement en cours d'itération.

Équilibrage des charges

Lorsqu'ils choisissent leurs tâches, les développeurs doivent s'assurer que la charge est uniformément répartie. Au besoin, les développeurs les plus sollicités peuvent se défaire de certaines tâches, qui seront reprises par les développeurs les moins «chargés».

Les développeurs peuvent également souhaiter vérifier qu'ils ne prennent pas plus de tâches qu'ils ne pourront en traiter réellement dans l'itération. Par exemple, sachant que les développeurs travaillent en binôme, et qu'ils ne peuvent donc s'occuper de leurs propres tâches que pour 50 % de leur temps, si l'équipe utilise des itérations de deux semaines, les développeurs devront éviter de prendre des tâches qui excèdent cinq jours de temps idéal.

Remarque

Certaines équipes XP équilibrent la charge de travail des programmeurs en introduisant la notion de *vélocité individuelle*, voisine de la notion de vélocité utilisée par l'équipe. Cette mécanique nous semble cependant alourdir inutilement le mécanisme de planification des itérations.

Scission/fusion de tâches

Il peut arriver que des tâches aient une durée trop courte ou trop longue, ce qui rend difficile leur estimation ou leur planification. Dans ce genre de situation, les développeurs peuvent fusionner les petites tâches entre elles, ou au contraire scinder les tâches trop longues en tâches dont la durée est raisonnable.

Ajout/suppression de scénarios client

Comme les estimations des tâches se basent sur une analyse plus précise que pour celles menées dans le cadre des scénarios utilisateurs, il peut arriver que la charge finale de l'équipe en termes de tâches ne corresponde pas à son potentiel sur l'itération courante.

Par exemple, dans le cas où l'équipe utilise des itérations de deux semaines, on peut aboutir à une situation dans laquelle tous les développeurs ont sélectionné des tâches pour cinq jours de temps idéal, alors qu'il leur reste encore des tâches à répartir. Dans ce cas, ils doivent se tourner vers le client et lui demander de retirer un scénario. Rappelons en effet que ces mécanismes ont pour objet de limiter la surcharge des développeurs afin de leur assurer un rythme de travail optimal.

Inversement, il arrive également qu'au terme de la répartition, certains développeurs soient «sous-chargés». Dans ce cas, l'équipe se tourne également vers le client pour lui demander d'intégrer un scénario supplémentaire à l'itération.

La phase de pilotage

Une fois que le plan d'itération est établi, les développeurs se lancent dans la réalisation de leurs tâches. Comme nous l'avons expliqué au chapitre 5 consacré à l'organisation de l'équipe, les développeurs choisissent eux-mêmes leurs binômes, et c'est à eux d'équilibrer, d'une part, le temps qu'ils passent sur leurs propres tâches et, d'autre part, le temps passé à aider d'autres personnes. Ils sont également libres d'en définir l'ordre de réalisation.

Lorsqu'ils terminent une tâche, les développeurs se reportent au tableau sur lequel figure le plan d'itération (voir figure 6-12) pour l'annoter en conséquence. Mais le suivi des tâches en cours d'itération ne s'arrête pas là, et c'est ici qu'intervient le rôle du *tracker* introduit au chapitre 2.

Le rôle du tracker

Au moins deux fois par semaine, le tracker fait le tour des développeurs pour obtenir un instantané de l'avancement de l'itération. Il pose à chacun les deux questions suivantes :

- combien de temps a été passé sur chaque tâche ;
- combien de temps le programmeur pense devoir encore passer sur chaque tâche.

Le seul objectif de cette démarche est de déceler tout dérapage au plus tôt. Si un développeur a pris du retard sur la réalisation de ses tâches, le tracker se contente simplement de le signaler afin que le coach et le reste de l'équipe puissent prendre les mesures nécessaires et alléger la charge du développeur. L'équipe dispose de plusieurs options dans ce genre de situation. Un brainstorming permet souvent d'identifier une technique permettant de simplifier la solution. Sinon, au moyen de quelques réorganisations, on peut espérer sortir la tâche de l'ornière, soit en associant le développeur à un binôme plus aguerri sur le sujet technique qui «coince», soit en confiant certaines de ses tâches à d'autres développeurs moins chargés. Parfois (notamment lorsqu'il s'agit d'un défaut introuvable), le mieux est de changer complètement le binôme. C'est une pratique XP dont il ne faut pas s'offenser. Et si le glissement est irrémédiable et que l'ensemble de l'équipe est trop chargée, le coach peut s'adresser au client pour lui demander de retirer un scénario de l'itération.

Le cas de figure inverse – un développeur qui termine ses tâches en avance – peut également se produire. Le développeur aidera alors sans doute ses collègues, mais si la situation se généralise le coach peut demander au client d'ajouter un scénario à l'itération en cours.

Les stand-up meetings

Une fois par jour, de préférence le matin, le coach réunit toute l'équipe pour faire un point d'une dizaine de minutes environ. La brièveté de cette réunion est assurée en partie par le fait que tous les intervenants se tiennent debout, d'où le nom prêté à cette technique.

Au cours de cette réunion, chaque développeur fait un point rapide sur sa situation et les éventuelles difficultés qu'il rencontre. Si l'intervention d'autres développeurs est requise, le coach

s'assure que le débat ne s'instaure pas au cours de cette réunion mais que les développeurs concernés se réuniront juste après. Le *stand-up meeting* ne doit en effet pas devenir une réunion technique, ni (en particulier si l'équipe possède un tracker) un suivi des tâches. C'est surtout l'occasion d'exprimer ses besoins : binômer avec une personne particulière, faire une petite réunion de conception, obtenir de l'information (du coach, du client…), et donc d'organiser la journée.

Fin de l'itération

En dehors des livraisons officielles, les fins d'itérations donnent lieu à une mini-livraison du logiciel au client, qui peut ainsi s'assurer que tous les tests de recette qu'il avait définis passent avec succès.

Par ailleurs, les développeurs mettent à jour les coûts des scénarios qu'ils ont réalisés, en comparant la somme des durées des tâches correspondantes à celles d'autres scénarios précédemment réalisés. Cela permet de construire une base de coûts de scénarios fiable pour les estimations à venir.

Comme tout événement d'un projet XP, la fin de l'itération se célèbre : boissons et gâteaux sont donc à l'ordre du jour !

Deuxième partie

L'Extreme Programming facteur de compétitivité des entreprises

Si l'Extreme Programming offre une réponse aux inconvénients et limites d'approches traditionnelles de gestion de projet informatique, il n'en reste pas moins que le passage à XP peut être perçu au premier abord comme un risque pour les managers de l'entreprise. La nature même des pratiques (à l'image de la programmation en binôme) invite instinctivement à la prudence. Par ailleurs, outre qu'il n'est pas aisé de mesurer le réel retour sur investissement d'un passage à XP, la complexité des aspects juridiques inhérents à l'Extreme Programming (comme l'acceptation du changement) et le décalage – tout apparent – d'XP par rapport aux plans qualité mis en œuvre ont de quoi tempérer bien des ardeurs.

Pour couper court à certaines idées reçues, nous montrerons dans les chapitres suivants comment l'Extreme Programming doit s'inscrire dans un projet d'entreprise, aussi bien du point de vue opérationnel que financier et juridique.

Tout d'abord, le chapitre 7 est consacré à la problématique de la transition vers XP. Quelles conditions faut-il réunir pour que la méthode porte ses fruits ? Comment gérer la transition ? Par quelles pratiques commencer ? Comment se former ? D'une manière plus générale, ce chapitre pose la question de l'impact d'XP sur la culture existante de l'entreprise.

Le chapitre 8 aborde la difficile question du retour sur investissement. Les études sur le ROI invitent souvent à la méfiance, car il est assez facile de faire parler les chiffres dans un sens ou dans un autre. Pour éviter ce genre de dérive, nous avons adopté une démarche qui soit le plus objective possible. En identifiant les centres de coût et en montrant sans détours comment XP peut avoir un impact favorable sur ces coûts, étude de cas à l'appui, le chapitre s'efforce de donner une vision aussi claire que possible du retour sur investissement possible.

Le chapitre 9 est consacré aux aspects contractuels. Tout projet informatique s'inscrit dans une relation client-fournisseur généralement étayée par un contrat. Le chapitre dresse un panorama des différents types de contrat, en présentant la compatibilité de l'Extreme Programming avec chacun d'eux.

Le chapitre 10 permet quant à lui de mieux comprendre comment une démarche XP peut s'inscrire dans un plan d'assurance qualité. On s'intéressera en particulier au positionnement d'XP par rapport à l'ISO 9000, mais aussi par rapport à des méthodes telles que Rational Unified Process. Le chapitre propose également une mise en perspective d'XP par rapport à d'autres méthodes agiles telles que Crystal, Scrum, ASD, FDD, DSDM, Pragmatic Programming.

7

Plan d'accès et formation

Vous voulez savoir comment peindre un tableau parfait ? C'est facile.
Devenez parfait, et ensuite peignez tout naturellement.

– Robert Pirsig

C'est une idée contraire aux tendances de l'éducation moderne que d'apprendre aux enfants à programmer. Quel plaisir pourrait-on trouver à planifier des activités, acquérir la discipline qui permet d'organiser la pensée, prêter attention aux détails, et apprendre à faire son autocritique ?

– Alan Perlis

Les pratiques présentées dans les chapitres précédents promettent des résultats ambitieux : un code clair, toujours ouvert au changement et consolidé par une batterie complète de tests automatiques ; une équipe soudée, bien organisée, composée de développeurs motivés et polyvalents ; un projet apte à suivre tous les changements de cap demandés par le client, tout en proposant un pilotage de plus en plus fiable… Tout cela peut paraître un peu utopique – voire suspect – à quelqu'un qui connaît la réalité actuelle des projets de développement, et, quoi qu'il en soit, il est clair que toutes les équipes qui adoptent XP ne parviennent pas automatiquement à ce niveau d'excellence.

Deux grands types de questions se posent quand on envisage d'appliquer XP sur un projet donné. En premier lieu, quels sont donc les facteurs qui conditionnent le succès d'un projet XP ? Le projet en question présente-t-il le bon «profil» ? En second lieu, comment s'y prendre pour mettre la méthode en place ? Quelles questions convient-il de se poser ? Comment se former ? Par quelles pratiques commencer ?

Les facteurs de succès d'un projet XP

La présentation des pratiques XP fait déjà entrevoir les caractéristiques du projet XP idéal : une équipe de taille réduite (typiquement moins de dix personnes), qui développe un nouveau produit pour un client bien identifié et prêt à s'investir dans le projet, le tout dans un environnement technique qui permet une structuration progressive du code et des évolutions rapides.

Cependant, ces conditions initiales ne sont ni nécessaires ni suffisantes. Nous avons par exemple pu tirer concrètement parti de l'utilisation d'XP dans des domaines industriels dits «lourds», au prix certes de quelques aménagements dans la mise en œuvre de la méthode. Martin Fowler, auteur du livre de référence sur le remaniement, a mis en place la méthode sur un projet qui compte une centaine de développeurs.

Il ne faut donc pas s'attendre à trouver ici une sorte de *checklist* des conditions à remplir avant de démarrer le projet. Comme il en va pour tout autre type de projet, la réussite d'un projet XP dépend d'une multitude de facteurs, et nous nous bornerons ici à en articuler les principales caractéristiques.

Présence d'un «champion de la cause XP»

Tout d'abord, l'équipe doit avoir une compréhension très fine de la méthode afin d'atteindre deux buts complémentaires. Le premier est d'appliquer les différentes pratiques avec succès, le second est de convaincre son entourage – en particulier la direction du projet et/ou le client – de jouer le jeu. Il est bien sûr possible de se lancer avec une connaissance superficielle de la méthode, mais il y a alors fort à parier qu'aux premiers signes de difficulté, équipe et client reviendront par un réflexe naturel aux anciennes habitudes plutôt que d'essayer de progresser au moyen d'XP.

Dans la pratique, nous avons constaté que ce double rôle était le plus souvent pris en charge par une même personne, quelqu'un qui avait à la fois une bonne connaissance de la méthode et une confiance suffisante pour «convertir» son entourage. XP propose d'ailleurs un rôle particulier pour ce personnage : celui de coach. Cela tient peut-être à ce que la méthode en est encore à ses débuts mais, compte tenu des efforts à déployer pour assumer correctement ces deux missions, il nous semble que la personnalité du coach reste aujourd'hui un facteur déterminant dans une mise en place réussie d'XP.

Quelques développeurs expérimentés pour guider la réalisation

Nous avons beaucoup insisté dans les chapitres précédents sur l'importance du code lui-même en tant que facteur de succès des autres pratiques. Nous pensons que l'équipe doit au moins disposer d'un ou deux développeurs suffisamment expérimentés qui puissent aider le reste de l'équipe à donner à l'application le niveau de qualité et de souplesse requis.

Dotés d'une bonne maîtrise des techniques de programmation et de conception, ces développeurs doivent être sensibles à la propreté du code, et ils doivent avoir la capacité de faire émerger une conception simple au cours du développement. Cela nécessite une tournure d'esprit particulière : ils doivent pouvoir penser globalement, afin d'améliorer progressivement la structure d'ensemble du logiciel plutôt que de la laisser se dégrader en permettant des «entorses» locales répétées à la règle.

Cette dernière remarque vaut pour l'ensemble des développeurs du projet. Chacun doit se plier aux exigences strictes d'XP en matière de programmation, quitte à faire les efforts nécessaires pour combler d'éventuelles lacunes personnelles.

Si nous soulignons l'influence particulière de quelques membres de l'équipe, il n'est pas du tout question d'attribuer des rôles précis à certains développeurs. La conception et la programmation restent des activités collectives dans XP, aussi ces développeurs expérimentés doivent-ils savoir aider sans diriger, et surtout ne pas chercher à revendiquer la responsabilité de l'architecture mise en place – une qualité parfois désignée par le terme *egoless programming* – la programmation sans ego.

Un contexte technique adéquat

XP suppose des cycles de développement/compilation/test très courts. Si ce n'est pas le cas, par exemple lorsque l'équipe travaille sur une application volumineuse qui n'a pas été conçue selon les préceptes XP, l'équipe doit revoir le découpage de l'application et la gestion des dépendances entre modules afin de pouvoir travailler sur des portions réduites de l'application sans qu'il faille systématiquement tout recompiler. Il peut également être envisagé d'acquérir du matériel ou des logiciels (compilateurs, etc.) plus performants.

Le langage de programmation utilisé doit offrir des mécanismes efficaces d'abstraction et de factorisation, pour permettre en particulier l'application du principe de *Once and Only Once*. Les langages orientés objet en vogue aujourd'hui supportent pour la plupart le traitement ; en revanche, il reste à déterminer s'il est possible de pratiquer l'XP avec des langages de type assembleur ou BASIC lorsque l'application devient volumineuse.

Enfin, l'équipe doit disposer d'un outil de gestion de configuration efficace, à la fois pour synchroniser les développements parallèles et pour être à même de revenir à une version stable du logiciel dans les rares cas où un binôme s'empêtrerait dans un développement difficile ou un refactoring ambitieux. Comme nous l'avons évoqué au chapitre 5, consacré à l'organisation de l'équipe, cet outil doit être adapté à une situation qui verrait tous les développeurs travailler simultanément sur *les mêmes* fichiers. Les mécanismes de verrous imposés par certains outils sont susceptibles d'entraver le travail d'une équipe XP ; mieux vaut donc un outil capable de gérer efficacement la fusion des versions au moment de l'intégration.

Une équipe apte au travail… d'équipe

En amenant tous les membres de l'équipe à travailler réellement ensemble, XP peut révéler certaines incompatibilités de caractère qui auraient pu rester occultées dans un mode de travail plus classique. Étant donné l'importance de la dimension collective du travail dans XP, ces problèmes ne peuvent être ignorés : la composition de l'équipe doit donc être effectuée en privilégiant les qualités humaines des candidats.

Le trait de caractère qu'il faut particulièrement éviter dans une équipe XP est certainement le «nombrilisme». Des développeurs talentueux mais égocentriques peuvent paralyser le fonctionnement du reste de l'équipe en rendant difficile le travail en binôme, ou encore en rendant plus rigide le processus de répartition des tâches. Cela est également vrai pour le coach lui-même, qui doit faire preuve d'ouverture et laisser une grande autonomie à l'équipe. En outre, la culture de l'entreprise – plus précisément, le mode d'évaluation et de récompense des employés – doit favoriser le travail collectif plutôt que les comportements individualistes.

Un environnement de travail adapté

XP joue beaucoup sur des échanges d'informations informels et continus entre les membres de l'équipe. Pour que cela soit possible, ces derniers doivent travailler à proximité immédiate les uns des autres. À notre sens, une séparation géographique des membres de l'équipe (par exemple, sur plusieurs étages d'un même bâtiment) peut représenter un obstacle majeur au bon fonctionnement d'une équipe XP.

La disposition des bureaux a également un impact important sur le fonctionnement de l'équipe. Nous vous renvoyons pour ce point précis à la description de l'environnement XP idéal donnée au chapitre 5.

Des décideurs convaincus, ou au moins consentants

Les facteurs de succès présentés dans les paragraphes précédents portent sur l'équipe elle-même et sur son fonctionnement interne. Toutefois, même dans le cas où une équipe parviendrait à réunir toutes ces conditions, il reste à savoir comment elle pourra appliquer la méthode dans le contexte plus large de l'entreprise.

De ce point de vue, toutes les pratiques XP qui exigent une implication forte du client peuvent sembler délicates à mettre en place dans un contexte «standard». Cependant, nous avons vu au chapitre 2 que le rôle de client XP ne devait pas être nécessairement tenu par le client «final» – celui qui finance le projet – mais qu'il pouvait parfaitement revenir à une personne dotée d'une bonne connaissance du domaine fonctionnel et habilitée à prendre des décisions relatives à la maîtrise d'ouvrage du projet.

Ce contournement apporte une certaine souplesse dans l'introduction de la méthode, mais n'exonère en aucun cas l'équipe de ses dépendances avec l'extérieur. En effet, qu'il s'agisse

de l'approche incrémentale adoptée pour les spécifications ou de la régulation du rythme de travail par remaniement du contenu fonctionnel des livraisons, une équipe ne peut adopter une démarche XP sans que les autres acteurs du projet – le client final, la direction, et les équipes qui en dépendent –, ne soient concernés d'une façon ou d'une autre.

La culture générale de l'entreprise reste donc un facteur important de réussite du projet, et une équipe aura du mal à tirer de forts bénéfices d'XP dans des environnements dont la culture est radicalement différente, par exemple :

- dans une culture où le mérite se mesure aux heures supplémentaires, et où l'on considère qu'une équipe ne fonctionne à plein rendement que sous une forte pression ;
- dans une culture trop axée sur le jeu politique, où les relations de type gagnant-gagnant ne sont pas de mise ;
- dans une culture trop attachée aux démarches «linéaires», où la qualité du logiciel s'exprime en nombre de documents produits, ou bien lorsque le projet global impose l'utilisation d'outils de modélisation et de génération de code tout au long du développement.

Dans le cas de développements réalisés pour des entités extérieures à l'entreprise, nous verrons au chapitre 9 que les aspects contractuels ne doivent pas être négligés, et que les contrats au forfait se prêtent assez mal à une mise en place complète d'XP.

Toutes ces situations ne sont cependant pas désespérées. Une équipe peut tirer parti des pratiques «internes» d'XP – celles relatives à la programmation et peut-être celles liées à l'organisation de l'équipe – et profiter le cas échéant de ses premiers succès pour gagner la confiance des décideurs du projet et créer progressivement les conditions d'une mise en place plus complète de la méthode.

Une mise en place bien menée

Dès lors qu'une transition vers XP est décidée, la façon dont cette transition est négociée est également déterminante. Il faut en effet s'assurer que tous les intervenants du projet aient bien compris de quoi il retourne, et que les diverses pratiques soient adoptées progressivement et appliquées convenablement. Ce point important fait donc l'objet de la suite de ce chapitre.

Ordre de marche

On ne change pas une équipe qui gagne… ou, pour adapter le proverbe : on ne change pas une méthode qui fonctionne. Si, au sein d'une entreprise, les différents intervenants des projets informatiques sont satisfaits des résultats – les programmeurs sont motivés par leur travail, les projets sont livrés en temps et en heure et répondent pleinement aux besoins des clients ou des utilisateurs, l'entreprise elle-même trouve une rentabilité dans ces projets –, rechercher le

changement pour le changement ne sera pas d'une grande utilité et risque même d'être néfaste.

Dans la plupart des cas, l'intérêt pour une nouvelle méthode – qu'il s'agisse d'XP ou d'une autre méthode, d'ailleurs – sera motivé par la prise de conscience d'un certain nombre de problèmes ou de défauts, soit structurels, soit relatifs à un projet donné. Ces problèmes peuvent avoir trait à la gestion des projets (dépassements des délais, difficultés d'encadrement), à la qualité des projets (nombre important d'anomalies ou d'incidents, inadéquation des produits livrés aux besoins) ou à des facteurs techniques (problèmes d'intégration, de mise à jour, ou de déploiement).

Adopter XP : à l'initiative de qui ?

Encore faut-il que les problèmes soient identifiés – ce qui suppose *a fortiori* que l'on ait conscience que des problèmes existent et qu'un changement est nécessaire pour les résoudre. Par ailleurs, cette prise de conscience fonctionnera d'autant mieux qu'elle est partagée entre les différents acteurs des projets : programmeurs, clients, encadrement.

Il est important de noter que le passage à XP représente en général un véritable changement de culture ; nous reviendrons plus en détail sur ces aspects. Comme tout changement, il peut susciter un certain nombre de résistances, voire de situations de blocages, s'il n'est pas sous-tendu par un dialogue honnête et ouvert, basé sur un respect mutuel.

Nous devons garder à l'esprit ces deux aspects – prise de conscience commune des problèmes, ouverture du dialogue pour installer le changement – afin de rendre compte des deux cas de figure les plus courants, où l'intérêt pour XP provient soit des programmeurs, soit de leur direction.

Programmeurs

Les programmeurs auront naturellement une conscience plus aiguë des problèmes qui les concernent directement : d'une part, ceux qui ont trait à la qualité de leur vie professionnelle et, d'autre part, ceux qui relèvent directement de leur activité. Dans la première catégorie, figurent notamment le stress et les diverses pressions qui peuvent résulter des surcharges de travail dues à des délais trop serrés ou à des retards conséquents, ou le manque de qualités humaines dans l'organisation du travail en équipe. Dans la seconde, on pourra citer le manque de développement des compétences, l'impression de ne pas pouvoir faire un travail de qualité… Les pratiques d'XP qui retiendront l'intérêt des programmeurs seront naturellement en rapport avec ces perceptions : rythme durable, programmation en binôme, tests unitaires…

Les difficultés qui peuvent se poser lors d'un passage à XP à l'initiative des programmeurs viendront de la perception, nécessairement différente, que peuvent avoir de la situation les clients et décideurs. Ces derniers pourront par exemple avoir une vision plus «macroscopique» du projet dans laquelle le respect des délais est un facteur plus important que les aspects relatifs à la qualité ; les bénéfices à retirer des tests unitaires ou fonctionnels

peuvent de ce point de vue apparaître marginaux. La programmation en binôme peut être perçue comme un doublement des coûts – deux personnes travaillant sur une même tâche alors qu'une seule suffirait.

Décideurs

Inversement, un passage vers XP qui s'opère à l'initiative de l'encadrement ou de la direction peut également déboucher sur une impasse : le risque le plus saillant est que ce changement soit perçu, de la part des programmeurs, comme une ingérence dans leur mode de fonctionnement. Les problèmes que perçoivent généralement les responsables d'équipes techniques auront plus souvent trait au respect des délais, et par conséquent à la productivité ou à la performance de ces équipes ; l'évaluation précise de la vélocité d'une équipe peut ressembler à une mesure de « flicage ».

Dans un cas comme dans l'autre, il est important de ne pas véhiculer un message qui se résumerait à ceci : « Il y a un problème et *vous* devez changer. » Tant les programmeurs que leur hiérarchie peuvent prendre l'initiative d'un passage vers XP, mais uniquement dans la mesure où chaque « spécialité » aura à cœur de fournir un meilleur travail pour l'autre, et que ce changement sera apprécié comme une véritable collaboration.

Par quoi commencer ?

Conformément à l'esprit général d'XP, Kent Beck formule dans *Extreme Programming Explained* une procédure simple pour adopter XP :

- identifier le problème le plus sérieux que l'on rencontre ;
- résoudre ce problème en appliquant les pratiques XP ;
- lorsque le problème est résolu, un autre devient à son tour « le problème le plus sérieux » : il faut donc reprendre au début.

Cette « recette » apporte une réponse possible à la question récurrente : « Doit-on pratiquer XP à 100 % ? » Nous l'avons souvent répété, les diverses composantes d'XP se complètent mutuellement et forment un tout cohérent ; elles sont efficaces lorsqu'elles sont pratiquées ensemble. Pour autant, chacune peut apporter des bénéfices concrets ; on peut tout à fait envisager d'adopter XP graduellement, selon la procédure qui vient d'être indiquée.

Points de départs courants

Cette démarche est évidemment plus difficile à appliquer qu'à formuler, mais elle a le mérite d'orienter la réflexion. Chaque contexte d'entreprise est différent ; les problèmes qui s'y posent le seront donc également, et il n'est pas possible de proposer une logique à l'adoption d'XP qui fonctionne pour tout le monde. Cependant, un certain nombre de situations

communes à de nombreux projets, ainsi que les liens logiques entre les diverses pratiques XP, permettent d'envisager des points de départ «canoniques» à la mise en place d'XP.

Une de ces situations classiques est celle de projets grevés par de très nombreux rapports d'incidents ou d'anomalies, parfois au point que la mise en place de nouvelles fonctionnalités est sans cesse retardée. La mise en place de tests unitaires ou fonctionnels apportera rapidement des bénéfices tangibles : après un certain temps, le code existant sera stabilisé et il sera possible d'aborder avec plus de confiance la planification des fonctionnalités demandées – on pourra alors se concentrer sur les problèmes sous-jacents qui sont à l'origine de ces débordements.

Commencer par les tests automatisés

La mise en place de tests automatisés est un point de départ logique et efficace pour un passage à XP pour plusieurs raisons : les bénéfices qu'apporte la pratique en termes de qualité des logiciels livrés peuvent être rapides et sensibles ; une fois en place, ils permettent de pratiquer le remaniement et de simplifier ou de consolider la conception ou l'architecture des produits ; enfin, lorsqu'elle est bien présentée, la notion de tests automatisés et en particulier de tests unitaires «façon XP» est rapidement perçue par les programmeurs comme un outil indispensable. (On parle même de programmeurs *test infected* – atteints par le virus des tests !)

Dans le cadre de nombreux projets, les problèmes les plus importants sont dus aux rapports entretenus avec les clients : ceux-ci se plaignent du manque de visibilité sur le projet, ou à l'inverse interviennent de façon excessive ; les logiciels livrés ne correspondent pas à l'expression initiale des besoins, ou au contraire cette dernière est trop incomplète ou imprécise pour que les programmeurs puissent s'assurer de la qualité de leur travail. Dans de telles situations, ce sont les techniques de planification et de répartition des responsabilités proposées par XP qui sont susceptibles d'apporter une amélioration : par exemple, la formalisation des besoins par le biais des tests fonctionnels, leur planification sous la forme de scénarios client dans le cadre d'itérations de durée fixe.

Commencer par la planification itérative

La mise en place du planning par itérations – y compris, sous une forme ou une autre, la notion des scénarios client – est un autre point de départ recommandé. Elle peut se faire indépendamment du rapport entretenu avec le «vrai» client, en désignant en dehors de l'équipe technique un «client interne», uniquement chargé des décisions de planification et de découpage des besoins, quelle que soit la forme sous laquelle ceux-ci auront été initialement exprimés. Une fois en place, ce mode de planification permet de jauger très efficacement l'effet des autres pratiques sur la vélocité de l'équipe – ou, le cas échéant, de déterminer ce qui la retarde, et donc ce qui constitue «le problème le plus sérieux».

Les principales synergies parmi les composantes d'XP se situent autour de ces deux pratiques de base. Un exemple : les tests unitaires mènent aux tests fonctionnels, qui rendent plus précise la formalisation des besoins et permet donc d'éviter tout flottement autour de la planification itérative, laquelle, à son tour, permet de normaliser les horaires de travail et donc d'augmenter la qualité de vie des programmeurs, et par conséquent la qualité de leur travail. Ou encore : les tests unitaires amènent à pratiquer le refactoring, puis à prendre conscience des bénéfices d'une conception simple ; la pratique fréquente du refactoring conduit à adopter des règles de codage uniformes, et par conséquent permet à l'équipe d'être collectivement responsable du code.

À l'inverse, certaines des pratiques d'XP ne découlent pas naturellement de cette démarche que l'on pourrait qualifier d'organique ou d'incrémentale. La présence du client au sein de l'équipe, la programmation en binôme ou l'utilisation d'une métaphore pour décrire l'architecture nécessiteront sans doute un effort conscient et délibéré pour être mises en place. Si ces aspects sont les seuls à faire défaut dans une méthode «quasi-XP», il sera important de ne pas passer à côté par manque de courage, parce qu'on se satisfait du chemin accompli. Ce serait certainement se priver de ce que peut apporter XP lorsqu'il est pratiqué au niveau 0.

Votre niveau XP

Ron Jeffries aime à rappeler – non sans humour – qu'il y a trois niveaux de conformité à XP. Au *niveau 0*, toutes les pratiques XP sont en place, maîtrisées et appliquées avec succès. Au *niveau 1*, on continue à faire tout ce qu'on faisait au niveau 0, avec un certain nombre d'adaptations qui permettent d'être encore plus efficaces. Au *niveau 2*, on est tellement efficace qu'on ne se soucie même plus de savoir si ce qu'on pratique est encore XP…

Choisir un projet pour XP

Aux deux axes que nous venons d'examiner – les personnes (qui peut décider d'un virage vers XP) et les pratiques (par quoi commencer) –, il faut adjoindre un troisième : parmi les différents projets qui peuvent être en cours ou en préparation dans une entreprise, lequel favorisera ou facilitera la mise en place de la démarche ?

Le choix de pratiquer XP sur un projet plutôt qu'un autre pourra avoir des répercussions importantes : bien sûr, le projet en question peut se solder par un échec ou par un succès mitigé, qui invaliderait temporairement ou définitivement la décision d'adopter XP. Inversement, c'est sur des résultats concrets qu'on jugera de l'intérêt de la méthode : elle ne pourra pas faire ses preuves si elle est cantonnée à des projets très réduits, à des expérimentations ou à des prototypes. Il s'agit donc d'arbitrer judicieusement entre prise de risques et potentiel de résultats.

Projets en cours ou projets de maintenance

Dans le cadre d'un projet en cours de réalisation, qui a atteint un rythme de croisière, la mise en place graduelle d'XP selon le schéma que nous avons exposé permettra d'obtenir rapidement des résultats et des retours d'expérience. Elle interviendra alors que les principales décisions techniques, fonctionnelles ou d'architecture auront déjà été prises et que des habitudes de travail auront été adoptées : l'introduction progressive de pratiques nouvelles dans un cadre familier pourra se faire sans que les différents intervenants ne perçoivent ces nouveautés comme menaçantes. Les prestations du type «tierce maintenance applicative» peuvent également se prêter à cette logique.

Cependant, les bénéfices que l'on pourra attendre de la méthode seront d'autant plus limités que le projet est avancé : si des défauts de conception ou des problèmes de qualité pèsent sur le code déjà produit, la pratique assidue des tests unitaires ou du refactoring ne les résoudra pas instantanément. Un travail de longue haleine sera nécessaire avant de rembourser la «dette technique» précédemment accumulée.

Un projet proche de sa date de livraison prévue, ou l'ayant dépassée, représente le cas limite de cette tactique. L'enjeu consistera alors à «sauver» le projet en mettant en place un nombre restreint de pratiques XP, de façon plus agressive : planification par itérations courtes, livraisons et intégrations fréquentes, refactoring et tests pour stabiliser des portions du système particulièrement sensibles.

Nouveaux développements

A contrario, une prestation de développement, en phase de démarrage ou de préparation, peut être l'occasion d'une mise en place plus complète d'XP, voire peut donner lieu à un premier projet «100 % XP». C'est dans ce contexte que les bénéfices promis en termes de conception simple et claire, de couverture complète par les tests unitaires, de collaboration étroite avec le client, auront le plus de chances de se concrétiser.

Il convient alors de garder à l'esprit que la mise en place d'XP représente en elle-même un changement important pour les équipes concernées, et par conséquent un facteur de risque. Il est essentiel de ne pas cumuler pour le même projet d'autres facteurs de risque significatifs : une équipe entièrement nouvelle ou réunissant des personnes qui n'ont jamais travaillé ensemble ; un budget nettement supérieur à ceux qui sont engagés habituellement ; des technologies nouvelles et mal connues ; un domaine fonctionnel peu connu, etc.

Accompagner la mise en place

Nous avons principalement discuté de la façon dont *peut être entamé* un virage vers XP. Il est tout aussi important de savoir l'accompagner sur la durée. C'est particulièrement vrai lorsqu'on vise une mise en place complète d'XP – on est alors largement en terrain inconnu, et il est important de prendre des repères – mais également lorsqu'on adopte «à la carte» telle ou telle pratique.

Une fois au moins au cours de chaque itération, il est souhaitable qu'une équipe XP ait l'occasion de faire le point sur la démarche elle-même, au-delà des aspects concernant le projet lui-même. Ces bilans se feront idéalement sous la responsabilité du coach. Ils peuvent être dressés de façon relativement formelle : par exemple, chaque membre de l'équipe indiquera, sur une liste des différentes pratiques XP, le «degré de maîtrise» qui lui semble refléter la situation actuelle.

Exemple d'auto-évaluation

Client Sur Site : 100 % [Michel est disponible pour répondre aux questions, les tests de recette sont pertinents].

Planification itérative : 50 % [Plans d'itérations OK, mais pas de revue du plan de livraisons depuis 2 itérations].

Livraisons fréquentes : 100 % [Plusieurs mises à jour du moteur de notre plate-forme Web par semaine].

Tests de recette : 90 % [Il est toujours aussi difficile de tester le code JavaScript dans nos pages].

Conception simple : 50 % [L'utilisation du «standard» EJB nous complique la vie].

Programmation en binôme : 50 % [On ne change pas assez fréquemment de binôme].

Développement piloté par les tests : 90 % [Quelques tests unitaires encore écrits après-coup].

Remaniement : 20 % [À améliorer d'urgence ! La qualité interne se dégrade].

Intégration continue : 100 % [Plusieurs intégrations par jour].

Responsabilité collective du code : 100 % [Tout le monde maîtrise la totalité du système].

Règles de codage : 80 % [Le code est presque homogène, mais pas encore de consensus sur les accolades].

Métaphore : 20 % [Un vocabulaire commun, mais encore hétéroclite : «page», «dossier», «stock»].

Rythme durable : 80 % [Pas d'heures sup', c'est bien, mais va-t-on pouvoir prendre des congés cette année ?].

On peut également représenter l'adhésion à la démarche telle qu'elle peut être ressentie par l'équipe au moyen de la figure proposée par Bill Wake, «l'Écran Radar» (figure 7-1). Cette approche, peut-être un rien cartésienne, sera utile aux équipes qui ont opté pour une adoption graduelle, qui pourront ainsi corréler productivité et niveau d'adhésion aux pratiques, ainsi qu'à celles qui expriment encore des doutes ou des inquiétudes vis-à-vis d'XP.

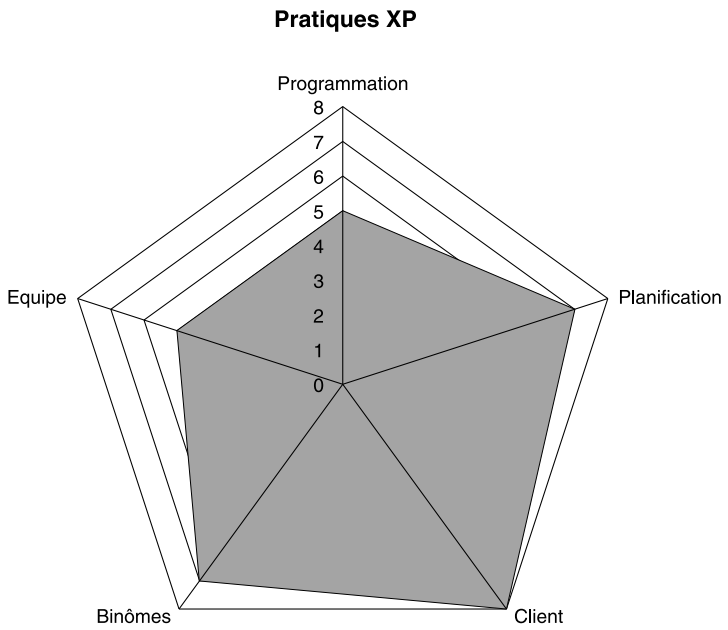

Figure 7-1. Adhésion à XP représentée sur l'Écran Radar

Des équipes plus enthousiastes ou plus confiantes pourront se contenter de discussions à bâtons rompus, et se dispenser de tout formalisme – mais pas, bien entendu, de la réflexion elle-même ! Voici les questions de fond qu'une équipe XP se posera régulièrement :

- Qu'est-ce qui a bien marché pendant cette itération ?
- Qu'est-ce qui aurait pu mieux marcher pendant cette itération ?
- Qu'est-ce que l'on pourrait tester, tenter ou expérimenter lors de la prochaine itération ?

Une équipe n'est réellement «extrême» que dans la mesure où elle a pleinement intégré cette notion fondamentale : *les pratiques d'XP sont un point de départ, non un aboutissement*. Une de ses principales caractéristiques sera sa capacité, sa volonté, et même son enthousiasme, à se remettre continuellement en question, à adopter de nouveaux points de vue, et plus généralement à se maintenir dans un état permanent d'apprentissage. Nous aborderons plus précisément ce sujet après avoir effectué un tour d'horizon des formations concernant directement la méthode elle-même.

Panorama des formations

Pour mettre en œuvre efficacement XP, il est indispensable d'être sensibilisé au fonctionnement des pratiques afférentes et à leur articulation. Et, bien entendu, la meilleure façon de comprendre comment les différentes parties d'XP s'organisent et se complètent, c'est de les avoir pratiquées.

Un paradoxe ? Plutôt une évidence : se lancer, sans formation préalable, dans la pratique d'une méthode qui requiert beaucoup de rigueur et d'attention, c'est prendre le risque de ne pas en retirer les bénéfices attendus. Cette formation peut être réduite à l'essentiel – la consultation d'ouvrages tels que celui-ci – mais, dans la plupart des cas, il sera souhaitable, d'une part, d'y consacrer un plus large investissement et, d'autre part, d'y faire participer le plus largement possible les personnes concernées.

Le coaching

La voie royale pour s'approprier XP consiste à s'adjoindre les services d'un consultant spécialisé. Ce consultant est appelé à jouer dans l'équipe qui aborde un virage vers XP le rôle du coach, décrit au chapitre 2. Sa mission et ses compétences seront cependant plus larges que celles requises d'un coach compétent « interne » : il (ou elle) devra se faire l'artisan d'un changement à la fois progressif et durable, et devra non seulement connaître les tenants et les aboutissants d'XP mais également disposer d'une expertise approfondie dans des domaines variés : techniques, organisationnels, psychologiques…

Évaluation préliminaire

Dans un premier temps – conformément au principe selon lequel on ne changera pas ce qui marche déjà –, il s'agira de dresser un état des lieux. Cela concerne l'entreprise autant que les intervenants techniques qu'elle emploie ; comme nous le verrons, XP n'est pas nécessairement compatible avec toutes les cultures d'entreprise. Les bénéfices que l'entreprise peut attendre de la démarche, sa capacité structurelle à procéder aux changements nécessaires pour la pratiquer, et la motivation ou les résistances éventuelles à ces changements, feront l'objet d'une évaluation objective.

La durée de cette évaluation peut varier de quelques jours à une ou deux semaines. Le rôle du consultant est principalement celui d'un observateur ; il assiste aux activités relatives aux projets informatiques de l'entreprise ou conduit des entretiens approfondis avec les différents intervenants des projets. En fonction des conclusions formulées par le consultant à l'issue de cette première phase, il peut suggérer des objectifs plus précis ou des recommandations spécifiques :

- ne pas modifier le processus en place ;
- adopter des pratiques spécifiques d'XP pour combler des lacunes ponctuelles ;

- mettre en place des formations ciblées ou générales, relatives à XP ou non ;
- mettre en place XP sur un projet identifié.

Actions de formation

Dans une phase plus active de cette transition accompagnée vers XP, le consultant est amené à intervenir de façon plus directe sur une période prolongée, typiquement de plusieurs semaines, en restant directement au contact des équipes concernées. Ces interventions peuvent être ponctuelles, à intervalles réguliers – par exemple, lors des réunions de bilan ou de planification des itérations –, pour évaluer les résultats fournis par une équipe plus autonome, et faciliter les «audits» ayant trait à la démarche : comment l'itération s'est-elle déroulée, quels incidents éventuels se sont produits et quelles solutions il convient d'y apporter.

On privilégiera un contact rapproché, dans lequel le consultant jouera à plein temps le rôle du coach : il sera disponible pour travailler en binôme avec l'un ou l'autre des programmeurs en cas de besoin, veillera au respect de la démarche, signalera les écarts éventuels et cherchera à faciliter le travail d'adaptation fourni par l'équipe. Tout particulièrement, il aura à cœur de s'assurer qu'à l'issue de sa mission un ou plusieurs des membres de l'équipe qu'il accompagne seront à même de le remplacer.

Ce rôle de coach peut se doubler d'un rôle de «mentor» auquel on demandera de développer des compétences spécifiques de l'équipe : formations à l'utilisation de langages ou d'outils, aux techniques de gestion de configuration, d'intégration ou de conception objet, etc. Ces formations privilégieront les aspects participatifs ; elles se feront plus souvent «les mains sur le clavier» que sous la forme d'un cours magistral.

Choisir un consultant

Il est important de rappeler qu'un bon consultant n'a pas vocation à «vendre» son outil ou sa méthode préférée : ses attributions consistent avant tout à fournir un réel service à l'entreprise, et aux personnes qui la composent. Si cette remarque est également valable pour n'importe quel consultant, elle s'applique doublement à un coach professionnel, dans la mesure où XP revendique comme seul objectif la réussite des projets. Inversement, les objectifs que vous fixerez à un consultant doivent correspondre à ses propres valeurs : votre coach XP ne sera pas un consultant spécialisé dans les certifications ISO ou CMM…

Les sessions d'immersion XP

Ces formations représentent le produit phare de ObjectMentor, la première entreprise de formation aux États-Unis à s'être spécialisée dans l'Extreme Programming. Elles s'adressent aux développeurs, mais également aux clients et aux décideurs, et ont pour objet de distiller, sur une durée de cinq jours, l'essentiel de ce que peut enseigner un coach au cours d'un projet réel.

Le format «Immersion» est, comme son nom l'indique, essentiellement axé sur la mise en pratique des enseignements d'XP, au contact d'experts. Kent Beck, Ron Jeffries et les autres créateurs d'XP y participent régulièrement en tant que formateurs ; les participants sont réunis par équipes de six personnes, chacune sous la responsabilité d'un des formateurs.

Un véritable projet de programmation en C++ ou en Java – à une échelle réduite, bien sûr – sert de fil conducteur à la formation. Au cours de la première journée, des exercices pratiques permettent de se familiariser avec les principaux éléments de l'environnement technique dont l'apprentissage est nécessaire à la pratique de la méthode : les outils de programmation, de gestion des versions, tels que CVS, ainsi que les procédures de création et d'exécution des tests unitaires et tests de recette.

Au cours des journées suivantes, les différents aspects de la méthode sont abordés par le biais des scénarios client constituant le projet, qui sont implémentés lors de courtes itérations, de l'ordre de quelques heures ; des cours plus théoriques, mais également des présentations et exposés délivrés par des invités et des discussions informelles complètent le contenu de ces sessions.

L'Heure Extrême, les ateliers thématiques

Dans ce format de présentation d'XP, il est proposé une approche théorique des pratiques qui constituent la méthode, suivie d'un exercice pratique. Cette session tient davantage de la sensibilisation que de la formation ; elle permet de communiquer brièvement le vocabulaire et la définition formelle d'XP, de préciser les bénéfices que cette méthode peut apporter et de recueillir les observations et les réserves des participants. Surtout, la phase pratique permet de montrer de façon concrète comment un projet XP se déroule et quelle en est la dynamique.

Au cours de la phase pratique (qui dure une heure), on applique la méthode XP au développement d'un système imaginaire (une machine à café, un véhicule écologique…). Les participants sont groupés par quatre, deux jouant le rôle de client/testeur, les deux autres celui de développeur. Le «produit final» donne lieu à un simple dessin au tableau ou sur une feuille de papier. La formulation des besoins, la planification du projet (on utilisera des itérations de dix minutes), l'organisation de la production, se font selon les pratiques XP applicables dans le contexte de cette simulation. Le formateur tourne parmi les groupes, jouant le rôle de coach.

De même que le format «Immersion», l'Heure Extrême permet d'aborder les principaux aspects d'XP d'une façon concrète, voire ludique. Si elle ne peut se substituer à une formation plus complète et rigoureuse, l'Heure Extrême présente un avantage notable, en ceci qu'elle peut être organisée de façon interne à l'entreprise, à l'initiative d'une ou de plusieurs personnes intéressées par XP. En outre, compte tenu qu'abstraction est faite des difficultés techniques, il est possible (voire conseillé) de faire jouer aux participants un rôle qui ne correspond pas à leur véritable rôle.

Enfin, et dans le même ordre d'idées, il est possible d'obtenir, ou d'organiser en interne, des formations plus restreintes autour de pratiques clés d'XP telles que le remaniement, la mise en œuvre de tests unitaires ou de recette, ou la programmation en binôme.

À qui s'adresser pour une formation ?

XP est encore, à ce jour, une discipline relativement nouvelle et (trop) peu pratiquée. Sa diffusion se fait pour l'essentiel selon un modèle artisanal – les « maîtres » forment quelques disciples qui, à leur tour, et après en avoir maîtrisé les bases, pourront en dispenser les enseignements. Ou encore – et c'est principalement le cas en France –, certains programmeurs ou chefs de projet découvrent XP sur Internet ou dans des forums de discussion, se forment avec les ouvrages existants et en font l'expérience en appliquant la méthode sur leurs projets. Il est donc encore trop tôt pour pouvoir se fier à une éventuelle certification ou labélisation XP qui garantirait la pertinence et la qualité des formations. D'ailleurs, la notion même d'une « certification » est assez peu compatible avec la philosophie d'XP…

Pour recevoir l'assistance d'un coach professionnel ou suivre une formation complète, l'idéal est de s'adresser à ObjectMentor, qui fédère les activités de formation de ces « gourous ». Ce n'est évidemment pas la solution la plus économique pour les entreprises basées en Europe… Par chance, une deuxième génération apparaît actuellement, composée de pionniers qui ont d'ores et déjà mis en œuvre XP, et, pour certains, suivi ces formations officielles. Des sessions XP Immersion, notamment, sont désormais organisées plus ou moins régulièrement en Europe avec la participation officielle d'ObjectMentor.

Enfin, comme nous l'avons indiqué précédemment, la formation en interne est une solution peu coûteuse et tout à fait envisageable.

L'esprit XP et la culture d'entreprise

Au-delà de recommandations concrètes quant à la façon d'aborder le travail de programmation, la méthode XP s'appuie sur des valeurs et une philosophie. On peut en dire autant, au demeurant, de toute méthode de travail ou d'organisation ; ce qui distingue XP à cet égard, c'est que ces valeurs sont formulées explicitement. Par ailleurs, XP est une approche « politiquement incorrecte », qui va à l'encontre de nombreuses idées et traditions solidement ancrées dans la plupart des entreprises de l'industrie du logiciel.

À ce titre, sa mise en place s'accompagne inévitablement d'une réflexion sur la culture de l'entreprise ; elle ne peut pas se résumer à une décision purement formelle sur les règles d'organisation du travail, sans que ne soit prise en compte sa compatibilité avec les autres modes de fonctionnement, tels que le recrutement, la formation, la pratique contractuelle, l'affectation des personnes aux missions ou aux projets, la communication interne, etc.

Quelques exemples concrets

La pratique de la programmation en binôme est en soi un outil de formation : l'un de ses fondements est que, lorsque le travail de programmation se fait à deux, des transferts de compétences ou de connaissances s'opèrent naturellement entre les deux personnes du binôme. Elle implique également un mode de recrutement adapté, dans la mesure où il faudra donner plus d'importance à la compatibilité d'humeur et de personnalités entre les développeurs amenés à travailler ensemble de cette façon. Comme nous l'avons vu, elle peut conduire à réorganiser, physiquement, les bureaux d'une équipe, ce qui implique de la part de l'entreprise une flexibilité que toutes n'auront pas nécessairement…

Établir un lien entre la culture de l'entreprise et la méthode qu'elle pratique pour ses développements logiciels n'est pas une question abstraite ou de principe. C'est une nécessité qui découle de ce qu'on a appelé la loi de Conway – soit l'observation, qui remonte à 1968, selon laquelle la structure d'un programme informatique reflétera nécessairement celle de l'entreprise ou de l'organisation qui l'a créé. Le corollaire suivant est donc pertinent : si les systèmes produits par une équipe ou une entreprise donnée souffrent de défauts de conception, ils ne seront véritablement résolus qu'en diminuant les problèmes d'organisation correspondants.

Sans prétendre être exhaustif, voici quelques pistes de réflexion sur ce que peut être ce travail qui doit accompagner une mise en place réussie d'XP.

Culture technique

Trois des valeurs affichées par XP ont principalement trait à la dimension humaine des problématiques du développement logiciel (communication, *feedback*, courage), une seule étant plus directement liée à la dimension technique (simplicité), encore que cela peut aussi être une qualité humaine. C'est le reflet d'un des principes qui font certainement partie de la «culture XP» : les projets n'échouent généralement pas pour des raisons techniques, mais plutôt relatives aux personnes, à leur organisation, leur communication ou leur motivation.

Pour autant, cette réflexion sur la culture d'entreprise ne doit pas négliger les aspects techniques. L'un des exemples les plus importants en est l'utilisation des techniques de programmation objet. XP n'impose pas *a priori* l'utilisation des langages objet et peut s'appliquer aussi sur des projets réalisés dans des langages tels que C, voire Lisp, ASP ou Visual Basic ; mais c'est probablement dans le domaine de l'objet que les bénéfices qu'on peut en attendre sont les plus grands.

Quels que soient les choix de l'entreprise dans ce domaine, ils devront être cohérents, et mobiliser le plus largement possible les développeurs pour contribuer à consolider cette culture technique. Si c'est le monde de l'objet qui apparaît comme un choix stratégique, il faudra éviter de limiter sa vision à un seul langage, en s'intéressant à des technologies connexes telles que les bases de données objet, en proposant des formations ciblées dans ce domaine,

ou encore en étudiant des ouvrages de référence, par exemple *Object Oriented Software Construction* de Bertrand Meyer[1].

Culture de la qualité

Tout est fait au sein d'XP pour donner une place prépondérante à la qualité des développements. La notion de qualité qui y est intégrée recouvre aussi bien la qualité interne (c'est-à-dire telle qu'elle est perçue par les programmeurs) que la qualité externe (celle à laquelle les clients attachent le plus d'importance). Cette place de premier plan est garantie par la charte des droits du programmeur ainsi que par celle du client (voir annexe IV) ; elle trouve son application pratique dans la notion de client sur site, qui fait des deux camps une seule et même équipe attachée au succès du projet.

Cette importance accordée à la qualité doit bien entendu trouver un écho au-delà de la méthode elle-même, dans les relations que l'entreprise entretient avec ses clients et ses utilisateurs avant, pendant et après la réalisation d'un projet. L'esprit qui anime XP est sous certains rapports assez proche de celui de William Deming (Qualité Totale) ou Philip Crosby (Zéro Défaut), qui, dans des domaines entièrement différents, ont montré comment la gestion de la qualité pouvait être un atout compétitif pour l'entreprise.

Les principes énoncés par XP en matière de qualité sont relativement génériques et cantonnés à une notion restreinte, rationnelle, de « validité » des systèmes développés. Ils ne tiennent pas compte, par exemple, de jugements plus subjectifs, voire émotionnels, comme on peut en rencontrer par exemple dans le domaine de l'ergonomie – la facilité de prise en mains de l'outil, ou l'efficacité avec laquelle il soutient la tâche de l'utilisateur. Cela peut donner lieu à l'intégration de pratiques supplémentaires, telle la démarche de conception des interfaces centrées sur l'utilisateur (UCD) proposée par Larry Constantine.

Plus généralement, selon les domaines où intervient l'entreprise et selon les types de produit qu'elle développe, la culture spécifique de la qualité qui y est pratiquée pourra compléter la vision qu'en donne XP.

Culture d'encadrement

De tous les aspects que recouvre le terme de « culture d'entreprise », le plus important est certainement celui de l'encadrement ; notre discussion de la répartition des rôles et des responsabilités au sein d'XP a déjà suggéré un décalage par rapport à une vision plus classique de l'organisation du travail.

La conception de l'encadrement implicite dans XP se rapproche notamment de celle détaillée par Gerald Weinberg dans la série *Quality Software Management*, et tient compte des spécificités psychologiques du travail de développement logiciel. Elle voit l'équipe de développement comme un système dynamique complexe plus que comme une structure strictement

1. *Conception et programmation orientées objet,* Eyrolles, 2000, pour la traduction française.

hiérarchisée et parfaitement prévisible. Elle préfère le terme de «leadership» à celui de «management», et suggère que le principal enjeu de l'encadrement des équipes techniques est de favoriser le développement des compétences individuelles et les synergies au sein des équipes.

Pour le responsable d'une équipe, et même pour les dirigeants supérieurs, la notion de «ressources humaines» ne pourra avoir qu'une connotation sinistre. Il (ou elle) se rappellera que les programmeurs, comme tous les autres intervenants des projets, ne sont pas interchangeables entre eux – et plus encore, comme le rappelle Fred Brooks (*Le Mythe du Mois-Homme*), que les hommes ne sont pas interchangeables avec les mois ; que l'adage «diviser pour régner» s'est toujours révélé redoutable contre l'ennemi et qu'il faudrait être stupide pour l'appliquer à ses propres équipes ; que, malgré tout l'effort investi pour la «capitalisation» ou la «réutilisabilité» des développements, une équipe qui fonctionne bien reste le meilleur capital de l'entreprise, et le plus facilement réutilisable.

Encore une fois, pour une équipe «extrême», mais encore plus pour une entreprise «extrême», XP n'est pas un aboutissement mais un point de départ. C'est au sein d'une culture d'entreprise qui n'aura de cesse de remettre en question les idées reçues, de cultiver la diversité parmi ses collaborateurs, de respecter l'individu et de poursuivre en permanence son apprentissage qu'XP trouvera pleinement sa place.

8

Coûts et retours sur investissement

Des investissements, c'est de l'argent. De ce côté-là, ça ressemble pas mal à des dépenses.

– Ronald Lavallée

Le succès d'XP passe par une perception économique favorable

Si l'Extreme Programming connaît aujourd'hui un succès médiatique incontestable, force est de constater que son utilisation au sein des entreprises françaises reste encore très expérimentale. L'un des principaux freins à l'adoption de la méthode tient probablement au caractère incertain du retour sur investissement. Dans un contexte économique plus tendu, la tendance naturelle des entreprises est de se réfugier auprès de méthodes, qui, à défaut d'avoir prouvé leur retour sur investissement, bénéficient d'une reconnaissance générale.

L'adoption d'XP peut de ce fait être vue comme une prise de risque inconsidérée. La mise en avant de certaines pratiques, comme la programmation en binôme, ne facilite pas l'adhésion des chefs de projets et des directeurs de systèmes d'information : «Comment pourrai-je justifier devant la direction financière le fait de mettre deux développeurs par poste ?»… L'adoption d'XP passe alors nécessairement par une identification claire des économies réalisées : un exercice d'autant plus complexe que le calcul du retour sur investissement sur ce type de sujet relève souvent d'un numéro d'équilibriste.

Dans ce chapitre, nous allons donc examiner la façon dont la mise en œuvre d'XP peut faciliter la maîtrise des coûts, et ce avec une approche aussi objective et pragmatique que possible.

En fonction des spécificités de chaque projet, et en s'appuyant sur les enseignements théoriques et pratiques des chapitres précédents, il sera ainsi possible d'identifier, d'une part, les pratiques d'XP les plus faciles à justifier en termes de retour sur investissement et, d'autre part, celles qui soulèveront des objections de nature financière.

Quatre variables clés

Pour étudier le sujet objectivement, il est essentiel de l'appréhender dans sa globalité. Or, comme nous l'avons vu au chapitre 6 consacré à la gestion de projet, le coût d'un projet ne peut être observé indépendamment de plusieurs autres facteurs : la qualité livrée, le temps imparti pour réaliser le projet et la largeur du périmètre fonctionnel que l'on va couvrir (figure 8-1). Ces quatre variables sont en effet profondément interdépendantes.

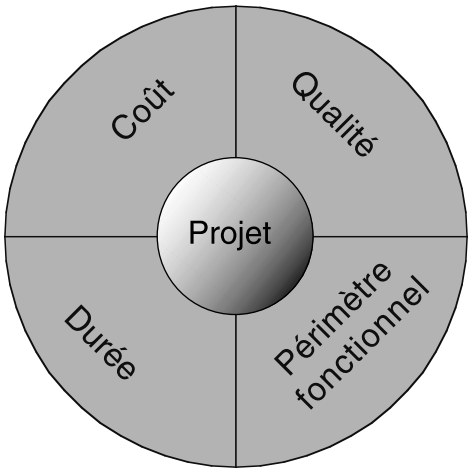

Figure 8-1. Le projet est soumis à quatre paramètres qui s'influencent mutuellement

1- Les coûts directs et indirects

L'identification des coûts associés à un projet est un exercice complexe, souvent trompeur par sa simplicité apparente. En effet, au-delà des coûts directs rapidement identifiés (nous y reviendrons dans ce chapitre), la prise en compte des coûts indirects est bien moins évidente, alors qu'elle en constitue souvent la composante déterminante. Outre les coûts directs, nous avons choisi de mettre en évidence cinq coûts indirects classiques : les coûts de débordement, les coûts liés à l'inadéquation de la réalisation aux besoins réels, les coûts du changement fonctionnel, les coûts associés aux défauts de qualité et les coûts liés au turnover des équipes. Chacun de ces coûts, nous le verrons, est affecté par l'Extreme Programming. Ils n'ont en

aucun cas un caractère exhaustif, bien au contraire. Chaque projet doit faire l'objet d'une étude de coûts précise et adaptée au contexte, qui sort du cadre de cet ouvrage.

2- La qualité livrée (interne, externe)

La qualité est une variable complexe, souvent utilisée à tort et à travers. Une définition simple et essentielle permet de bien aborder les problématiques : la qualité est la conformité aux exigences.

Remarque

La définition de la qualité comme conformité aux exigences est dûe à Philip Crosby qui, dans son livre *Quality is Free*, en déduit une «grille de maturité» qui inspirera directement le modèle dit *Capability Maturity Model* ou CMM, qui sera abordé au chapitre 10. Il est intéressant de la comparer à celle qu'en donne Jerry Weinberg : «Est de qualité ce qui a de la valeur pour une personne.» Renvoyer dos à dos ces définitions en qualifiant l'une d'objective et l'autre de subjective ne fait qu'effleurer la surface du débat philosophique que ce simple terme engendre !

Si les exigences sont faibles, la qualité peut être très bonne sans effort de la part de l'équipe. Il est donc essentiel de se référer à des indicateurs de mesure qui permettront de définir le niveau de conformité aux exigences. Ces indicateurs pourront être définis dans le plan d'assurance qualité. La qualité, nous allons le voir, est très dépendante des autres variables. Les indicateurs pourront être «externes» (niveau de qualité mesuré par le client) ou «internes» (niveau de qualité mesuré en interne, par le responsable qualité ou par les équipes). Les indicateurs internes et externes sont généralement de nature très différente.

3- La durée des projets

La durée des projets ne correspond pas au temps passé cumulé par l'équipe mais bien à la durée temporelle du projet (date de début/ date de fin). C'est une variable qui est généralement fixée à l'avance et qui influe elle aussi fortement sur les autres variables.

4- Le périmètre fonctionnel des projets

Le périmètre fonctionnel correspond au champ des fonctionnalités couvert par le projet. Si l'on considère une méthode classique, le périmètre fonctionnel sera décrit dans les spécifications fonctionnelles. Dans le cadre d'un projet XP, le périmètre fonctionnel est défini par les scénarios client.

Dépendance entre les variables

Ces quatre variables sont étroitement dépendantes les unes des autres. Pour mieux le percevoir, il suffit de mesurer l'impact d'une fixation arbitraire de l'une d'entre elles.

1. Fixer un coût à l'avance (donc définir un budget) est une pratique fréquente – voire «obligatoire» selon le modèle contractuel, mais nous examinerons plus en détail cette question au chapitre suivant. C'est aussi un choix lourd de conséquences, en particulier si le budget a été sous-dimensionné ou surdimensionné. Un budget sous-dimensionné aura souvent un impact sur le niveau de qualité délivré, ce qui influera généralement sur les coûts indirects (coûts de maintenance, coûts d'indisponibilité des applications livrées, etc.). L'expérience montre qu'un projet dont le budget initial est trop faible finit par coûter plus cher que si le budget initial avait été plus généreux. À l'autre extrême, un budget surdimensionné peut également être lourd de conséquences : il poussera souvent à déployer dès le démarrage du projet des équipes surdimensionnées, qui allongeront la durée des projets et pourront même nuire à la qualité de service livrée (fractionnement trop important du travail, difficultés à coordonner l'ensemble des intervenants, etc.).

2. Il est fréquent de penser que baisser le niveau de qualité permettra de réduire les coûts. Or, l'expérience montre que l'impact est souvent inverse : baisser la qualité fait croître les coûts, et ce, que l'on agisse sur les indicateurs externes ou internes. Lorsqu'il s'agit des indicateurs externes, le client – sauf cas exceptionnel – finit toujours par découvrir la baisse des exigences en dessous d'un seuil qu'il jugeait nécessaire, ce qui peut engendrer des coûts à la fois directs et indirects (nécessité de recommencer une partie des développements dans le budget initial, coûts de maintenance en forte hausse, etc.). La baisse du niveau de qualité interne se répercute également sur la motivation de l'équipe, augmentant le taux de turn-over (et donc les coûts de remplacement au sein des équipes par exemple).

3. Fixer la durée du projet agit fortement sur le périmètre fonctionnel mais aussi sur le niveau de qualité (donc, indirectement, sur les coûts) : une échéance trop courte sera pénalisante à la fois pour les fonctionnalités du projet (nécessité de faire des coupes dans les spécifications), mais aussi pour la qualité : des équipes sous pression permanente finissent par manquer de recul et par abaisser leur niveau d'exigence.

4. Modifier le périmètre fonctionnel permet aussi de piloter les autres variables : réduire les fonctionnalités couvertes a nécessairement un impact favorable sur les coûts, la durée, et donc indirectement sur la qualité. L'évidence du propos peut faire sourire. Et, pourtant, la conduite du projet par la surveillance active du périmètre fonctionnel est l'un des axes forts de l'Extreme Programming, et probablement l'un des points clés permettant d'identifier le plus simplement l'impact favorable d'XP sur les coûts…

Maîtrise des variables par adaptation du périmètre fonctionnel

La plupart des retours médiatiques sur l'Extreme Programming tendent à faire oublier que les fondateurs d'XP ont défini cette nouvelle approche avant tout pour maîtriser les risques et les coûts. L'Extreme Programming s'appuie sur les dépendances entre les quatre variables étudiées ci-dessus pour formuler le constat suivant : si l'on souhaite maîtriser à la fois les risques et les coûts des projets, la variable la plus efficace sur laquelle il faut agir est le périmètre fonctionnel.

> Modifier le périmètre fonctionnel ne signifie pas que l'on va priver le client de fonctionnalités essentielles. Cela signifie simplement que le client et l'équipe projet vont se concentrer d'abord sur les fonctionnalités essentielles, celles qui permettent de créer le plus de valeur pour le client, et vont ensemble ajuster le périmètre des fonctionnalités pour respecter les contraintes fixées à l'avance sur les coûts, les délais et le niveau de qualité souhaité.

La réduction du périmètre fonctionnel aux éléments identifiés comme primordiaux permet de limiter la taille des équipes affectées au projet (donc, encore une fois, de faciliter la maîtrise de la qualité et des coûts). Cette limitation permet également d'adopter une démarche itérative : à l'issue de la première livraison, des fonctionnalités identifiées comme moins essentielles seront peut-être devenues inutiles, ou auront fortement changé, ce qui permettra de démarrer un nouveau cycle en se concentrant à nouveau sur les fonctionnalités clés.

La proposition de valeur de l'Extreme Programming consiste donc à maîtriser et même à réduire les coûts d'une gestion de projets traditionnelle, tout en gardant la maîtrise des délais et de la qualité. De ce pilotage des paramètres par le périmètre fonctionnel découlent pratiquement toutes les pratiques de l'Extreme Programming, et notamment les pratiques relatives à la planification.

Pour pousser un peu plus loin cette présentation générale de la maîtrise des coûts par une démarche XP, il est important de rentrer plus en profondeur dans le sujet en analysant dans le détail les coûts d'un projet informatique et en regardant les impacts d'XP sur ces coûts. Mais gardons une nouvelle fois à l'esprit que l'Extreme Programming promet à la fois la maîtrise des coûts mais aussi celle de la qualité et des délais, ce qui peut être également déterminant pour une direction financière ou générale.

Les coûts d'un projet informatique

Le recensement des coûts associés à un projet informatique relève, nous l'avons dit, d'un exercice d'équilibriste. L'analyse présentée ici ne prétend donc pas être exhaustive. Nous nous sommes efforcés de nous concentrer sur les coûts les plus représentatifs et les plus classiques d'un projet type. Chacun adaptera naturellement les éléments décrits à son contexte.

Les coûts de réalisation directs «traditionnellement» identifiés

Les coûts directs naturellement identifiés sur un projet informatique (C1 à C9) se répartissent comme suit.

Main-d'œuvre pour la conception et la réalisation (C1 et C2)

Ce sont les coûts des ressources affectées à la conception et à la réalisation du projet, qu'elles se situent côté maîtrise d'ouvrage ou maîtrise d'œuvre. On y retrouve logiquement :

- le coût du temps passé par le client pour spécifier ses besoins, gérer sa relation avec l'équipe projet, tester et «recetter» les applications livrées, etc. (C1) ;
- le coût du temps passé par l'équipe projet, que ce soit sur la rédaction de spécifications et de documentation, sur la conception, le développement, les tests, la recette (C2).

Logiciels (C3 et C4)

Le coût logiciel se subdivise lui aussi en plusieurs coûts :

- le coût des licences logicielles des outils de conception, de développement, de tests, de documentation automatique, bref le coût des licences associées au génie logiciel (C3) ;
- le coût des licences relatives à la production des applications développées (serveurs d'applications, middlewares, composants payants, outils de surveillance, etc.) (C4).

Déploiement (C5)

Les coûts de déploiement concernent à la fois la maîtrise d'ouvrage, la maîtrise d'œuvre, les équipes système et réseaux et les équipes d'exploitation. Ce sont essentiellement des coûts liés au temps passé (C5).

Infrastructure (C6)

Les coûts d'infrastructure concernent tout ce qui est lié au matériel engagé permettant l'exploitation des applications : serveurs, sécurité, salles blanches, etc. (C6).

Formation (C7)

Les coûts de formation ont trait au temps passé à la réalisation des supports et à la formation elle-même, qu'ils engagent le formateur ou les stagiaires (C7). Nous avons souhaité volontairement les isoler du coût de main-d'œuvre Conception/Réalisation du projet.

Maintenance (C8 et C9)

Les coûts de maintenance se subdivisent en deux : coûts de maintenance corrective (C8) et coûts de maintenance évolutive (C9). La maintenance corrective prend en charge la gestion et la correction des incidents, la maintenance évolutive peut correspondre à des interventions

ponctuelles ou à de vrais chantiers comparables en fait à des projets informatiques (d'où une nouvelle subdivision des coûts).

Les coûts indirects

La pondération des différents coûts indirects présentée ci-après dépend bien sûr de la nature des projets. Néanmoins, force est de constater que les coûts indirects (C10 à C14) sont souvent négligés alors qu'ils peuvent représenter en fait l'essentiel du budget.

Les coûts de débordement (C10)

Le prix d'un débordement (C10) dû au non-respect des délais est souvent complexe à chiffrer. Les coûts cachés sont en effet importants. Outre une augmentation des coûts de main-d'œuvre C1 et C2, les impacts financiers pour la maîtrise d'ouvrage peuvent constituer l'essentiel des coûts (non-disponibilité d'une application entraînant du chômage technique, etc.).

Les coûts d'une inadéquation aux besoins réels (C11)

Il s'agit là aussi d'un coût indirect (C11) difficile à évaluer. Il correspond à une inadéquation entre l'application livrée et les besoins réels (souvent différents des besoins exprimés) du client. Cela peut engendrer une perte de temps des utilisateurs, un manque à gagner dû à l'absence d'une fonctionnalité, etc.

Les coûts du changement (C12)

Les coûts du changement (C12) sont définis comme les coûts intrinsèques associés à une modification substantielle des fonctionnalités, indépendamment de l'amplitude ou de la nature de ces modifications. Ainsi une demande d'évolution portant sur l'adaptation au Web aura-t-elle un impact différent selon que le système a été à l'origine conçu pour le Web, et écrit en Java par exemple, ou conçu comme un produit Windows, et écrit en C, même si la fonctionnalité décrite est rigoureusement la même. L'une des hypothèses des méthodes traditionnelles est que le coût du changement évolue très fortement en fonction du moment où il est pris en compte. La courbe traditionnelle illustrant ce propos est présentée à la figure 8-2.

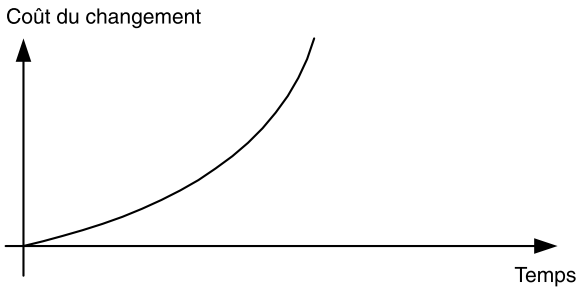

Figure 8-2. Évolution du coût du changement en fonction de l'avancement du projet

Les coûts des défauts de qualité (C13)

Les coûts associés aux défauts de qualité (C13) pourraient être tout simplement rangés dans les coûts de maintenance corrective. Néanmoins, nous avons préféré les isoler pour bien distinguer les incidents mineurs, incontournables dans tout développement informatique, des coûts associés à une problématique de qualité : défauts incessants liés à des exigences de qualité fixées trop bas, etc. Ces coûts peuvent parfois prendre des proportions très importantes, voire dépasser le budget initial du projet.

Les coûts du turn-over (C14)

Les coûts de turn-over (C14) sont ceux qui doivent être imputés à l'instabilité des équipes : démissions, départs en tous genres devant entraîner un transfert de connaissance, une formation au contexte projet, etc.

Exemple d'étude de coûts

Afin de mieux percevoir la part respective des coûts, nous allons présenter une étude de cas – volontiers simple, mais qui permet néanmoins de concrétiser les éléments abordés précédemment. Cette étude de cas met en évidence les budgets attribués à des défauts de qualité, à des débordements qui restent naturellement très subjectifs. Cependant, nous nous sommes efforcés de nous appuyer sur des quotes-parts typiques en la matière.

On s'attachera à un projet d'application trois-tiers destinée à l'entreprise et gérée au quotidien par trois utilisateurs. La charge de travail pour la conception et le développement est évaluée par le fournisseur à 1 chef de projet et 4 ingénieurs de développement pendant 4 mois. Le chef de projet est facturé 900 euros par jour, les ingénieurs de développement 600 euros par jour.

On considère que le client, lors de la réalisation du projet, attribue l'équivalent d'une personne à plein temps sur le projet (réparti en fait entre plusieurs utilisateurs). Le coût journalier de l'interlocuteur client est pour l'entreprise de 750 euros par jour.

La quote-part en termes d'outils de développement est évaluée à 30 000 euros, l'investissement matériel et serveurs d'application à 150 000 euros.

Le déploiement de l'application mobilise un interlocuteur client à plein temps pendant une semaine, l'équipe de développement et un administrateur réseaux (900 euros).

La formation des utilisateurs et administrateurs (et sa préparation) mobilise le chef de projet pendant 10 jours. Côté utilisateurs, on prend en compte 1 jour de formation pour trois personnes et deux jours pour deux exploitants.

Les coûts de maintenance (corrective et évolutive) sont de l'ordre de 15 % du budget de réalisation et de mise en place, soit quelque 60 000 euros.

La réalisation du projet connaît un dépassement de l'ordre de 25 %. Le retard pris par le lancement du projet est supposé ne pas avoir d'impact sur la bonne marche de l'entreprise.

Le projet ne correspond pas en fait aux attentes exactes des gestionnaires du système, qui perdent un temps précieux dans l'utilisation du logiciel (3 % du temps, soit pour 215 jours travaillés 6,5 jours par an).

L'application n'a pas été conçue dans une logique de changement et d'évolutivité. La mise en œuvre de la V2 (qui dépasse le cadre de la maintenance évolutive) oblige à redévelopper 30 % de l'application existante, soit un coût de l'ordre de 90 000 euros.

Tableau 8-1. Répartition de coûts

	Typologie de coût	Importance relative du coût dans le projet (usuelle) (en euros)
	Coûts Conception/ Réalisation	
C1	Coût main-d'œuvre client	80 × 750 = 60 000 euros
C2	Coût main-d'œuvre projet	80 × 900 + 80 × 600 × 4 = 120 000 euros
C3,C4, C6	**Coûts Logiciels/ Infrastructure**	180 000 euros
	Coûts de déploiement	
C5	Coût de main-d'œuvre	5 × (750 + 900 + 2400 + 900) = 24 750 euros
C7	**Coûts de formation**	10 × 900 + 3 × 750 + 2 × 2 × 900 = 14 850 euros
	Coûts de maintenance	
C8,C9	Coût de main-d'œuvre (maintenance évolutive et corrective)	60 000 euros
	Coûts de débordement	
C10	Coût de main-d'œuvre	75 000 euros
C10	Coûts «business» dus au débordement	-
	Coûts d'inadéquation aux besoins réels	
C11	Coûts «business» dus à l'inadéquation aux besoins	3 × 6,5 × 750 = 14 625 euros
	Coûts de changement	
C12	Coût de main-d'œuvre	90 000 euros
	Coûts de défauts de qualité	
C13	Coût de main-d'œuvre (maintenance)	(déjà évalué dans la maintenance)

Cette étude de coût, même sommaire, permet d'évaluer la répartition des coûts. On observe, en particulier, que les coûts associés au débordement, à l'inadéquation aux besoins réels et au changement représentent généralement une part importante du budget total.

La mise en œuvre de ce type d'étude sur vos propres projets peut s'articuler en deux phases. Dans un premier temps, on étudiera la décomposition des coûts réels constatés sur plusieurs projets représentatifs. (Il convient de ne pas commettre des «erreurs de sélection» à cette étape, au risque de fausser les résultats : ne prendre en compte par exemple que les projets qui ont bien réussi, ou que ceux qui ont dérapé.) Ces études pourront, dans un second temps, guider des décompositions prévisionnelles sur des projets envisagés et jouer un rôle important dans l'étude de faisabilité.

Nous allons maintenant étudier quels sont les centres de coût particulièrement bien adressés par XP, et ceux qui peuvent au contraire faire pencher la balance du côté d'une gestion de projet traditionnelle.

Un projet XP coûte-t-il plus cher qu'un projet traditionnel ?

Déterminer si un projet XP revient plus ou moins cher qu'un projet traditionnel, chiffres à l'appui, relève naturellement de la pure utopie. En supposant que le même projet soit réalisé par deux équipes en parallèle, l'une s'appuyant sur XP, l'autre sur une méthode traditionnelle, cela ne permettrait naturellement pas de départager les méthodes ; l'importance du facteur humain est telle qu'à lui seul il suffit à faire pencher la balance d'un côté ou de l'autre.

Néanmoins, en s'appuyant sur les centres de coûts identifiés dans la section précédente, il est possible d'identifier les points où l'Extreme Programming peut apporter des solutions ou au contraire présenter des risques par rapport à une méthode classique.

Le coût sur lequel se concentrent pour l'essentiel les débats est le coût de la main-d'œuvre humaine (C1 + C2 + C5 + C7 + C8 + C9 + C10 + C12 + C13 + C14). Les opposants à XP mettent en exergue plusieurs points qui contribuent selon eux à l'alourdissement des charges : la programmation en binôme risque de tout simplement multiplier C2 par deux ; la présence du client sur site augmente également C1 ; l'absence de documentation suffisamment détaillée peut également influer sur C5, C8 et C9.

En réponse à cet argumentaire, les partisans de l'Extreme Programming mettent en avant plusieurs points :

- La mise en place de la programmation en binôme, tout comme celle des tests automatisés, permet non seulement de réduire les défauts de qualité (C11, C13 et C8) mais également de minimiser les temps passés en conception et développement (réalisation d'un code propre dès le départ, et donc peu de bogues à rechercher, etc.).
- La présence du client sur site ne nécessite pas forcément sa collaboration effective toute la journée, mais simplement sa disponibilité, notamment pour lever des ambiguïtés bloquantes pour l'équipe, permettant ainsi des économies sur C1. En outre, un client XP effectue des tâches qui incomberaient sinon à l'équipe de réalisation : rédaction de spécifications, définition de tests de recette, choix des priorités lors de la planification.

- Les pratiques mises en œuvre par XP permettent de monter le niveau de satisfaction de l'équipe projet, et ainsi de réduire le taux de turnover (C14). Ce dernier est d'ailleurs moins pénalisant pour une équipe XP, en raison de la responsabilité collective du code, de la diffusion des compétences et de l'intégration rapide des nouveaux venus.

XP offre également sur les coûts relatifs au changement et à la maintenance évolutive (C9 et C12) une vision radicalement différente de celle qui est habituellement proposée. En effet, XP remet en cause la courbe du coût de changement présentée plus haut, en y opposant celle de la figure 8-3.

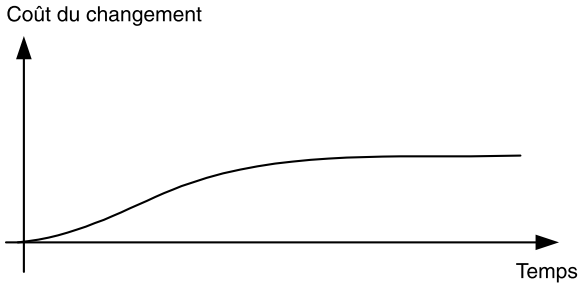

Figure 8.3. Évolution du coût du changement en fonction du temps dans une démarche XP

Quels sont les fondements de cette vision ? Comme nous l'avons vu dans la première partie de cet ouvrage, les pratiques de programmation (conception simple, remaniement, tests unitaires) et les pratiques de collaboration (travail en binôme, responsabilité collective du code) permettent de garantir la qualité interne de l'application, et ainsi de faciliter les modifications et les évolutions tout au long du projet. De ce fait, l'approche du coût de changement (C12) devient radicalement différente, et les gains engendrés par une réduction de C12 peuvent à eux seuls faire incliner pour XP.

Il nous est apparu intéressant de rassembler l'ensemble des coûts dans un tableau de synthèse (tableau 8-2) en notant ceux qui sont particulièrement susceptibles d'être réduits par XP. Ce tableau doit évidemment être adapté en fonction du contexte dans lequel évolue l'entreprise.

Tableau 8-2. Retours sur investissement d'XP

	Typologie de coût	Importance relative du coût dans le projet (usuelle)	Impact potentiel d'XP sur le coût usuel
	Coûts conception/ réalisation		
C1	Coût main-d'œuvre client	Moyenne	Augmentation : présence à plein temps du client.

	Typologie de coût	Importance relative du coût dans le projet (usuelle)	Impact potentiel d'XP sur le coût usuel
C2	Coût main-d'œuvre projet	Forte	Diminution : petites équipes privilégiées, pas de surdimensionnement. La proximité du client réduit les temps d'attente et l'écart avec les besoins réels. Diminution *ou* augmentation : par la programmation en binôme. C'est l'une des controverses autour d'XP. Cf. chapitre 6.
	Coûts logiciels		
C3	Outils conception/développement/ tests…	Moyenne	Diminution : l'obsession de la simplicité se répercute aussi dans le choix des outils de génie logiciel.
C4	Plates-formes serveur	Forte	Aucun.
	Coûts de déploiement		
C5	Coût de main-d'œuvre	Moyenne	Diminution : le processus d'intégration continue permet d'anticiper des problèmes de déploiement. Augmentation : l'approche minimaliste de la documentation peut conduire à des surcoûts si elle est prétexte à négliger ces aspects.
C6	**Coûts d'infrastructure**	Forte	Aucun
C7	**Coûts de formation**	Moyenne	Diminution : si le « client XP » est lui-même l'utilisateur (ou s'il s'agit d'un groupe de représentants des utilisateurs), les besoins en formation sont réduits à néant par la forte participation du client et par les livraisons progressives fréquentes.
	Coûts de maintenance		
C8, C9	Coût de main-d'œuvre (maintenance évolutive et corrective)	Forte	Diminution : la qualité et la simplicité du code produit facilitent la maintenance ; les tests automatisés accélèrent et réduisent la maintenance corrective. Augmentation : l'approche minimaliste de la documentation peut conduire à des surcoûts en cas de discontinuité totale entre l'équipe de développement et l'équipe de maintenance. (S'il y a continuité, la programmation en binôme assure le passage de relais.)
	Coûts de débordement		-
C10	Coût de main-d'œuvre	Forte	Diminution : la démarche XP répond fondamentalement à ce problème, notamment par son approche de la planification.
C10	Coûts « business » dus au débordement	Variable	

	Typologie de coût	Importance relative du coût dans le projet (usuelle)	Impact potentiel d'XP sur le coût usuel
	Coûts d'inadéquation aux besoins réels		
C11	Coûts «business» dus à l'inadéquation aux besoins	Variable	Diminution : la présence du client sur site ; la démarche itérative réduit considérablement ce type de risque.
	Coûts de changement		
C12	Coût de main-d'œuvre	Forte	Diminution : XP intègre fondamentalement le changement dans sa démarche.
	Coûts de défauts de qualité		
C13	Coût de main-d'œuvre (maintenance)	Forte	Diminution :les tests automatisés permettent de réduire les défauts de qualité.
C13	Coûts «business» associés à un incident	Forte	
	Coûts de turnover		
C14	Coût de main-d'œuvre (formation, transition)	Moyenne	Diminution : XP améliore la motivation du développeur et réduit le taux de turn-over. Le travail collaboratif diminue l'impact d'un départ et facilite l'intégration des nouveaux. Augmentation : XP insiste fortement sur la formation continue de tous les personnels

Dans ce contexte très controversé, il est difficile de démontrer clairement la supériorité d'une approche XP globale par rapport à une méthode traditionnelle. Le nombre conséquent de paramètres en jeu rend l'exercice de style pratiquement impossible. Pourtant, certains gains possibles au travers d'une approche XP apparaissent nettement pour les éléments que nous venons d'évoquer. Toute la difficulté réside dans le fait que ces «gains évidents» ne sont pas les mêmes pour tous : le contexte de l'entreprise, la subjectivité de la mesure de certains indicateurs, agissent comme autant de facteurs d'influence dans la vision que l'on peut avoir de l'efficacité d'XP.

9

Aspects contractuels

Le droit est la plus puissante des écoles de l'imagination. Jamais poète n'a interprété la nature aussi librement qu'un juriste la réalité.

– Jean Giraudoux

La mise en œuvre de l'Extreme Programming au sein des entreprises françaises reste encore confidentielle. L'une des étapes clés dans la démocratisation de cette approche sera la capacité d'XP à s'inscrire dans un cadre contractuel qui permette au client et au prestataire (qu'il soit interne ou externe à l'entreprise) de bénéficier d'un certain nombre de garanties. Les ouvrages et les retours d'expérience sur XP occultent encore largement cette problématique contractuelle qui reste pourtant essentielle dans l'essor de cette approche. Il faut dire qu'une première lecture des pratiques XP a de quoi inquiéter un juriste. La minimisation du formalisme affiché comme étendard XP va à l'encontre d'une démarche contractuelle qui, justement, doit s'assurer que les deux parties contractantes ont bien formalisé par écrit l'ensemble des éléments qui leur apportent les garanties nécessaires.

Remarque

On notera que pour des développements internes ne faisant pas l'objet d'un contrat, d'autres éléments tiennent lieu de cadre contractuel.

La problématique contractuelle, passage obligé de l'Extreme Programming

XP est une démarche optimiste qui se focalise sur le succès du projet. L'existence d'un contrat en est d'autant plus nécessaire : même s'il s'inscrit dans une démarche positive, le contrat est

aussi là pour prévoir les cas où les choses ne se dérouleraient pas comme prévu. Il formalise les parties contractantes, les responsabilités, les échéances, les périmètres de réalisation, les livrables, toutes sortes de problèmes pouvant survenir. La liste des éléments pouvant rentrer dans le contrat peut être longue – sans qu'il soit d'ailleurs possible de se prémunir contre tous les dangers.

Comme nous allons le voir, la mise en œuvre d'un contrat dans le cadre d'une démarche XP n'est pas simple. Elle ne doit pas pour autant être éludée. Même dans le cas d'un projet où client et fournisseur font partie de la même entreprise, il est essentiel de formaliser un minimum les règles du jeu pour que chaque partie puisse réaliser ce que l'autre partie en attend.

Aujourd'hui, le faible retour d'expérience sur les projets XP ne facilite pas la tâche. L'approche contractuelle passe pour beaucoup par la jurisprudence : ce sont les expériences passées (les succès et surtout les échecs) qui permettent d'intégrer peu à peu dans les contrats de nouvelles clauses pour, au final, gérer un maximum de cas. Dans le cadre de l'Extreme Programming, le terrain est encore vierge et le plus simple est de se rattacher à des cadres contractuels existants pour voir comment ils peuvent s'inscrire dans une démarche XP ou comment ils peuvent évoluer pour s'en approcher.

Pour cela, il est essentiel de rappeler aujourd'hui les trois grands types de cadres contractuels dans lesquels travaillent les entreprises et leurs fournisseurs : le cadre forfaitaire, le cadre de l'assistance technique et le cadre de l'assistance technique dite forfaitée. Il nous a semblé également important de rappeler que les contrats, quel que soit leur type, mettaient eux-mêmes en relation deux grands types d'interlocuteurs : la maîtrise d'ouvrage et la maîtrise d'œuvre.

Terminologie

La *maîtrise d'ouvrage* représente l'utilisateur, c'est-à-dire la partie bénéficiaire du produit à réaliser. C'est elle qui détermine les fonctionnalités à réaliser, leur priorité, les délais à respecter…, et est responsable de l'expression des besoins et de la recette. Il s'agit souvent de l'entité qui finance le projet, mais pas nécessairement. La maîtrise d'ouvrage peut prendre une part plus ou moins active dans le pilotage du projet.

La *maîtrise d'œuvre* représente l'entité qui va prendre en charge la réalisation des travaux, et qui sera responsable de leur conformité avec les exigences de la maîtrise d'ouvrage. Elle spécifie et organise le travail nécessaire à l'aboutissement, en temps et en heure, d'un produit conforme au besoin exprimé par la maîtrise d'ouvrage. La maîtrise d'œuvre prend toujours une part active dans le pilotage du projet, tout au moins pour la gestion de ses propres ressources.

Ce binôme peut être associé à un modèle «client/ fournisseur», mais il est néanmoins important de faire une distinction. En effet, maîtrise d'ouvrage et maîtrise d'œuvre peuvent être prises toutes deux en charge par le client contractuel – dans ce cas, le fournisseur n'est sollicité que pour fournir de la main-d'œuvre. Inversement, la maîtrise d'ouvrage peut être partiel-

lement déléguée à un fournisseur, et la maîtrise d'œuvre à un second fournisseur. En fait, toutes les combinaisons sont à peu près possibles. Le contrat doit dans tous les cas clarifier ces aspects.

Contrats forfaitaires

Terminologie

Le contrat forfaitaire établit qu'un fournisseur s'engage à obtenir un résultat donné dans un délai donné et pour un prix fixé à l'avance.

Le contrat forfaitaire correspond à ce que les Anglo-Saxons ont l'habitude de dénommer par l'expression *Fixed Time*, *Fixed Price*. Ce type de contrat implique l'engagement à l'avance des deux parties sur trois éléments :

- **Les travaux à réaliser et les livrables à fournir.** C'est probablement l'aspect le plus complexe du contrat forfaitaire, puisque l'on suppose que ce qui sera réalisé est connu à l'avance avec exactitude. Or, c'est là pure vue de l'esprit puisque seul le travail de *spécifications* – généralement réalisé après démarrage du contrat – permet d'avoir une vue précise sur les travaux à réaliser…
- **La charge de travail à fournir**. Le fournisseur s'engage sur un prix fixe, et donc sur une charge de travail – rappelons que le fournisseur s'appuie sur une grille tarifaire journalière. Là encore, compte tenu de la difficulté qu'il y a à cerner à l'avance les travaux à réaliser, l'exercice de dimensionnement de la charge est extrêmement complexe. Les débordements de charge de travail dus à une mauvaise évaluation sont dans ce cas financés en principe par le fournisseur, et le client n'en subit pas directement les conséquences.
- **Les délais de réalisation**. Le fournisseur s'engage sur des dates pour la fourniture des livrables. Cela signifie qu'une mauvaise évaluation de la charge de travail est doublement pénalisante pour le fournisseur : il lui faudra non seulement fournir un travail supérieur à celui vendu mais, qui plus est, ce travail devra être exécuté par plus de personnes que prévu pour tenir des délais, ce qui peut représenter d'une part une contrainte très forte, notamment en cas d'indisponibilité d'ingénieurs supplémentaires, d'autre part un risque accru de nouveaux débordements[1].

Des pénalités de retard peuvent être mises en place dans les deux sens, à l'encontre du fournisseur s'il ne livre pas les travaux à temps, mais également, dans ce cas, à l'encontre du client s'il ne fournit pas à temps des éléments ou des décisions qui bloquent l'avancement des travaux.

1. Si l'on en croit la « loi de Brooks » qui veut qu'adjoindre des ingénieurs supplémentaires sur un projet revient à jeter de l'huile sur le feu, cela n'a pour effet que de retarder encore plus le projet. (*Adding manpower to a late software project makes it later.*)

Le contrat forfaitaire reste très utilisé en France. Que ce soit pour des administrations publiques ou pour des entreprises privées, ce type de contrat présente l'avantage de permettre une allocation budgétaire claire et une planification des travaux assez stricte. Lorsque le client ne possède pas de capacité de maîtrise d'œuvre, le forfait constitue également une solution séduisante : la maîtrise d'ouvrage reste côté client, la maîtrise d'œuvre côté fournisseur.

Limites du modèle forfaitaire

Pourtant, force est de constater que le modèle atteint rapidement ses limites, et ce pour plusieurs raisons.

Le rapport client/ fournisseur peut rapidement devenir conflictuel, notamment si le périmètre de réalisation n'a pas été clairement établi à l'avance, chaque partie ayant l'impression que l'autre ne pense qu'à tirer la couverture à soi : le client aura l'impression que des fonctionnalités identifiées au départ ne sont pas implémentées, alors qu'aux yeux du fournisseur le client sera perçu comme étant de mauvaise foi. Très vite, le rapport peut tourner à la bataille rangée (en particulier si des pénalités de retard sont en jeu), chacun faisant parler les clauses contractuelles ou essayant de faire dire au cahier des charges ce qui lui semble opportun. Lorsque les rapports prennent cette tournure, une seule chose est sûre : client et fournisseur perdront la partie, il n'y aura pas de gagnant. Les débats s'enlisent, la plupart du temps le projet capote, les équipes de part et d'autre passant plus de temps à se faire la guerre qu'à faire avancer les travaux. La motivation des équipes dans ce type de situation est naturellement très faible, ce qui ne favorise pas une issue positive.

Un cahier des charges extrêmement détaillé peut éviter ce genre de conflit – encore faut-il s'assurer de sa pertinence. Mais dans ce cas, le temps nécessaire à la réalisation du cahier des charges provoque un déphasage tel qu'entre le moment où les besoins ont été exprimés et le moment où ces besoins sont implémentés et en production, il n'est pas rare que le projet lui-même ait totalement perdu de son intérêt.

Le contrat forfaitaire est-il compatible avec une démarche XP ?

L'esprit d'un contrat forfaitaire s'oppose bien sûr par nature à celui d'une démarche XP. On peut même penser que les dérives du contrat forfaitaire ont contribué au mouvement XP.

L'Extreme Programming s'oppose à une démarche forfaitaire avant tout parce que le forfait suppose que (outre le prix) le périmètre fonctionnel est fixe ; or c'est justement sur cette variable qu'un projet XP va jouer le plus. Par ailleurs, il est fréquent que le cahier des charges ou la réponse à l'appel d'offres précise un planning et des jalons contractuels. L'objectif déclaré est de figer un maximum de détails, ce qui est assez loin de la souplesse qui doit être donnée à un client XP.

Pourtant, même les contrats au forfait peuvent parfois se dérouler dans un cadre XP, en particulier lorsque le client et le prestataire entretiennent une relation partenariale de longue date

et lorsqu'ils identifient ensemble un objectif commun dont le périmètre sera affiné au fil des itérations. Les cahiers des charges sont souvent assez généraux pour laisser une place importante à l'interprétation. Une négociation sur le périmètre fonctionnel peut avoir lieu. En particulier, les deux parties pourront envisager de remplacer certaines fonctionnalités décrites dans le cahier des charges par d'autres de volumétrie équivalente, en intégrant des avenants au contrat signé entre les deux parties.

Mais nous verrons dans les sections qui suivent d'autres alternatives permettant aux deux parties d'atteindre à coup sûr un but fixé, dans une démarche qui joue davantage sur le rapport «gagnant-gagnant» que peuvent nouer client et fournisseur.

Contrats d'assistance technique (ou «régies»)

Les contrats d'assistance technique – encore surnommés «régies», même si ce terme ne revêt pas de caractère légal – sont également très populaires en France. Ils ont fait le succès de grandes sociétés de services dont le modèle reste toutefois très controversé.

Terminologie

Le contrat d'assistance technique ou régie se caractérise par la mise à disposition de personnel compétent par le fournisseur, le client assurant dans ce cas la maîtrise d'ouvrage *et* la maîtrise d'œuvre. Le fournisseur facture au client le temps passé par ses collaborateurs, la responsabilité du succès des travaux ne lui incombant pas.

Ce modèle connaît un franc succès en France, car il permet aux entreprises de faire appel à du personnel qui a une formation pointue et des compétences sans l'embaucher, simplement donc pour la durée du projet cible. Même si la tarification à la journée reste élevée, la souplesse induite par le contrat (possibilité d'arrêter rapidement la prestation) et surtout l'accès à des compétences pointues sans avoir à fournir de formation reste un modèle très séduisant pour les grandes entreprises. L'absence de cahier des charges initial permet au prestataire de s'engager et motive le recours à ce type de contrat, car elle exclut de fait la possibilité de réaliser un contrat forfaitaire.

Limites du modèle régie

Ce modèle est-il plus favorable au succès des projets ? Rien n'est moins sûr. D'abord, parce qu'il ne fait que repousser la problématique de la maîtrise d'œuvre. Cette dernière passe du fournisseur au client. Mais quelle est l'expérience du client en matière de maîtrise d'œuvre ? Il faut prendre en compte la fréquence et la taille des projets menés en interne ; dans tous les cas, cette question doit être examinée avec attention.

La motivation, toute relative, de collaborateurs «mis à disposition» chez un client qu'ils n'ont pas choisi à l'avance peut également être un écueil.

Le contrat de régie est-il compatible avec une démarche XP ?

La réponse n'est pas évidente. Le contrat d'assistance technique ne garantit rien quant à la facilité de mettre en œuvre XP, ni dans un sens ni dans l'autre. Tout dépend de la manière dont le client assure sa responsabilité de maîtrise d'œuvre. Si le client travaille en interne dans une logique forfaitaire avec une rigidité très forte au changement, l'adoption d'une démarche Extreme Programming sera complexe.

En somme, si le client est capable de mettre en œuvre XP avec des ingénieurs puisés dans sa propre main-d'œuvre, rien ne l'empêche de le faire avec des ingénieurs en assistance technique. Dans le cas d'une première mise en œuvre d'XP, notamment si la volonté de changement en interne n'est pas très forte, le fait de faire intervenir des ingénieurs extérieurs peut même faciliter la transition. En particulier, l'assistance technique est sans doute la meilleure façon de trouver un coach expérimenté en XP.

Contrats d'assistance forfaitée : la combinaison des modèles régie-forfait ?

Les contrats d'assistance forfaitée sont en plein essor. Souvent difficiles à accepter par le client, ce sont pourtant eux qui font les grands succès. Mais ils nécessitent l'établissement d'une véritable relation de confiance entre les deux parties.

Ces contrats restent avant tout des contrats d'assistance technique car le mode de facturation du fournisseur demeure au temps passé. Mais à la différence des contrats d'assistance technique classiques, le fournisseur travaille selon une logique de maîtrise d'œuvre : ce ne sont pas des individus isolés qui interviennent chez le client, mais une équipe ayant en charge la réalisation du projet. Cependant, la souplesse qu'autorise le contrat d'assistance (par comparaison avec le forfait) permet aux deux parties d'itérer et de construire ensemble les spécifications détaillées. Ce type de contrat est adapté lorsque le client souhaite atteindre un but fixé (par exemple, la réalisation et la mise en ligne d'un site de commerce électronique) dans un délai précis (par exemple, six mois), et sur la base d'un budget évalué, mais *sans* avoir réalisé de cahier des charges détaillé au préalable.

Le budget client et les tarifications journalières du fournisseur vont permettre de «dimensionner» l'équipe adaptée. Ainsi constituée, celle-ci va travailler en étroite collabora-

tion avec le client pour atteindre l'objectif fixé, redéfinissant sans cesse les priorités et les fonctionnalités clés avec le client pour tenir le planning.

Exemple

Le client dispose d'un budget de 300 000 euros et doit sortir un site de commerce électronique dans un délai de 3 mois. Les tarifications fournisseurs sont de 900 euros/jour pour un chef de projet, et de 600 euros/jour pour un développeur. Comment dimensionner l'équipe ?

Les 3 mois représentent 60 jours. Le chef de projet est présent de bout en bout du projet ; il représente un budget de : 60 j × 900 € = 54 000 €. Le nombre de développeurs est donné par :

Nb Développeurs = (300 000 – 54 000) / (600 × 60) = 6,8

On notera que cet exemple reste simplifié, dans la mesure où il est nécessaire de comptabiliser dans le projet d'autres types de compétences que celles du chef de projet et des développeurs : ingénieur système et réseau, architecte, etc. Le travail de dimensionnement complet est donc un peu plus complexe mais il s'appuie sur le même type de démarche. Il peut également être souhaitable de mesurer l'ensemble des coûts et pas seulement ceux relatifs à la main-d'œuvre, comme cela a été vu au chapitre précédent.

Ce type d'approche peut naturellement soulever des interrogations, notamment en matière de responsabilité. L'impossibilité de distinguer la part de responsabilité des deux parties conduit généralement à sa prise en charge par le client.

Dans les faits, cette approche fonctionne quasiment à 100 % car la gestion du risque est prise très en amont. Lorsqu'une dérive se fait sentir, ce sont client et fournisseur qui analysent ensemble la source du problème et qui y remédient ensemble. Cela suppose naturellement que le fournisseur travaille en toute transparence avec le client, qui doit avoir une visibilité forte sur les capacités de l'équipe. En particulier, lorsque les risques de débordement se font sentir, les deux parties doivent revenir sur le périmètre des fonctionnalités afin d'étudier la façon la plus indiquée pour rester dans le budget et les délais requis.

L'avantage majeur de cette approche est qu'elle met les deux parties dans une logique gagnant-gagnant : un but à atteindre à une date fixée, une équipe pour atteindre cet objectif, avec à la clé soit le succès qui satisfera les deux parties, soit l'échec qui sera analysé et qui pourra se traduire par l'évincement du fournisseur. La démarche itérative et la très forte proximité entre le client et le fournisseur permettent généralement de juger dès les premières semaines si le tandem constitué par le client et le fournisseur fonctionne. Soit les parties constatent l'impossibilité de travailler ensemble, soit un véritable partenariat se développe entre les parties.

Il est très rare qu'un client accepte ce type de contrat d'emblée, sans connaître le fournisseur, ou son équipe. La prise en charge de la responsabilité par le client en est la principale raison. En revanche, ce type de contrat est fréquemment adopté après un premier forfait : le client a appris à connaître les équipes de son fournisseur, une relation de confiance s'est engagée entre

les parties, et le client souhaite donner de nouveaux travaux à l'équipe mais dans un cadre plus souple : c'est un contrat d'assistance technique qui sera signé, mais l'équipe restera dans une logique forfaitaire, très itérative et très proche du client.

Mise en œuvre d'XP dans un cadre d'assistance technique forfaitée

Le contrat assistance technique forfaitée est celui qui, naturellement, reste le plus adapté à une démarche de type Extreme Programming. Il s'inscrit en effet dans une logique très itérative, complètement en phase avec XP.

« XP » contractualisé ?

Contractualiser XP reviendrait à faire figurer en bonne place dans le contrat les éléments suivants :

- le cycle de vie itératif, avec livraison rapide et régulière d'un logiciel opérationnel en état de marche ;
- les droits du client de changer d'avis sur le contenu fonctionnel ou les priorités ;
- la colocalisation du client (ou d'un représentant ayant tous pouvoirs) et de l'équipe du fournisseur. Si cette colocalisation n'est qu'à temps partiel, les modalités doivent en être précisées ;
- la fourniture *par le client* et *à l'avance* de tests fonctionnels automatiques et l'utilisation de ces tests comme critère unique de recette de chaque livraison ;
- la possibilité pour le client de se dégager du contrat à tout moment, sur décision motivée et à condition d'avoir rémunéré le prestataire pour, au minimum, toutes les itérations qui ont fait l'objet d'une recette-livraison.

C'est sur cette dernière clause qu'il convient bien entendu d'être très prudent, pour l'une comme l'autre des parties; en tout état de cause nous recommandons de faire valider tout cadre contractuel proposé par un juriste ou autre personne compétente.

Dans les sections suivantes, nous allons donner quelques compléments d'informations qui pourront utilement aider à la mise en œuvre XP dans le cadre de ces contrats d'assistance forfaitée.

Le contrat peut s'appuyer sur un plan d'assurance qualité

La mise en place d'un plan d'assurance qualité entre les deux parties (voir chapitre 10), qu'il soit adjoint au contrat ou non, peut se révéler un complément important pour le client comme pour le fournisseur. Le plan d'assurance qualité permet de définir clairement le mode de fonctionnement des équipes tout au long du projet : acteurs, rôles, livrables, étapes du projet, etc.

Par la définition qu'il donne d'un certain nombre de termes, il permet également de s'assurer que les deux parties ont la même compréhension de ce qu'est une livraison, une recette, etc.

Ainsi le plan d'assurance qualité constitue-t-il un garde-fou en stipulant clairement quels moyens sont engagés, et selon quel mode de fonctionnement, pour atteindre l'objectif, ce qui à défaut d'avoir pu définir exactement le périmètre de l'objectif est déjà un point essentiel.

Nécessité de distinguer maîtrise d'ouvrage et maîtrise d'œuvre

Un certain nombre de règles définies dans les paragraphes précédents sont bien sûr appliquées dans le cadre de ces contrats «XP», en particulier la nécessité de définir clairement qui est en charge de la maîtrise d'ouvrage (celui qui définit les fonctionnalités) et qui est en charge de la maîtrise d'œuvre (celui qui les réalise).

Comme nous l'avons vu au chapitre 2, XP définit un certain nombre de rôles précis dans le projet. Certains sont facilement identifiables : ainsi, le «client» représente bien ce que l'on appelle la maîtrise d'ouvrage. En revanche, la maîtrise d'œuvre reste plus difficile à identifier. En effet, la démarche privilégie un rapport très fort et direct entre le client et les développeurs ; doit-on en conclure que la maîtrise d'œuvre est représentée par les développeurs ? Certainement pas, d'un point de vue contractuel.

De ce point de vue, la maîtrise d'œuvre est plutôt prise en charge par le «coach» de l'équipe, que l'on pourrait rapprocher du «chef de projet» – même si cette dénomination n'est pas utilisée dans le schéma proposé par XP. Selon les cas, le coach pourra faire partie de l'équipe du fournisseur ; mais il pourra également être désigné par le client lui-même (d'où la nécessité de bien distinguer le client de la maîtrise d'ouvrage proprement dite, le client au sens contractuel du terme pouvant englober davantage que la maîtrise d'ouvrage).

Dans tous les cas, les champs de responsabilité des différents intervenants doivent être établis au préalable. Le rôle du coach vis-à-vis de la maîtrise d'ouvrage et des développeurs doit être clair.

Documentation et standards de code

Il est essentiel de préciser contractuellement ce qui devra être livré en termes de documentation. Documentation d'architecture globale, documentation de mise en production, peuvent être considérées par le client comme étant incontournables. Même si XP invite à une documentation «légère», la documentation reste un droit du client : celui-ci pourra donc exiger par contrat la livraison d'un ensemble de documents. Il faudra veiller dans ce cas à ce que les tâches de réalisation de ces documents soient bien prises en compte dans la planification du projet, au même titre que les besoins fonctionnels : concrètement, chaque document (ou chaque portion d'un document plus important) fera l'objet d'un «scénario documentation», assorti d'un «test de recette».

Scénarios et tests documentaires

Un test de recette pour un scénario de documentation pourrait par exemple prendre la forme suivante : « Fournir une documentation d'architecture : acceptée, si elle permet à un développeur de maintenance d'identifier en moins de deux minutes le module auquel appartient la classe "RoutageDesOrdres". »

Le client peut souhaiter obtenir des garanties en termes de documentation du code : il pourra alors exiger, contractuellement, que soient utilisées des normes spécifiques de notation, et demander à l'équipe que le code fasse l'objet de commentaires conformes à ces règles, pouvant éventuellement être extraits automatiquement par le biais d'outils du type JavaDoc – pour créer par exemple une documentation de référence des librairies de fonctions ou d'objets. Là encore, il est souhaitable de fixer des critères d'acceptation portant sur des objectifs fonctionnels («Permettre la prise en main par un développeur familier avec les technologies XML») plutôt que d'imposer des règles dont le caractère arbitraire risquerait de démotiver les programmeurs.

Client sur site – déclinaisons possibles

Le contrat pourra définir les modalités de la présence du client (au sens maîtrise d'ouvrage) sur site. Plusieurs modes opératoires sont possibles :

- Le client est en permanence sur le site du fournisseur, et travaille avec l'équipe.
- Le client est à temps partiel sur le site, sur des créneaux journaliers définis.
- L'équipe de développement se déplace chez le client.

Ce dernier mode de travail peut être l'un des plus efficaces : l'équipe a directement accès à l'infrastructure cible et peut facilement rencontrer différents interlocuteurs chez le client si le projet l'exige. Ce mode de fonctionnement permet de lever plus rapidement des doutes et d'améliorer le *feedback* utilisateur. En revanche, il peut être pénalisant car il isole l'équipe de l'environnement humain et technologique dans lequel elle a l'habitude de travailler.

Au-delà de la présence du client, sa réelle disponibilité vis-à-vis des équipes pourra faire l'objet de clauses contractuelles.

Déploiement – recette

Que ce soit dans le plan d'assurance qualité ou le contrat lui-même, les modalités de déploiement des fonctionnalités, de recette et de mise en exploitation doivent là aussi être explicitées. Sur ces aspects, les préconisations XP qui privilégient une intégration continue (voir chapitre 5) sont plus ou moins simples à mettre en œuvre.

En effet, le processus d'intégration peut nécessiter, sur des projets complexes fortement liés à l'infrastructure du client, l'intervention d'équipes tierces : administrateurs réseaux, responsables sécurité, etc. La mise à disposition fréquente du personnel correspondant n'est pas toujours possible. De ce fait, il est essentiel de définir à l'avance le mode d'organisation adopté.

Les étapes de recette seront facilitées par le déroulement automatisé des tests de recette, qui sont implémentés en même temps que les fonctionnalités correspondantes tout au long projet.

Remarque

Les tests de recette couvrent généralement les aspects fonctionnels du produit, mais la complexité des architectures mises en œuvre implique souvent des tests qui dépassent le cadre applicatif : tests de montée en charge, tests de reprise sur incident, test de sécurité avec tentatives d'intrusion… Il faudra donc veiller à ce que les besoins correspondants soient exprimés au cours du projet, et que les tâches correspondantes soient planifiées, de sorte que les tests de recette associés soient implémentés et disponibles au moment de la recette.

Indicateurs de suivi possibles

L'engagement du fournisseur n'étant pas forfaitaire, il peut être utile de définir des indicateurs de suivi qui permettent de surveiller la bonne marche des travaux. On pourra par exemple s'appuyer sur des indicateurs XP.

Les scénarios client réalisés et la vélocité de l'équipe

La démarche XP a l'avantage d'être orientée sur des cycles itératifs avec livraison de scénarios client. La livraison ou non des scénarios client et leur rythme de livraison constituent des indicateurs. Plus généralement, l'évolution de la vélocité de l'équipe peut constituer un moyen de surveiller l'activité. Il faut toutefois veiller à manipuler cet indicateur avec parcimonie : la vélocité est avant tout un outil de planification, qui permet aux développeurs de proposer des estimations fiables sans qu'ils ressentent le besoin de « gonfler » leurs estimations pour se protéger. L'utiliser comme indicateur de performance d'une manière abusive, c'est risquer de fausser les estimations et par conséquent de mettre à mal la fiabilité du planning.

Les tests

L'écriture des tests de recette en parallèle avec le code donne également des assurances d'un point de vue contractuel en matière de conformité avec les exigences requises des fonctionnalités livrées. Encore faut-il que les tests soient suffisamment exhaustifs et représentatifs du niveau d'exigence attendu : le client devra donc réellement jouer le rôle que lui réserve XP et

s'assurer de bien spécifier tout au long du projet des tests qui lui apportent les garanties adéquates.

Limites du contrat : nécessité de la confiance

Nous avons illustré dans la section précédente quelques pistes qui facilitent l'établissement d'un contrat. Dans tous les cas, il faudra s'appuyer sur l'existant et essayer de le faire évoluer de façon à l'adapter à un contexte XP. Cependant, tous les juristes l'avoueront : à force de tout contractualiser, on finit par entraver la bonne marche des opérations…

Dans un contexte XP, la nature même de la démarche impose de privilégier la confiance par rapport aux négociations contractuelles. C'est la gestion des risques opérée très en amont qui offre la meilleure garantie aux deux parties. Des joutes contractuelles prolongées où client et fournisseurs ont l'impression permanente que l'autre essaie de tirer la couverture à lui ne facilitent pas le fonctionnement serein d'un projet XP – mieux vaut dans ce cas considérer une autre approche qui puisse donner des garanties contractuelles plus fortes aux deux parties.

Difficultés de mise en œuvre en France

Une culture française cartésienne

L'Extreme Programming connaît un franc succès aux États-Unis. Rien d'étonnant à cela. La culture américaine privilégie le pragmatisme, le «gagnant-gagnant» et les approches dites «bottom-up», c'est-à-dire issues de la réalité du terrain, à périmètre local, mais n'ayant pas nécessairement d'ambition systémique. L'Extreme Programming, mais aussi UML quelques années plus tôt, s'inscrivent dans cette logique.

La France hérite d'une culture cartésienne radicalement différente, qui a toujours privilégié les approches «top-down», c'est-à-dire issues du *top management*, à vocation systémique, et déclinées sur les unités opérationnelles, à l'image de Merise. Naturellement, la culture française n'est donc pas XP, et c'est dans le contrat que cet écart se cristallise le plus.

Évangélisation des directions administratives et financières

Les directions administratives et financières restent généralement très attachées aux contrats forfaitaires, car ces derniers permettent d'avoir une vue budgétaire prévisionnelle claire, dans un périmètre bien défini. La satisfaction du client quant à la conformité des réalisations par rapport aux besoins *réels* des utilisateurs ne fait généralement pas partie des considérations financières, parce qu'elle est difficilement mesurable.

Or, on a encore peu de recul quant à la mise en œuvre contractuelle et au retour réel sur investissement des projets XP, ce qui invite naturellement à la prudence des départements dont l'une des valeurs fortes est justement… la prudence.

Un véritable travail d'évangélisation auprès des directions administratives et financières s'impose donc afin qu'XP ne soit pas considéré, à tort, comme un moyen de se soustraire aux contraintes de rentabilité, ni comme prétexte à démission des deux parties opérationnelles dans le domaine contractuel. À ce titre, il faudra attendre les premiers retours d'expérience, voire les premiers succès pour que l'adoption d'une telle démarche soit acquise.

10

Qualité, processus et méthodologie

La seule chose à faire d'un bon conseil est d'en faire profiter un autre. On n'en a jamais l'usage soi-même.

– Oscar Wilde

L'Extreme Programming propose une méthode très complète et autonome de développement logiciel, couvrant le recueil des besoins, la planification et le pilotage de projet, ainsi bien sûr que tous les aspects du développement à proprement parler : tests, codage, spécifications.

Cependant, XP peut être amené à s'inscrire dans un cadre plus large, par exemple lorsque la méthode est mise en œuvre au sein d'une organisation qui possède déjà une culture méthodologique, un système de management de la qualité ou une démarche d'amélioration des processus logiciels. Une telle cohabitation est notamment indispensable lorsque l'organisation concernée fait du développement de systèmes, où le logiciel n'est qu'une composante, même si elle est prépondérante.

Ce chapitre vise donc à situer l'Extreme Programming dans le paysage méthodologique et doit pouvoir servir de guide pour l'insertion d'XP dans un système complémentaire, existant ou à venir.

Nous espérons aussi contribuer à une meilleure compréhension mutuelle entre des mondes qui paraissent parfois antagonistes. Si, à la lecture de ce chapitre, un pratiquant XP vient à s'intéresser par exemple à l'ISO9001, ou si un chef de projet expérimenté et satisfait de la méthode RUP décide qu'il peut ajouter une corde à son arc en essayant XP, nous aurons atteint au moins partiellement notre objectif.

De la qualité au processus

Aucune méthode n'a jamais pu garantir qu'au bout du compte, le client serait satisfait. C'est là qu'interviennent le contrôle et l'assurance qualité. Le contrôle qualité consiste à vérifier (rétrospectivement) que le produit est satisfaisant. L'assurance qualité consiste quant à elle à s'assurer (et assurer le client) que la façon dont est réalisé le logiciel garantit que le client sera satisfait du résultat. Pour cela, on ne peut se contenter de contrôler la qualité : en cas d'anomalie (défaut, non-satisfaction du client…), il faut à la fois remédier au problème lui-même et en analyser et supprimer la cause.

Remarque

Dans de nombreuses entreprises des secteurs industriels, mais également, de plus en plus souvent, dans les secteurs tertiaires, ces activités sont dévolues à un département distinct et bien identifié. C'est notamment le cas dans de nombreuses entreprises de l'industrie du logiciel. XP recommande que le département Qualité, s'il existe, soit fortement impliqué dans l'élaboration des tests de recette, mais ne fait pas par ailleurs de recommandations précises quant à l'organisation de la fonction qualité.

Pour analyser et supprimer les causes de non-conformité, il faut comprendre comment est produit le logiciel : cela suppose que l'on identifie ce qui est nécessaire en amont, que l'on analyse comment travaille l'équipe de développement (qui utilise une méthode, mais sans doute se passe-t-il aussi des choses qui ne font pas partie de la méthode…), que l'on identifie aussi les interfaces avec d'autres équipes, que l'on mesure la qualité du résultat. En bref, il faut comprendre le processus complet, et pouvoir le modifier pour supprimer les causes qui influent négativement sur la qualité. Les développeurs (et plus largement les autres intervenants du projet), mettent en œuvre une méthode, mais font partie d'un processus.

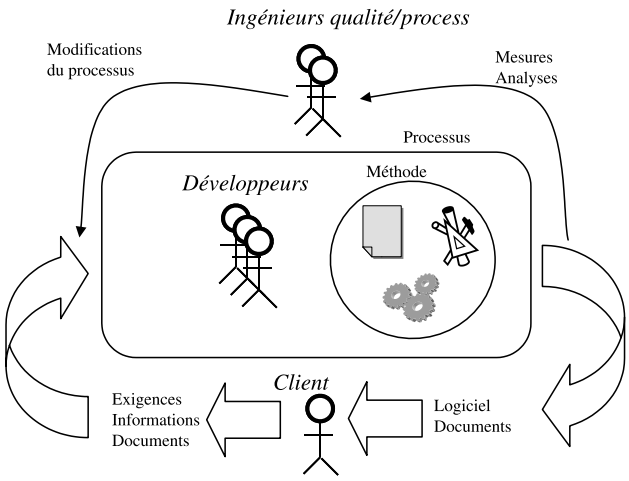

Figure 10-1. Méthode, qualité, processus

Comme l'indique la figure 10-1, les ingénieurs qualité mesurent et analysent la qualité du produit (en fait, la satisfaction du client) afin d'ajuster le processus. Ce principe se révèle si puissant qu'il peut être mis à profit pour d'autres objectifs que la seule satisfaction du client : c'est l'objectif des efforts dits d'*amélioration des processus*. Un ingénieur process va mesurer et analyser la qualité mais aussi d'autres paramètres (la productivité, la capitalisation du savoir-faire, la satisfaction et l'évolution des développeurs…), et ajuster le processus pour améliorer tout ce qui doit l'être.

L'ISO et le management de la qualité

Termes, définitions et normes

Les définitions suivantes sont généralement admises, lesquelles reprennent pour l'essentiel les définitions de l'ISO.

ISO – Organisation internationale de normalisation (*International Organization for Standardization)*

Le mot grec *isos* signifie « égal ». Basée à Genève, l'ISO est chargé de favoriser le développement de normes internationales dans tous les domaines. Elle fédère quelque 140 organismes nationaux de normalisation, tels l'AFNOR en France et l'ANSI au États-Unis.

La qualité d'un produit est sa capacité à satisfaire le besoin et les attentes du client. Les caractéristiques intrinsèques de qualité pour un logiciel sont la capacité fonctionnelle, la fiabilité, la facilité d'utilisation, le rendement, la maintenabilité et la portabilité.

Le contrôle qualité, qui est apparu dans les années 1920 avec les chaînes de production, consiste à vérifier rétrospectivement la qualité d'un produit, généralement au moyen d'échantillonnages, puis à appliquer des mesures correctives en cas de constat de défauts. Dans les années 1980 voit le jour le concept d'assurance de la qualité, qui consiste à définir la manière de travailler (la procédure) assurant au client un certain niveau de qualité mais aussi permettant de démontrer que la procédure a été respectée. Les actions préventives, consistant à éviter l'apparition de défauts, visent à réduire la nécessité d'actions correctives, plus coûteuses.

L'ISO définit une famille de normes dite ISO9000 (comprenant plusieurs normes, notamment l'ISO9001) qui fournit un cadre permettant à des entreprises de nature très diverses (industrie, services, artisanat…) de se doter d'un système d'assurance qualité et de se faire certifier. Les pays européens adoptent massivement la norme ISO9000. Le Japon et les États-Unis la boudent : l'ISO9000 ne vise que la satisfaction du client, alors qu'ils courent déjà après un nouvel idéal : la qualité totale. Cette dernière tend à couvrir trois objectifs : satisfaire tout à la fois le client, les actionnaires (en générant durablement des bénéfices) et les salariés (plus généralement, répondre aux aspirations du corps social).

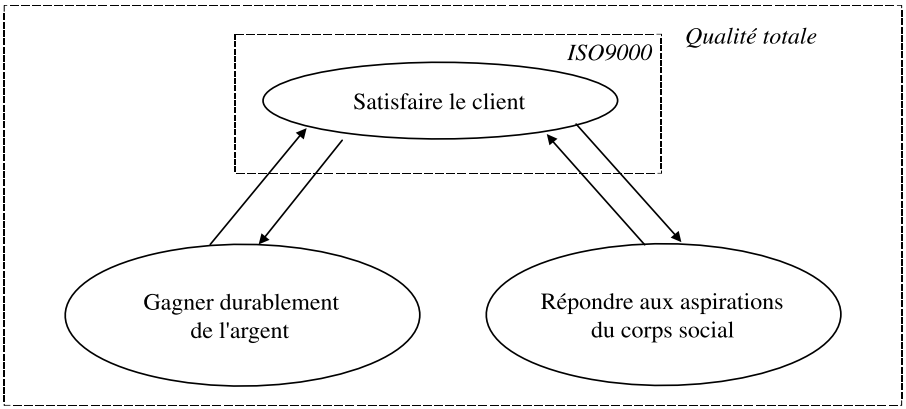

Figure 10-2. Périmètre de l'ISO9000 et de la Qualité Totale

Cependant, les trois objectifs de la qualité totale sont très interdépendants. En outre, en introduisant des considérations économiques et sociales, il devient très difficile de définir une norme qui couvre la diversité financière, juridique et sociale des entreprises. Américains et Japonais en viennent donc à adopter l'ISO9000.

Cette norme a connu de récentes évolutions. Sa nouvelle version, dite ISO9000:2000, est sortie en l'an 2000 – la norme originale étant nommée ISO9000:1994. Désormais, on parle de système de management de la qualité, ce qui traduit deux tendances : d'abord, au lieu d'assurer un niveau constant de qualité, c'est une amélioration continue qui est recherchée ; ensuite, ce n'est plus la seule qualité du produit qui est visée mais la qualité des processus employés. La qualité est dès lors considérée comme un outil de management.

La qualité dans XP

La notion de «qualité» dont il est question en XP est-elle bien la même que celle dont la norme ISO9000 fait l'objet ?

Il faut tout d'abord noter que les principes XP de conduite de projet font explicitement apparaître la qualité du logiciel comme une variable de pilotage. C'est d'ailleurs la seule des quatre variables (qualité, temps, périmètre et coût) avec laquelle il est déconseillé de jouer : la qualité est bien l'un des principaux objectifs de la méthode XP, avec la rapidité de développement. La qualité dont on parle ici concerne bien avant tout la satisfaction du client : le logiciel doit lui rendre service et être exempt de défauts.

Mais pour les développeurs XP, la qualité se mesure également dans le code. Un code de qualité présente à la fois des caractéristiques concrètes de simplicité, de lisibilité, de maintenabilité et des caractéristiques subjectives, quelque chose qui échappe à l'analyse et qui, aux yeux de certains programmeurs, lui confère une certaine beauté. XP met explicitement en valeur ce côté créatif, voire artistique de la programmation.

Cette vision des choses est troublante pour un ingénieur qualité ISO9000, en raison de la difficulté qui peut être rencontrée pour décrire un processus reproductible qui permettra d'obtenir systématiquement un code dont la «qualité» est de cette nature, mais aussi parce qu'il est difficile de définir des critères objectifs permettant de dire si le but a été atteint.

C'est là un différend presque philosophique dont il faut s'accommoder. Comme nous allons le voir, la méthode XP présente par ailleurs de nombreuses caractéristiques très favorables à son utilisation dans le cadre d'un système qualité ISO9000.

Convergences entre XP et les huit principes de l'ISO9000

Nous n'allons pas, dans cet ouvrage, nous livrer à une étude détaillée des normes ISO9000. Dans ses nouvelles normes ISO9000:2000 et ISO9004:2000, l'ISO définit huit principes de base. Afin de mieux cerner en quoi la méthode XP et un système qualité ISO9000 peuvent être compatibles, nous allons tout d'abord examiner ces huit principes et voir ensuite comment l'Extreme Programming peut éventuellement s'y inscrire.

Principe 1 – L'orientation client

> «Les organismes dépendent de leurs clients ; il convient donc qu'ils en comprennent les besoins présents et futurs, qu'ils satisfassent leurs exigences et qu'ils s'efforcent d'aller au-devant de leurs attentes.»

Il est évident que, sur ce plan, l'Extreme Programming est une méthode particulièrement performante. L'importance donnée au client est telle qu'il fait partie intégrante de la méthode et même de l'équipe chargée du projet. La proximité physique, la disponibilité et la participation active du client ou de son représentant sont autant d'exigences mises en avant par la méthode. Sa mise en œuvre complète va dans le sens d'une compréhension parfaite des besoins du client.

Principe 2 – Leadership

> «Les dirigeants établissent la finalité et les orientations de l'organisme. Il convient qu'ils créent et maintiennent un environnement interne dans lequel les personnes peuvent pleinement s'impliquer dans la réalisation des objectifs de l'organisme.»

Ce principe concerne surtout, mais pas uniquement, des questions de management d'entreprise. On peut cependant noter des résonances avec la façon dont XP conçoit la direction des équipes. Ainsi l'ISO compte-t-elle au rang des bénéfices à retirer de ce principe «l'établissement de la confiance et l'élimination de la peur». On retrouve là deux des valeurs d'XP, communication et courage.

Plus concrètement, l'allusion à l'environnement interne participe d'une préoccupation récurrente pour les pratiquants d'XP : les programmeurs doivent bénéficier d'un environnement de travail idéal, et c'est au manager de s'en assurer (voir la description du rôle de manager au chapitre 2). On peut également penser que les recommandations de l'ISO visant à «créer et faire vivre des valeurs partagées», «inspirer, encourager et reconnaître les contributions de chacun»sont les fondements mêmes du rôle de coach.

Principe 3 – Implication du personnel

> «Les personnes à tous niveaux sont l'essence même d'un organisme et une totale implication de leur part permet d'utiliser leurs aptitudes au profit de l'organisme.»

On touche encore une fois un point central de la méthode XP : les individus sont plus importants que les technologies ou les processus. La méthode XP met les personnes en valeur, favorise leur épanouissement, responsabilise les individus et, ce faisant, atteint le double objectif de rendre l'équipe plus performante tout en faisant le bonheur des participants.

Principe 4 – Approche processus

> «Un résultat escompté est atteint de façon plus efficiente lorsque les ressources et activités afférentes sont gérées comme un processus.»

L'adéquation entre ce principe et la méthode XP n'apparaît pas de façon immédiate. D'un côté, XP s'insurge précisément contre cette tendance à faire du développement logiciel un processus presque industriel, récuse l'idée selon laquelle, en matière de logiciel, on peut «systématiquement définir les activités qui mènent à un résultat voulu» et insiste sur la part de créativité, l'indispensable ingrédient humain de cette activité.

Il ne faut pas y voir une lecture mystique du développement logiciel, ni un manque de sérieux. Au contraire, l'Extreme Programming est une méthode reposant sur une forte discipline et dont les règles permettent d'obtenir de façon efficace et fiable les ingrédients nécessaires à la réussite d'un projet logiciel, y compris les aspects moins rationnels, moins mécaniquement reproductibles. Par ses règles et pratiques bien définies, par son côté empirique et mesurable aussi, la méthode XP est à même de satisfaire les partisans d'une vision «processus».

Compte tenu de la cohérence et de l'interdépendance des pratiques XP, il est important de considérer l'ensemble de la méthode XP comme un processus indivisible, produisant du logiciel à partir de l'identification d'un besoin ; on ne saurait l'interpréter comme un ensemble relativement arbitraire de pratiques mises bout à bout. En outre, XP possède la caractéristique qui définit selon l'ISO un processus : la capacité de s'améliorer grâce à du feedback. Le feedback est une des valeurs d'XP. Il se concrétise au moyen de bien des techniques et indicateurs. Le cycle itératif d'XP permet d'utiliser ce feedback pour améliorer continuellement le

processus. Les plus grands adeptes d'XP ne voient la méthode elle-même que comme un point de départ : une fois les pratiques maîtrisées, l'amélioration du processus peut conduire à adapter la méthode aux circonstances ou à l'entreprise, à condition d'en respecter les valeurs et principes (voir principe 6).

Principe 5 – Management par approche système

> « Le fait d'identifier, de comprendre et de gérer des processus corrélés comme un système contribue à l'efficacité et l'efficience de l'organisme à atteindre ses objectifs. »

La méthode XP est elle-même issue d'une approche systémique aux problématiques de développement logiciel – les idées de Gerald Weinberg, notamment dans *Systems Thinking*, ont joué un rôle important dans l'élaboration de la méthode. Cette approche consiste pour l'essentiel à définir un « système » comme un ensemble d'éléments dont l'interaction est régie par les « processus » visés par le principe 4, et à l'analyser comme un tout sans nécessairement le réduire à la superposition linéaire de ses éléments.

Pour autant, cette approche systémique ne joue pas un rôle majeur dans l'application au quotidien de la méthode XP elle-même, et bien qu'elle puisse utilement guider la gestion des relations entre un projet XP et les autres activités d'un organisme, elle s'inscrit plus, comme le principe 2, dans le contexte du management d'une organisation que dans celui de la conduite d'un projet logiciel donné.

Principe 6 – Amélioration continue

> « Il convient que l'amélioration continue de la performance globale d'un organisme soit un objectif permanent de l'organisme. »

Dans le périmètre qui est le sien, la méthode XP contribue de deux façons à l'objectif d'amélioration continue. Tout d'abord, la méthode prône la notion d'amélioration continue au sein du projet. C'est une des raisons d'être du développement itératif, qui permet une remise en cause à chaque cycle à la fois des produits (le logiciel, le code, les outils) et des méthodes de travail. Comme on le dit en XP, « ce ne sont là que des règles ».

Il est important de bien avoir à l'esprit que les règles et pratiques XP ne sont là que pour servir un but. Une équipe XP chevronnée, ayant suffisamment d'expérience avec XP pour en maîtriser les mécanismes, est invitée à développer un esprit critique envers la méthode elle-même. Il est tout à fait permis de modifier, supprimer ou ajouter des règles pendant une période d'observation (typiquement une itération), afin de remédier à un problème constaté ou simplement afin de tenter de faire encore mieux. Le principe XP, « Jouer pour gagner », aiguillonne ici la motivation de l'équipe : elle ne se contente pas d'une méthode qui marche, mais cherche en permanence à l'améliorer.

Ensuite, du point de vue de l'organisation au sein de laquelle on pratique XP, la méthode est bénéfique dans la mesure où elle favorise grandement l'amélioration des compétences individuelles. En permettant à des développeurs débutants de participer aux activités traditionnellement réservées aux développeurs chevronnés (conception, spécifications), en exigeant que chaque développeur devienne compétent sur l'ensemble des techniques et technologies utilisées et sur l'ensemble des fonctionnalités du système, l'Extreme Programming tend à améliorer le développement personnel des développeurs, ce qui contribue à l'amélioration continue du potentiel de l'organisation.

Principe 7 – Approche factuelle pour la prise de décision

> « Les décisions efficaces se fondent sur l'analyse de données et d'informations. »

Ce principe fait profondément partie de la culture XP et peut être associé au précédent : l'amélioration des méthodes, les changements de règles mais aussi les décisions de conception ne sont mis en œuvre que sur la base d'un observable : la vélocité a-t-elle augmenté depuis la dernière itération ? Les régressions dans les tests de recette sont-ils moins fréquents ? Peut-on imaginer un test unitaire qui démontre que telle conception est supérieure à telle autre ? En outre, XP est particulièrement pragmatique quant aux observables : par exemple, seule la progression dans les tests de recette tient lieu de mesure d'avancement (les documents, les lignes de code et autres produits de travail ne sont pas des mesures utilisées).

Sur un tout autre plan, XP insiste pour que ces données et informations factuelles soient diffusées le plus largement possible au sein du projet, sans cloisonnement ni obstacle, de façon que la prise de décision de chacun en soit rendue plus efficace. C'est ainsi qu'un programmeur sera encouragé à annoncer un retard éventuel sur les tâches qui lui sont attribuées, ou que le client divulguera aux programmeurs son intention de considérer une tâche plus prioritaire qu'une autre. Choisir une « culture du secret » ou cloisonner la diffusion des informations, c'est priver des intervenants du projet des informations qui leur sont nécessaires pour être efficaces.

Principe 8 – Relations mutuellement bénéfiques avec les fournisseurs

> « Un organisme et ses fournisseurs sont interdépendants et des relations mutuellement bénéfiques augmentent les capacités des deux organismes à créer de la valeur. »

Ce principe exprime précisément la relation que la méthode XP vise à instaurer entre le client et les développeurs (qui sont les fournisseurs). En parlant d'interdépendance, il souligne l'absurdité des relations contractuelles trop antagonistes, où chaque partie chercherait à prendre l'avantage sur l'autre. En parlant de création de valeur mutuelle, il préconise tout simplement les relations « gagnant-gagnant » que nous avons évoquées à plusieurs reprises

dans ce livre. XP prend tout son sens et toute sa valeur quand la méthode est souhaitée et appliquée autant par le client que par les développeurs. Lorsque le client est aussi celui qui finance le projet, cette relation de partenariat voulue par XP (et par ce principe ISO) doit s'étendre au domaine financier et contractuel (voir le chapitre 9, «Aspects contractuels»).

Mise en place d'XP dans un contexte ISO9001

Sur le fond, on le voit, la méthode XP présente de nombreux atouts qui devraient permettre de l'intégrer à un système de management de la qualité moderne. Néanmoins, il existe, en France, un certain décalage entre la culture XP et la culture ISO9001 traditionnelle. Généralement synonyme de lourdeur, de paperasse, l'assurance qualité à la française risque d'être quelque peu bousculée par cette méthode iconoclaste au nom inquiétant. Pourtant, l'Extreme Programming a de quoi séduire un directeur qualité à l'esprit assez ouvert et moderne. En outre, aux États-Unis, les termes «assurance qualité» couvrent aussi le test logiciel, domaine dans lequel la méthode XP excelle.

Tout tient sans doute dans la forme, dans la manière de présenter la méthode en mettant en valeur les atouts que l'on vient d'énumérer en termes d'assurance qualité. Dans les sections qui suivent, nous présentons quelques approches susceptibles d'améliorer la perception de la méthode XP sans la dénaturer ni en perdre les bénéfices.

La définition du cycle de vie

Compte tenu de la nature *procédurale* de l'ISO9001/1994, il est fréquent que les systèmes qualité imposent un cycle de vie pour les développements, et, le plus souvent, il s'agit du cycle en V. Normalement, un système qualité doit permettre à un projet de déroger au cycle de vie standard de l'entreprise, mais cela demande un travail supplémentaire consistant à décrire le cycle de vie et la méthode employée.

Les plans (qualité, développement…)

Écrire ce que l'on fait et faire ce que l'on écrit. Même s'il n'apparaît pas dans les huit principes énoncés plus haut, ce concept est profondément ancré dans la culture ISO9001 – comme d'ailleurs dans celle, voisine, de la démarche CMM (*Capability Maturity Model*). Un respect trop strict de ce principe peut faire craindre une certaine rigidité, mais il n'en est rien dans la réalité, puisqu'il est parfaitement admis que l'on puisse modifier ce que l'on fait – à condition de changer d'abord ce que l'on a écrit…

Définition

Le CMM (*Capability Maturity Model*) est un modèle permettant d'évaluer et d'améliorer la «maturité» d'une organisation au regard de son aptitude à réussir ses projets et développer ses produits. Elle définit cinq niveaux croissants de maturité, pour lesquels un ensemble de processus et d'activités sont requis.

Pour un développement logiciel, cela se traduit par la rédaction d'un plan, généralement appelé plan de développement, parfois décomposé en plusieurs plans (d'assurance qualité, de développement, de gestion de configuration). Peu importe le détail : pour un projet XP, il s'agira d'une contrainte spécifique, celle qui consiste à rédiger un plan qui décrit de quelle manière sera développé le projet. En outre, comme il est probable que la méthode XP ne fasse pas partie intégrante du système qualité (voire que le cycle itératif ne soit pas standard), il faudra décrire la méthode de travail de façon bien plus détaillée que pour un développement utilisant le standard de l'organisation.

Le plan de développement, tout en faisant référence à un ouvrage tel celui-ci, doit comporter une synthèse de la méthode XP, avec certaines précisions factuelles. En particulier, il faudra décrire le cycle de vie XP (en précisant la durée retenue pour les itérations), résumer chacune des pratiques (en détaillant certains choix spécifiques, par exemple l'outil de gestion de configuration pour l'intégration continue) et les rôles décrits au chapitre 2 (en donnant la liste nominative des membres de l'équipe et de leur rôle).

En revanche, alors qu'un plan de développement classique comporterait vraisemblablement un planning, le plan du projet XP devra expliciter que deux types de plans (le plan de livraisons et les plans d'itération) seront produits et mis à jour régulièrement. Le plus simple consistera à insérer chaque version mise à jour du plan de livraison, ainsi que le plan de chaque itération, dans un classeur spécifique.

Les indicateurs

Par rapport à la plupart des méthodes, XP peut se vanter de disposer d'un certain nombre d'indicateurs, certains assez originaux comme la vélocité, tous assez concrets et «parlants». Ces indicateurs, pour peu que leur signification et leur interprétation soient expliquées, sont appréciés autant du management que des responsables qualité.

Plutôt que de n'utiliser les indicateurs que pour le pilotage interne du projet, il est souhaitable de les diffuser aux responsables intéressés. Par exemple, lors de chaque itération, un tableau des principaux indicateurs et des graphes montrant leur évolution dans le temps pourront être joints au plan d'itération et au plan de livraison mis à jour. Seront inclus en particulier : la vélocité, le nombre total de scénarios client, le nombre et la proportion de scénarios client terminés et «recettés», le nombre total de tests de recette, le nombre de tests de recette qui passent et le nombre de régressions. Si vous disposez d'un outil adapté, il est utile d'indiquer le taux de couverture du code par l'exécutable des tests unitaires…

La documentation

Si les esprits semblent prêts à admettre les cycles de vie itératifs, la culture de l'ISO9001 en matière de documentation est telle que le point de vue de l'Extreme Programming, selon lequel la documentation n'est pas indispensable, du moins pour les membres de l'équipe de développement, est très difficile à faire accepter. Il est donc bon de rappeler que la fourniture de documentation, serait-elle technique ou relative à la gestion de projet, est tout à fait envisa-

geable dans le cadre de la méthode XP. Simplement, une telle documentation est considérée comme un scénario client supplémentaire, qui doit faire l'objet d'une description par le client et d'une planification collective.

On dispose cependant de méthodes qui permettent de fournir à peu de frais une documentation assez détaillée :

- On peut compiler les scénarios client pour produire une «spécification détaillée». Il faut pour cela les rédiger sous forme électronique, de façon, d'une part, à pouvoir les imprimer individuellement pour faire des «cartons» qui seront utilisés lors des séances de planification et que les développeurs pourront garder sous les yeux lors du développement, et, d'autre part, afin de les réunir au sein d'un document unique qui sera complété d'une page de garde, d'une table des matières, etc.
- On peut utiliser l'un des nombreux outils générant des documentations (HTML ou papier souvent en divers formats) à partir de l'analyse de code et de l'extraction des commentaires. Afin de respecter au maximum la règle XP consistant à éviter les commentaires lorsque la même chose peut être exprimée par le code, il est impératif de choisir un outil qui analyse réellement le code, c'est-à-dire capable de produire une documentation à partir d'un code sans commentaires. Le résultat peut être légitimement considéré comme un document de conception détaillée. On peut même se permettre de mettre plus de commentaires que pour une génération automatique, sachant au moins que l'esprit, sinon la lettre, de l'Extreme Programming est respecté, puisque la documentation réside avec le code.
- Selon la façon dont les tests de recette sont écrits, la même technique peut être utilisée pour générer automatiquement un plan et un dossier de tests. Il suffit pour cela de coder les tests de recette dans l'un des nombreux langages de script (Python, par exemple) qui possèdent la faculté de générer automatiquement une documentation…

Quant au reste (par exemple un document de conception générale ou d'architecture, il faudra bien le réaliser «à la main», en s'efforçant de rester concis (dix pages maximum) et d'éviter les documents qui ne servent vraiment à personne…

La traçabilité

Le terme de traçabilité prend deux sens dans le contexte de l'assurance qualité :

1. Le fait de pouvoir retrouver la raison d'être de toute activité ou de tout produit, par exemple d'associer chaque test de recette à une spécification, chaque tâche à un élément de la conception, etc.
2. Le fait de conserver un historique daté et circonstancié de toutes les décisions, tous les changements et tous les événements ponctuant la vie d'un projet.

La méthode XP, compte tenu de son souci de satisfaction donnée au client, mais aussi de simplicité et d'efficacité, permet une très forte traçabilité au premier sens du terme. Le fait d'écrire les tests avant même de commencer le code en est un exemple : on sait à tout moment

pourquoi on écrit tel code. Le fait de ne pas faire beaucoup de conception et d'architecture en amont est une autre garantie de traçabilité : le code est écrit pour satisfaire directement une exigence du client, plutôt que pour se conformer à une architecture complexe qui n'est censée satisfaire une exigence du client qu'indirectement.

En revanche, la conservation d'un historique n'est clairement pas une priorité XP. Une solution peu contraignante peut consister à utiliser un cahier, sorte de livre de bord du projet, dans lequel les plus importantes décisions et les principaux événements sont notés. Ce cahier peut avoir une forme matérielle (on peut ainsi aisément prendre des notes lors des réunions) ou celle de fichiers textes ASCII, du type «notes techniques», gérés conjointement avec le code source. L'avantage de cette seconde solution est double : d'une part, ces notes restent proches du code dont elles parlent et, d'autre part, les outils de gestion des versions permettent sans effort supplémentaire de conserver un historique complet et daté de toutes les modifications portées à ces documents.

Témoignage d'un directeur qualité

Nous avons expérimenté la méthode XP pour le développement d'un logiciel destiné à une application de supervision dans le domaine des transports[a]. Le témoignage que je formule ici est établi du point de vue du qualiticien.

Riche d'enseignements, cette expérience a démontré l'efficacité du processus basé sur la «mise en ligne» d'une équipe intégrée, dont les membres travaillent en binôme. Ce processus, de toute évidence, et à condition, bien entendu, que les ressources affectées soient maintenues durant toute l'opération, sécurise le développement dans la mesure où, à tout instant, chaque équipier connaît l'état réel d'avancement, et le stade auquel est rendu l'objet en cours de développement.

De même, le caractère incrémental de la méthode semble efficace en termes de temps de développement, et assure une bonne maîtrise des régressions.

Du point de vue du management de la qualité, il est clair que la méthode XP induit un risque très sensible en termes de traçabilité, risque qui constitue, en quelque sorte, le revers de la médaille du processus incrémental.

L'application de la méthode doit donc être rigoureusement encadrée par la mise en œuvre d'indicateurs qualité pertinents, et de traces formelles qui garantissent, notamment, l'existence d'un référentiel de développement utilisable en cas de reprise ultérieure, même si l'équipe initiale a, alors, été dissoute.

Moyennant ces mesures conservatoires, parfaitement maîtrisables, la méthode XP s'avère efficace, et motivante pour les équipes qui l'appliquent, pour les logiciels de taille moyenne, la méthode classique du «cycle en V» s'avérant, semble-t-il, plus sûre pour les gros logiciels, ou pour ceux qui portent des fonctionnalités critiques, par exemple en termes de sécurité.

a. Ce projet est décrit en détail au chapitre 12.

La traçabilité et les fiches cartonnées

L'utilisation de fiches cartonnées dans le cadre de la méthode XP (pour les scénarios client et les tâches, en particulier) n'est pas inconcevable, à condition de respecter certaines règles relatives aux contraintes de traçabilité :

- Il faut attribuer un numéro unique à chaque carton (préciser dans le plan de développement une procédure pour l'attribution des numéros – un simple compteur sur un tableau peut suffire).
- Il ne faut pas jeter les cartons terminés, ni même ceux qui sont destinés à être récrits, scindés ou fusionnés… Il convient de conserver tous les cartons, annotés (ou tamponnés) «terminé», «scindé en…», «fusionné avec…», etc.
- Il est également utile d'annoter certaines dates sur les cartons («créé le…», «terminé le…»).

Toutefois, de nombreuses équipes XP gèrent leurs «cartons» sous forme électronique, quitte à imprimer des «cartons» pour les utiliser lors des séances de planification (où la manipulation d'un objet physique prend tout son sens).

Attention aux dérives lors de la gestion électronique des cartons

Il faut veiller à préserver la simplicité du système utilisé : par exemple, de simples fichiers textes peuvent suffire, dont on peut gérer les versions dans le même outil que le code source… Se rappeler qu'une gestion électronique doit avant tout faciliter la communication et non la limiter.

La gestion électronique des scénarios client, en particulier, est indispensable dès lors que ces scénarios client sont utilisés pour produire une documentation de «spécifications».

XP et les autres méthodologies de développement logiciel

Le cycle en V

La méthode XP a souvent été présentée, y compris dans le présent ouvrage, comme une solution – voire une réaction – aux méfaits du cycle en V. Dans cette section, nous allons présenter, de façon synthétique, le cycle en V, afin de mieux comparer cette approche avec XP et avant d'étudier la possibilité de les faire cohabiter le mieux possible.

Aperçu de la méthode

Le cycle en V est une approche descendante (*top down*) consistant à décomposer un problème complexe en un grand nombre de problèmes plus simples que leur taille permet de maîtriser.

Ensuite, chaque problème est résolu (ou plutôt la solution est réalisée), et il reste à recoller les solutions séparées pour remonter petit à petit vers une solution unique au problème initial.

La descente comporte un certain nombre de paliers (variables selon la complexité du problème), correspondant à un ordre de grandeur dans la décomposition, et ces paliers doivent être respectés aussi dans la remontée.

Afin d'être sûr qu'à tout moment on continue de travailler à la résolution du problème initial, le cycle en V exige qu'à l'étape initiale de décomposition, on s'adresse à la totalité du problème, puis qu'à chaque palier suivant, on s'adresse à la totalité des sous-problèmes avant de passer à l'étape suivante.

Enfin, à chaque palier dans la descente correspond un document qui décrit la décomposition obtenue, et à chaque palier de la remontée correspondent des tests montrant que certaines solutions recollées fonctionnent ensemble et constituent une solution pour l'un des problèmes de niveau supérieur…

La meilleure façon de comprendre le cycle en V (et la raison de sa dénomination, «en V») reste encore de recourir à un schéma. La figure 10-3 représente un cycle en V appliqué au développement logiciel. En effet, le cycle en V – c'est sans doute là un de ses atouts et ce qui explique qu'il est encore pratiqué malgré les problèmes qu'il pose dans le domaine du logiciel – peut convenir pour d'autres types de développements, notamment matériels ou systèmes. En particulier, nous allons étudier ci-après la cohabitation entre la méthode XP pour le logiciel et le cycle en V pour le système en général pour des développements de systèmes comportant un ou plusieurs sous-systèmes logiciels.

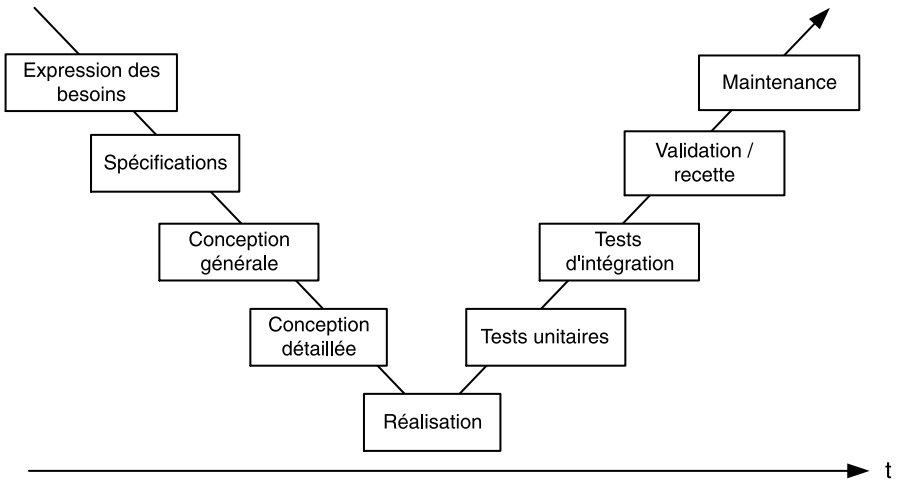

Figure 10-3. Le cycle en V logiciel

Une variante du cycle en V, plus souvent préconisée que réellement pratiquée, s'appelle le « V consolidé ». Elle consiste à définir, à chaque palier de la descente, les tests qui devront être pratiqués au palier correspondant dans la remontée (voir figure 10-4).

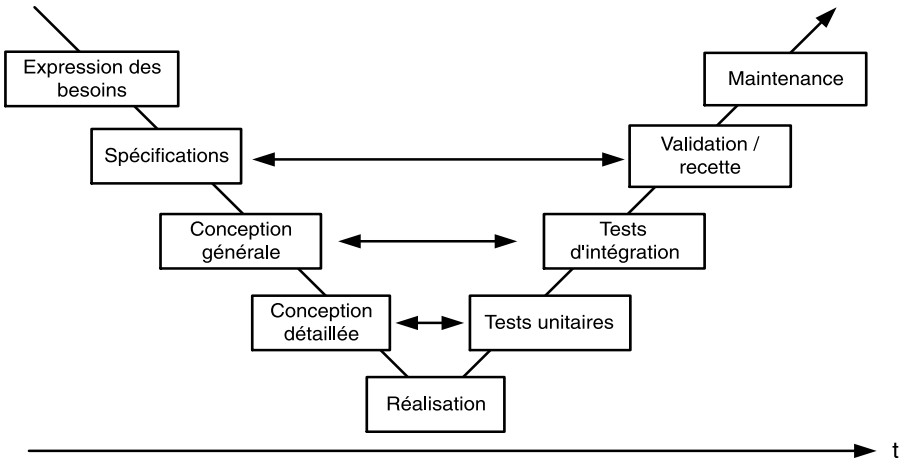

Figure 10-4. Le cycle en V consolidé

Convergences et divergences entre XP et le cycle en V

Le principe du V consolidé (considérer la définition de tests comme un complément d'une spécification, ou définir des tests comme partie intégrante d'une conception et avant de passer à la réalisation) est un exemple de technique que l'Extreme Programming n'a pas réellement inventé mais a amplifié, et qui, par conséquent, est réellement pratiquée. C'est en tout cas un sujet de convergence entre XP et cette variante du cycle en V.

Hélas, il est difficile de trouver d'autres points communs, tant ces méthodes divergent à la fois dans leurs pratiques concrètes et dans leurs points de vue fondamentaux (voir tableau 10-1).

Tableau 10-1. Divergences de fond entre le cycle en V et l'Extreme Programming

Cycle en V	Extreme Programming
Le problème doit être traité intégralement, de front.	**Le problème peut être décomposé en tranches qui sont traitées les unes après les autres.**
Le système doit être spécifié en totalité avant de passer à la conception.	**Il faut spécifier une partie du système, la développer entièrement et utiliser le résultat comme feedback pour la suite des spécifications.**
Il faut effectuer la conception de la totalité du système avant de passer à la réalisation.	**Il faut effectuer la conception d'une tranche à la fois, en faisant évoluer la conception obtenue lors de la tranche précédente.**

Cycle en V	Extreme Programming
Le résultat de la conception d'un logiciel est un document permettant de commencer à coder. Le codage correspond à la «fabrication» d'un logiciel.	**Le résultat de la conception d'un logiciel est son code. La «fabrication» du logiciel est réalisée par un outil (le compilateur) à partir du code.**
Chaque phase comporte une activité spécifique (spécification, conception, codage, tests) qui peut être confiée à des individus ou des équipes distinctes, qui peuvent travailler à l'aide des documents produits lors de la phase précédente.	**Les développeurs doivent être polyvalents et responsables de l'ensemble des activités. Une communication efficace n'est pas possible par la seule voie documentaire.**

Inscrire XP dans un cycle en V

Dans un cycle en V système (c'est-à-dire consistant à étudier et réaliser un système complexe qui comporte plusieurs sous-systèmes, dont certains sont matériels), le développement d'un lot logiciel est considéré comme une étape de réalisation (la partie la plus basse du V). Il est donc possible, et il est souvent admis, d'effectuer le développement logiciel lui-même selon un autre cycle de vie que le V.

Par exemple, la notion de cycle en W a été utilisée pour introduire une forme d'itération dans un contexte de cycle en V. Comme l'indique la figure 10-5, il s'agit, à l'intérieur d'une phase qui peut être vue de l'extérieur comme étant un seul V, de faire plusieurs V à la suite, en raffinant successivement les produits, qu'ils soient documentaires ou logiciels. Il est aussi tout à fait possible (voir le chapitre 12 pour un exemple) d'utiliser la méthode XP dans les phases de bas niveau. (Il faut toutefois rappeler que, dans la méthode XP, une itération ne ressemble en rien à un petit cycle en V – même si l'analogie est vraiment contestable, cela reviendrait à faire un cycle en V toutes les quelques minutes…).

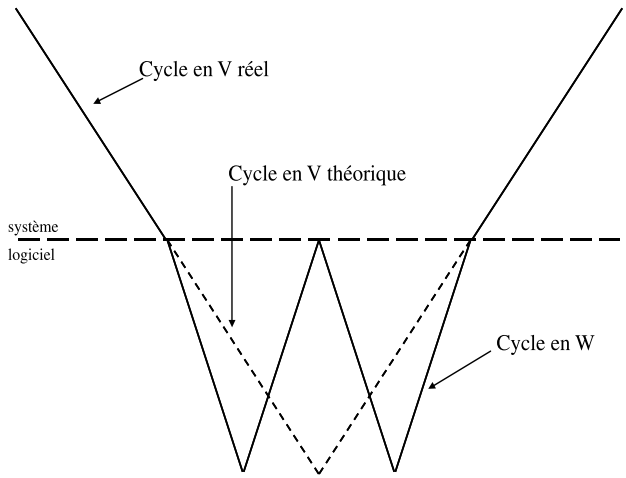

Figure 10-5. Le cycle en W

Fondamentalement, cette technique permet de satisfaire les attentes générales de l'équipe système : la phase de réalisation logicielle produit un logiciel à partir d'un document de spécifications. Que le logiciel ait été lui-même produit en V, en W, en Y ou en XP n'y change pas grand-chose.

Toutefois, il ne faut pas négliger un détail : si une production documentaire est prévue de la part du lot logiciel, l'équipe système s'attendra à recevoir assez tôt certains de ces documents (ceux correspondant à la partie descendante d'un cycle en V, soit les spécifications et la conception, voire les cahiers de tests si le V est consolidé). Or, dans le cas d'un cycle en W, les documents pourraient être complets lors du premier V, mais imparfaits et inexacts en attendant les raffinements des V suivants. Dans le cas d'XP, les documents seraient à la fois incomplets et inexacts jusqu'à un stade assez tardif, puisque l'ensemble des scénarios clients ne seraient pas traités, et que les scénarios eux-mêmes et surtout la conception seraient certains d'évoluer.

La solution consiste d'une part à annoncer des délais très tardifs pour la production documentaire (entre 50 % et 75 % du délai total estimé), d'autre part à jouer sur la seule souplesse permise au sein de la rigidité du cycle en V : réviser les documents et les rediffuser sous un nouvel indice… En espérant que ce qui a pu être écrit dans les premières versions n'ait pas engagé le reste du projet dans une mauvaise voie, ou le client dans de fausses espérances…

Rational Unified Process

Le Rational Unified Process ou RUP se positionne comme une solution moderne, itérative, plus adaptée aux développements orientés objet et à l'*e-business* que les méthodes très formalistes que nous venons de décrire. Sa philosophie diffère cependant assez nettement de celle d'XP et des autres méthodes de la mouvance agile, sur lesquelles nous reviendrons plus loin.

Aperçu de la méthode

Le RUP se veut la synthèse de six *best practices*, c'est-à-dire les principales pratiques auxquelles un grand nombre de projets ou d'organisations auraient attribué leurs réussites :

1. **Développement itératif**. Partant du constat qu'un cycle strictement séquentiel (V ou *waterfall*) est quasi impossible à respecter, le développement itératif est préconisé, permettant les raffinements successifs des spécifications et de la conception, et atténuant les risques grâces à des livraisons fréquentes.

2. **Traitement et formalisation des exigences**. Une méthode de gestion des exigences est définie, consistant surtout à les décrire selon un certain formalisme (les cas d'utilisation et scénarios UML) ainsi qu'à les documenter, les justifier et suivre leurs évolutions avec le temps, ainsi que leurs dépendances.

Un peu d'histoire

Au début des années 1990, trois principales méthodes se partageaient la majorité des développements orientés objet : la méthode OMT conçue par Jim Rumbaugh, la méthode OOAD de Grady Booch et la méthode OOSE d'Ivar Jacobson. Chacune, outre une notation (un formalisme) permettant de décrire des modèles objet, préconisait une méthode de travail, une marche à suivre, un processus de développement, qui pouvait être suivi pour spécifier, concevoir, développer et tester un logiciel orienté objet.

Lorsque Rumbaugh, et plus tard Jacobson, rejoignirent Grady Booch au sein de la société Rational, naquit l'espoir (et l'annonce) d'une méthode unifiée. Très vite, pourtant, les trois gourous eurent l'honnêteté intellectuelle de reconnaître qu'aucune méthode, aucun processus, ne pouvait convenir à tous les projets, à toutes les organisations et à tous les goûts. Ils se consacrèrent donc à l'objectif plus limité mais non moins important de définir une notation unique pour la modélisation de logiciels orientés objet : l'Unified Modeling Language (UML).

UML est désormais normalisé et développé au sein du consortium de l'OMG (Object Management Group), et la société Rational s'est contentée de développer à sa propre initiative (sans susciter l'adhésion de la communauté en général) une méthode unifiée. Le Rational Unified Process (RUP) est donc avant tout un produit commercial ; toutefois, profitant d'une confusion avec UML et surtout de la disparition des trois principales méthodes originelles de Booch, Rumbaugh et Jacobson, il s'agit aujourd'hui d'une des méthodes les plus répandues.

3. **Architecture à base de composants**. Le RUP met en avant une première technique de conception : l'architecture à base de composants. Elle consiste à définir une architecture robuste, flexible, compréhensible et réutilisable, avant de se consacrer pleinement au développement. Cette architecture doit être conçue sous forme de composants, dans un cadre standard (tel CORBA ou COM) ou spécifique.

4. **Modélisation visuelle**. La modélisation visuelle (à base d'UML) consiste à étudier et exprimer la conception en faisant abstraction du code grâce à la notation UML, en assemblant graphiquement des briques.

5. **Vérification de la qualité logicielle**. La cinquième pratique fondamentale a trait à la vérification de la qualité logicielle, qui doit faire partie intégrante de toutes les phases et activités du développement .

6. **Gestion des changements**. Derrière ce terme de gestion des changements sont rangées des procédures destinées à formaliser, justifier et tracer les changements que subissent les produits (documentaires ou logiciels), ainsi que les procédures de gestion des versions et des configurations dans le contexte d'un développement en équipe.

Outre ces six pratiques centrales, le RUP est défini par trois caractéristiques principales :

- son découpage temporel en phases et itérations,
- son support entièrement informatique,

- son aspect générique et paramétrable.

Le découpage temporel d'un projet RUP comporte quatre phases :

- L'*inception*, pendant laquelle les grandes lignes du projet sont définies : principales fonctionnalités, budget et délai, plan d'exécution. À la fin de cette phase, le *business case* (la justification en termes de retours sur investissements) du projet est établi, et il est encore temps d'annuler son lancement.
- L'*élaboration*, pendant laquelle l'ensemble des fonctionnalités et propriétés sont définies, l'analyse est réalisée et le système modélisé, l'architecture stabilisée et la planification revue et détaillée.
- La *construction*, pendant laquelle les fonctions et composants, dont certains ont pu être prototypés lors de la phase précédente, sont terminés, testés et rendus prêts à l'utilisation.
- La *transition*, pendant laquelle le produit est déployé, ses défauts de jeunesse corrigés, et lors de laquelle, de manière générale, le produit passe d'un état où il est connu et géré par l'équipe de développement au stade où il est entièrement entre les mains de ses utilisateurs.

Chacune de ces quatre phases peut être découpée en itérations (entre deux et quatre), chaque itération correspondant à un mini-développement qui aboutit à une livraison interne ou externe d'une version du produit.

Si les différentes activités (modélisation du domaine, analyse et conception, codage, test) peuvent être pratiquées à chaque phase et se chevauchent partiellement, chacune reste concentrée dans une seule phase.

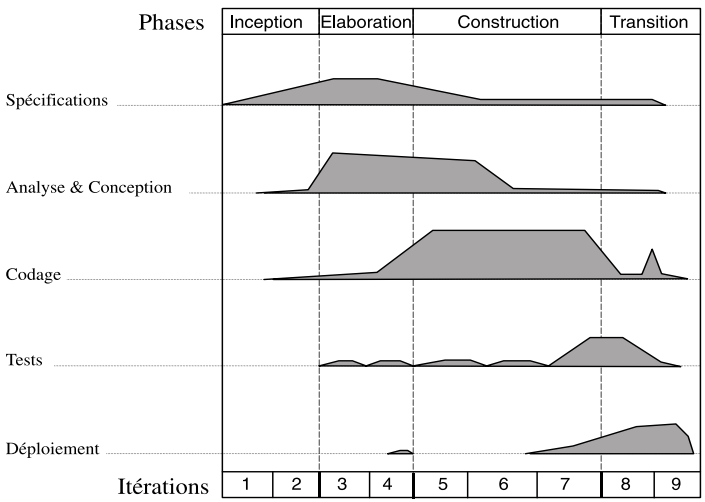

Figure 10-6. Phases et activités d'un projet RUP

Une indéniable innovation du RUP tient à ce qu'un support entièrement informatique est proposé : le produit lui-même, commercialisé par Rational, est en fait la description détaillée de la méthode sous forme électronique, consultable au moyen d'un Web intranet. L'aspect purement documentaire est largement complété de nombreux modèles pour appliquer le RUP au moyen de la panoplie d'outils commercialisés par Rational, correspondant aux différentes pratiques : par exemple, le produit Rose pour la modélisation visuelle et Clearcase pour la gestion des changements.

Enfin, le RUP est présenté comme étant un processus standard générique, modulable et qui peut être spécialisé à souhait. On l'a vu, le découpage des phases en itération est optionnel, et la durée des phases et des itérations peut varier énormément (de 1 mois à 1 an pour la phase de construction, par exemple). Le RUP définit plus de 100 choses à produire au cours d'un projet (comprenant plans, documents et formulaires, outre le logiciel lui-même), mais certaines de ces productions sont considérées comme optionnelles. Rational propose ses consultants pour adapter le RUP à une organisation donnée.

Convergences et divergences entre RUP et XP

RUP et XP présentent un certain nombre de points communs ou au moins compatibles. Robert C. Martin, grand expert de l'objet et désormais adepte de l'Extreme Programming, a utilisé l'aspect paramétrable de RUP pour présenter XP comme une possible spécialisation très minimaliste du RUP. Cette vision a le mérite de la diplomatie, et surtout de montrer comment introduire et justifier XP au sein d'une organisation où RUP serait le standard. Mais XP ne peut se résumer à une version minimaliste de RUP, et il existe entre les deux méthodes un grand nombre de divergences, autant en raison de leur fonctionnement concret que des points de vue adoptés.

Il est tout d'abord intéressant de noter que Rational a choisi d'articuler sa méthode autour de *best practices*, alors que l'Extreme Programming met en avant des valeurs et des principes. Les pratiques d'XP sont présentées comme la meilleure façon qui soit connue à l'heure actuelle de respecter au mieux l'ensemble des valeurs et principes ; à condition de bien prendre en compte ces derniers, les pratiques peuvent être modifiées, supprimées ou complétées par de nouvelles. Il n'est pas certain que les six *best practices* mises en avant par RUP soient considérées comme les meilleures indéfiniment, et RUP ne fournit pas de critères selon lesquels on peut juger de nouvelles pratiques.

Ensuite, la méthode XP, tout comme les autres méthodes agiles décrites à la fin de ce chapitre, attache plus d'importance aux personnes qu'aux outils. Les outils sont même souvent sacrifiés à l'autel de la simplicité. À l'inverse, le RUP est en soi un outil informatique (une base de connaissances consultable via internet). En outre, deux des six *best practices* du RUP sont purement technologiques (l'architecture à base de composants et la modélisation visuelle), et mis à part le développement itératif, chaque pratique préconisée par le RUP nécessite, pour être vraiment efficace, un ou plusieurs outils informatiques.

Enfin, si les deux méthodes peuvent être considérées comme itératives, ce terme ne prend pas le même sens dans les deux cas de figure. RUP maintient la notion de phases linéaires, où une activité domine à tout instant, successivement, la spécification, la conception, le codage et le test, et le déploiement. L'aspect itératif du RUP ne fait qu'atténuer, par rapport à un cycle en V, les frontières entre ces phases.

XP va beaucoup plus loin, en exigeant que, dès le début et à chaque itération, le logiciel soit fourni à l'utilisateur, incomplet, mais en état de fonctionnement. Cette approche extrême est une véritable gestion des risques et un formidable moyen d'obtenir un véritable feedback de la part du client. Avec le RUP, un logiciel exécutable est préconisé à chaque itération, mais il peut s'agir dans les premiers temps de simples maquettes, puis de prototypes, en général destinés à des livraisons internes uniquement. Le produit n'est prêt à être livré au client qu'à la fin de la phase de construction.

Cette distinction s'explique par une grande différence culturelle entre XP et RUP dans le domaine de la conception :

- Les pratiquants d'XP considèrent que le code, le langage de programmation, sont la seule expression nécessaire et suffisante de la conception d'un logiciel. En génie logiciel, l'activité de conception consiste à produire du code. En outre, ils considèrent qu'il est possible et souhaitable d'élaborer une architecture progressivement, et qu'à cette fin, il est possible de maintenir indéfiniment le code dans un état permettant des changements d'architecture.
- Les concepteurs du RUP considèrent que l'activité conceptuelle en génie logiciel consiste à produire des modèles, et qu'il est possible et utile de faire abstraction du code. Ils préfèrent chercher à se doter d'outils qui permettent de générer du code à partir des modèles. Enfin, ils considèrent qu'il est possible et souhaitable de définir et figer dès le début une architecture qui prend en compte l'ensemble des besoins fonctionnels et non fonctionnels du logiciel. À leurs yeux, une fois qu'un logiciel a atteint une certaine taille, il est trop tard pour faire évoluer son architecture en profondeur.

Si la première divergence (le code opposé aux modèles) reste assez métaphysique, de nombreux projets XP ont cependant déjà prouvé qu'il est possible (du moins en respectant les pratiques d'XP) de faire évoluer en profondeur l'architecture d'un logiciel pour prendre en compte des besoins nouveaux.

En conclusion, nous pensons que les méthodes sont compatibles au sens où il est possible de pratiquer XP en prétendant pratiquer RUP. En revanche, le fait de pratiquer seulement RUP ne permet pas de tirer tous les bénéfices de l'Extreme Programming, et la culture RUP est suffisamment différente pour qu'il y ait de fortes chances d'aboutir à quelque chose de très éloigné d'XP.

Les (autres) méthodes agiles

L'Extreme Programming s'inscrit dans un mouvement plus large dont elle est la figure de proue : l'Alliance agile, qui regroupe notamment les méthodes Crystal, ASD (*Adaptive*

Software Development), Scrum, FDD (*Feature-Driven Development*), DSDM (*Dynamic Systems Development Method*) et Pragmatic Programming.

Conscients des idéaux et valeurs que ces méthodes partagent mais mécontents de l'expression «méthodes légères» qui fut longtemps leur seul point commun explicite, les auteurs, responsables et autres inspirateurs de ces diverses méthodes se sont réunis en février 2001 pour tenter de décrire avec précision leur terrain d'entente. Ils y sont parvenus au-delà de leurs propres espérances, puisqu'ils ont rédigé un texte commun, le manifeste agile, et par là même ont identifié le concept d'agilité qui s'applique désormais à toutes les méthodes qui correspondent au manifeste.

L'Alliance agile définit l'agilité en développement logiciel ?

L'Alliance agile prône de nouvelles façons de développer des logiciels et préconise de renverser certaines priorités. L'importance est ainsi donnée :

- aux **individus et aux interactions** plus qu'aux processus et aux outils ;
- à des **logiciels immédiatement disponibles** plus qu'à une documentation exhaustive ;
- à la **collaboration avec le client** plus qu'à la négociation contractuelle ;
- à la **réactivité face au changement** plus qu'au respect d'un plan.

Non que les seconds termes soient dénués de valeur, mais préférence est donnée aux premiers : individus et interactions, fonctionnement immédiat des logiciels, collaboration et réactivité.

Les méthodes agiles s'appuient sur ces quatre valeurs pour décliner un certain nombre de principes, à l'instar de l'Extreme Programming. Au-delà de ces valeurs communes, chacune se distingue par ses pratiques spécifiques. Le cycle de vie itératif est une constante, permettant de produire régulièrement un logiciel opérationnel.

Crystal

Il s'agit en fait d'une famille de méthodes, car son auteur, Alistair Cockburn, estime qu'il faut prendre en compte différentes typologies de projet. Il a donc défini une matrice dans laquelle, en fonction de la taille de l'équipe et de la criticité du logiciel (si les conséquences d'une erreur sont négligeables, coûteuses, ou critiques pour l'existence du logiciel), on trouve une variante de Crystal dédiée à cette catégorie de projet.

L'autre particularité de Crystal est de mettre l'accent sur l'amélioration continue du processus, préconisant qu'à chaque itération l'équipe discute de la façon dont s'est déroulée cette itération, en identifiant les points forts et ceux qu'il faut améliorer dans la méthode de travail.

Si cette pratique existe sur de nombreux projets XP, elle n'est pourtant pas aussi centrale qu'elle peut l'être avec Crystal. En outre, XP instaure une discipline forte, et il est préférable

de pratiquer d'abord la méthode sans adaptations pendant assez longtemps pour en comprendre les dépendances entre les pratiques.

Pour Alistair Cockburn, les gens ont du mal à se discipliner et Crystal vise donc «le minimum acceptable de discipline» (encore une fois, ce minimum varie en fonction de la taille de l'équipe et de la criticité du logiciel). Il considère que Crystal est moins productif qu'XP, mais à la portée d'un plus grand nombre.

Contrairement à la plupart des méthodologistes, Alistair Cockburn ne s'appuie pas sur sa seule expérience personnelle. Lorsqu'IBM lui a demandé de réfléchir à la méthodologie logicielle au début des années 1990, Cockburn a observé pendant plusieurs années de nombreux projets et en a interviewé les acteurs.

Alistair Cockburn a annoncé début 2001 que Jim Highsmith et lui-même allaient désormais fusionner leurs méthodologies, Crystal et ASD (voir ci-après), mais le résultat de ce travail n'est pas connu au moment où nous rédigeons ces lignes.

Adaptive Software Development (ASD)

La collaboration entre Alistair Cockburn et Jim Highsmith est fort prometteuse, car si Cockburn se base sur l'observation de projets qui réussissent et propose par conséquent des idées très concrètes et pragmatiques, Highsmith, quant à lui, a contribué à fournir une base théorique aux méthodes agiles en appliquant les idées issues de la théorie du chaos au développement logiciel. En effet, après des années passées à développer, étudier et enseigner les méthodes prédictives, Highsmith a fini par conclure qu'elles étaient profondément erronées, considérant que le développement logiciel nécessite un environnement adaptatif, considérant aussi que l'on peut tirer parti de l'imprévu pour atteindre de meilleures solutions, qui n'auraient pas été envisagées *a priori*…

ASD comporte trois «phases» (même si elles sont non linéaires et superposées) : spéculation, collaboration et apprentissage. Le principal atout de la méthode ASD est en effet de favoriser un environnement propice à la collaboration et à l'apprentissage, lequel doit concerner tous les acteurs, les clients aussi bien que les développeurs et le management.

Scrum

La littérature de la méthode Scrum décrit surtout les pratiques de planification. Comme en XP, celle-ci est itérative, mais les itérations, appelées des «sprints», durent 1 mois. Avant chaque sprint, les fonctionnalités qu'il faut réaliser sont définies, mais par la suite l'équipe travaille de façon isolée par rapport au client, l'objectif étant de figer les fonctionnalités pour la durée du sprint. Mais l'isolement n'est pas total, et une réunion quotidienne de 15 minutes appelée *scrum* (ou mêlée de rugby) permet aux développeurs de passer en revue leurs activités du lendemain et de faire remonter vers le management les éventuels problèmes qu'ils ont rencontrés.

Il est apparemment possible d'utiliser les pratiques de programmation d'XP avec une planification inspirée de Scrum (ce à quoi l'un des inspirateurs de Scrum, Mike Beedle, s'emploie

sous la dénomination XBreed), mais à vrai dire cette planification ne diffère pas sensiblement de celle d'XP, et les *scrums* eux-mêmes ressemblent à s'y méprendre aux réunions debout (*standup meetings*) pratiquées par les équipes XP.

Feature Driven Development (FDD)

Cette méthode, développée par Jeff De Luca et Peter Coad, comprend cinq «processus» :

- développer un modèle global ;
- construire une liste de fonctionnalités *(feature)* ;
- planifier par fonctionnalité ;
- concevoir par fonctionnalité ;
- réaliser par fonctionnalité.

Les trois premiers processus sont effectués en début de projet, sur l'ensemble des fonctionnalités. Ensuite, les deux derniers sont répétés itérativement, en les appliquant à une sélection de fonctionnalités. Comme les autres méthodes agiles, les itérations sont courtes (deux semaines pour FDD) et visent à fournir une fonctionnalité limitée mais utilisable.

En revanche, l'organisation de l'équipe est sensiblement différente de celle préconisée par XP. Les développeurs sont de deux catégories : les architectes et les propriétaires de classes. Les architectes prennent en charge certaines fonctionnalités, alors que les propriétaires de classe sont… propriétaires d'une ou plusieurs classes ! Chaque architecte est entièrement responsable de la conception d'une fonctionnalité et de l'identification des classes qui y participent. Les propriétaires des classes correspondantes font l'essentiel de la programmation, sous la coordination de l'architecte concerné, qui joue aussi le rôle de mentor.

Dynamic System Development Method (DSDM)

DSDM est une méthode développée par un consortium d'entreprises, britanniques à l'origine, mais désormais internationales, souhaitant bénéficier du développement itératif et RAD (Rapid Application Development). Cette méthode est donc plus fortement supportée que les autres méthodes agiles, puisque le consortium finance la réalisation de guides et manuels, et organise des formations et des certifications. D'un autre côté, il faut devenir membre du consortium pour avoir accès à ces informations… il nous est donc difficile de donner beaucoup de détails sur cette méthode.

Fondamentalement, DSDM est itérative, privilégie une forte interaction avec les utilisateurs, responsabilise les développeurs et met l'accent sur les tests tout au long du développement, une forte qualité et l'adaptation aux changements des besoins. Sous ses apparences de méthodologie «sérieuse» et officielle, elle mérite donc bien son appartenance au mouvement agile.

Pragmatic Programming

Il ne s'agit pas vraiment d'une méthode, mais Andy Hunt et Dave Thomas, auteurs du livre *The Pragmatic Programmer*, font partie de l'alliance Agile et ont contribué à la rédaction de son manifeste. Ils mettent en avant un certain nombre de best practices, en fait, des conseils très concrets, destinés aux programmeurs afin qu'ils puissent améliorer leurs compétences et prendre de bonnes habitudes de travail. Un certain nombre de leurs recommandations sont presque d'ordre méthodologique, mais pour l'essentiel ce sont des conseils qui sont compatibles et complémentaires avec une méthode comme XP, qu'ils soutiennent sans lui attribuer le statut de solution miracle.

Troisième partie

Études de cas

Dans *La Structure des révolutions scientifiques*, le philosophe T. S. Kuhn établit à partir de l'histoire des sciences une conception du progrès des connaissances scientifiques qui tranche nettement avec la vision cartésienne dominante. L'histoire des sciences ne procède pas d'une accumulation patiente et progressive de faits et de théories toujours plus précises ; elle met en jeu un conflit périodique entre des «paradigmes» anciens et nouveaux, décrivant le monde de façon radicalement différente et mutuellement incompatible. Chaque progrès dans les sciences s'identifie à l'adoption d'un nouveau paradigme, dont il résulte que les problèmes sont posés d'une nouvelle manière ; des questions anciennes disparaissent ou deviennent vides de sens, de nouvelles apparaissent. L'ancien paradigme s'efface non parce qu'il est, de façon «démontrable», faux mais simplement parce qu'il se révèle moins efficace que le nouveau pour résoudre les questions auxquelles sont confrontés les chercheurs.

Le parallèle avec XP est tentant. Il nécessite une certaine prudence, puisque les arguments de Kuhn ne s'adressent qu'aux seules connaissances scientifiques : cela présupposerait une «science du logiciel» dont l'existence est sujette à controverse. Mais les similitudes imposent leur évidence :

- XP ne cherche pas à résoudre certains problèmes «classiques» du développement et de la gestion de projet, tels que l'estimation précise des coûts ou la mise au point d'une conception d'ensemble qui pourrait être formellement vérifiée comme «correcte» ; ces problèmes sont considérés comme sans valeur théorique réelle.
- Inversement, XP créée des problèmes nouveaux et des concepts sans équivalents dans l'ancien paradigme : comment les tests unitaires écrits avant le code donnent naissance à une architecture émergente ; comment les quatre valeurs fondamentales interagissent de façon systémique pour optimiser la productivité d'une équipe…
- XP est défini de façon communautaire ; ce qui «est extrême» et ce qui ne l'est pas est soumis au consensus de la communauté, non imposé *ex cathedra* ; la communauté se définit elle-même par son adhésion à un système de valeurs et de règles qu'elle articule et fait évoluer, parfois de façon implicite.

Kuhn souligne qu'un paradigme nouveau ne s'impose pas nécessairement par des arguments logiques, mais en grande partie parce que ceux qui l'ont formulé et ceux qui l'ont adopté par intime conviction sont capables de décrire de façon convaincante *ce que pourrait être* la pratique d'une discipline donnée dans le contexte de ce paradigme.

Au moment où ces lignes sont rédigées, deux documentaires filmés sur XP sont en préparation. À notre connaissance, c'est la première fois qu'un tel reportage télévisuel est réalisé dans le domaine de la méthodologie logicielle. Les études de cas présentées dans cette partie poursuivent un objectif similaire : non pas la dissection d'un prétendu «succès» lors de l'application d'XP, qui ne saurait avoir valeur de preuve, mais une mise en situation des idées qui ont été abordées tout au long de ce livre, de façon à suggérer *ce que peut être* la pratique de cette démarche.

Une première étude de cas nous fait vivre le quotidien d'une équipe chargé d'un projet Web. Les événements qui y sont relatés sont inspirés de faits réels, mais recomposés d'après une «mosaïque» de plusieurs projets échelonnés sur une période de deux ans environ. La seconde étude de cas retrace l'historique d'un projet industriel et met davantage en lumière la pratique d'XP dans des conditions qui nécessitent certaines adaptations.

11

Un projet Web en XP

Les batailles qui mettent en concurrence des paradigmes ne sont pas de celles qu'on peut arbitrer à l'aide de preuves.

– T. S. Kuhn

Un projet est une histoire

Chaque projet a une histoire, qui est une «tranche de vie» dans celle tout aussi imprévisible des personnes qui vont vivre autour de ce projet des moments de stress, de joie, de panique, de défaite ou de succès. Comme dans toutes les histoires, il s'y produit une rencontre : elle a lieu entre une vision, un désir – les grandes lignes de ce que sera, si le projet aboutit, le programme qu'il s'agit de créer – et une équipe qui fait sienne cette vision pour lui donner une forme concrète.

Cette idée et cette équipe sont les principaux personnages de notre histoire que vous allez découvrir «comme s'il s'agissait d'un roman»…

Les personnages

L'idée : gérer *via* le Web le processus de recrutement, de A à Z
Le prestataire : Advent, agence de communication interactive, filiale d'un groupe international
Les développeurs : Leslie, Sébastien, Benoît, Yves, Frank
Le client : David (Sonata Conseil)
Le Manager : Alex

Le temps et le lieu

L'action se situe à Paris et débute au dernier trimestre de l'année 2000. L'effondrement de la «bulle» spéculative des valeurs technologiques a jusqu'à présent épargné des entreprises telles qu'Advent, filiale spécialisée dans «le marketing et la communication portés par les médias interactifs» d'un groupe d'implantation mondiale issu de la «vieille économie». La direction d'Advent est cependant sous pression et priée de prouver l'intérêt d'un maintien de la structure si le contexte économique continue à se dégrader.

Ayant pris la direction d'Advent à la suite d'une récente restructuration, Alex a pour ambition de réussir une synthèse, imparfaitement menée jusqu'à présent, d'une part entre la communication et le marketing et d'autre part entre un savoir-faire et une infrastructure de pointe dans le domaine du Web. Pour soigner son image – un critère de réussite important dans ce secteur –, l'entreprise a investi dans des locaux de haut standing.

Deux grands *open space* ou bureaux paysagers permettent de regrouper d'un côté les créatifs, de l'autre les ingénieurs, tout en facilitant les échanges entre les deux pôles. Deux salles de réunion et un «coin détente» aménagé en fauteuils et tables basses les complètent. Les plans de travail sont larges, équipés de rangements sur roulettes qu'il est facile d'écarter pour circuler plus librement.

Les postes informatiques fournis aux collaborateurs sont tous des portables, à la seule exception de deux stations haut de gamme pour le travail graphique. Le niveau de bruit généré, bien moindre que celui de boîtiers PC classiques, contribue largement à une ambiance plutôt feutrée, propre à la réflexion ; seuls des concerts de klaxon ou un cortège de manifestants font, rarement, intrusion.

Ces aménagements n'ont pas été conçus spécifiquement pour XP ou la programmation en binôme ; ce sont avant tout des considérations d'image et de «productivité» (on peut emporter chez soi son portable pour rattraper du travail en retard…) qui les ont dictés. Ils joueront cependant un rôle important dans la mise en place de la méthode.

La théorie…

Tous les projets XP ne naissent pas égaux ; certains bénéficient de bonnes conditions structurelles pour telle ou telle pratique, et moins bonnes pour d'autres. Le fait d'observer l'environnement avant le démarrage du projet et de réfléchir à ce qu'il peut apporter en termes de dangers ou de bénéfices à la mise en place d'XP peut constituer un facteur de succès.

Naissance : avant le projet

David préside aux activités du cabinet de recrutement Sonata Conseil. Un partenariat concret le lie déjà à Advent, au-delà des relations amicales qu'il entretient avec Alex ; Sonata a mené le recrutement d'Yves et Benoît fut pendant quelque temps hébergé dans les locaux d'Advent

avant de s'installer dans ses nouveaux quartiers, trois stations de métro plus loin. Si David et Alex se connaissent et s'apprécient, «business is business», et l'un comme l'autre entendent bien rechercher un réel profit dans une éventuelle transaction – tout au plus conçoivent-ils leur objectif commun comme un marché gagnant-gagnant.

L'idée, embryon du projet

Selon David, «le recrutement des ingénieurs, mais aussi de la plupart des cadres, y compris les dirigeants, se fera majoritairement par le biais d'Internet d'ici quelques années. Les technologies de l'information et de la communication, amenées à prendre une place si importante dans ce secteur, y sont encore mal maîtrisées et mal exploitées. Les seuls qui s'y sont aventurés pour le moment sont les sites d'offres d'emploi ; ce n'est absolument pas ce que je veux faire. C'est une approche de masse sans réelle valeur ajoutée ; mon métier, c'est celui du conseil, qui consiste à suivre des missions sur la durée, à établir de réels contacts avec l'entreprise et le candidat.»

Le système envisagé par David comprend bien entendu un site Web décrivant les activités du cabinet et proposant une palette d'offres urgentes et de recrutements en cours, mais ce n'est que la partie la plus visible d'un outil plus complet. La principale valeur du système réside dans sa capacité à soutenir le travail des consultants auprès des entreprises ; il est conçu comme un complément «organique» à un processus rigoureux et bien rodé.

David a estimé approximativement les retours sur investissement qu'il peut retirer d'un système sur mesure : les gains de temps et d'efficacité potentiels lui permettraient de proposer à ses clients un tarif de commissions attractif et un fort engagement sur les résultats. Combinés à une visibilité accrue grâce au support Internet, ces atouts représentent un bénéfice potentiel chiffré en centaines de milliers de francs seulement sur les deux premières années d'activité, mais «explosant» avec la phase de croissance de l'entreprise.

Une approche «traditionnelle»

C'est sur la base d'un premier outil, «bricolé» à l'aide de Microsoft Access, que David a conçu dans les grandes lignes son système d'informations. Et même plus, à vrai dire, que dans les grandes lignes ; il a mis au point un «cahier des charges» relativement épais qui détaille un grand nombre de fenêtres, de champs de formulaire, de requêtes proposées…

C'est ce document qu'il a proposé à Alex, son homologue d'Advent, pour servir de base à une proposition commerciale. Il ne reflète pas nécessairement les besoins de façon précise – il a par exemple été rédigé sans qu'il ne soit véritablement question d'un système accessible par le Web ; cette orientation est relativement récente dans le projet d'entreprise de David, même si elle est depuis devenue fondamentale. Pour David, il est clair que c'est du ressort du prestataire de «refondre» cette première version pour établir une proposition précise et surtout chiffrée.

C'est à Leslie, en qualité d'architecte et de chef de projet, qu'incombent en principe cette refonte ainsi que la tâche de «chiffrer» ce cahier des charges – déterminer le nombre de «jours-hommes» nécessaires à la réalisation du produit qu'il décrit. C'est du moins la vision qui ressort de l'entretien initial au cours duquel Alex présente le projet à Leslie.

D'origine américaine, installée à Paris depuis un an, Leslie a découvert XP et en a adopté les principes de base depuis quelques mois. Elle possède plus de dix ans d'expérience professionnelle en programmation et, sur une échelle relativement modeste, en gestion de projet. Cette expérience, son pragmatisme et un succès sur une première mission «solo» au sein d'Advent (la réalisation d'une infrastructure pour la prise de contacts *via* le Web destinée au site de l'entreprise) lui valent ces nouvelles responsabilités.

Elle attribue ce premier succès en premier lieu à la pratique des techniques de programmation XP – tests, conception évolutive, refactoring, etc. – et ne s'en est pas cachée ; d'un naturel relativement timide, elle a pourtant fait preuve d'un enthousiasme remarqué lors de nombreuses discussions à ce sujet avec ses collègues.

Contractualisation selon XP

Pour Leslie, qui l'a plusieurs fois vécue par le passé, la démarche que lui a proposée Alex est un signal d'alarme. Avant même son démarrage – et, plus grave, sans que ce ne soit dit explicitement –, le projet s'oriente vers une approche forfaitaire classique. Elle implique, à l'issue d'une phase d'analyse, la rédaction rituelle d'un document conséquent, puis l'émission d'une évaluation subjective et forcément approximative du coût global – plus probablement déterminée par le budget du client, dont il a déjà été question, que par sa réelle adéquation au périmètre fonctionnel envisagé ; et qui fera fatalement l'impasse sur la capacité de production réelle de l'équipe envisagée, ladite équipe n'ayant jamais réalisé de projet similaire.

S'ensuivra une phase d'implémentation probablement chaotique, ponctuée de «points projet» qui constateront les retards accumulés ; ces réunions seront l'occasion de prendre acte de diverses demandes d'évolution de la part du client à mesure que l'expression initiale de son besoin sera confrontée avec la réalité de la solution mise en place. Ces demandes d'évolution donneront lieu à autant d'avenants au contrat initial, permettant d'aménager les débordements de délais et de budget. L'issue plus ou moins satisfaisante du projet dépendra principalement de l'élasticité dudit budget, de la capacité des programmeurs à rattraper le retard accumulé à coup d'heures supplémentaires, et très accessoirement de l'exactitude de l'estimation initiale.

C'est en tout cas le tableau que brosse Leslie du projet typique – nous lui laissons bien entendu l'entière responsabilité de ces propos. Alex, pour sa part, y voit une description assez fidèle de projets précédents, et comprend suffisamment bien la façon dont l'approche XP semble contourner ces difficultés, pour se laisser convaincre de proposer à David, d'une part, de s'impliquer de bonne volonté dans cette démarche et, d'autre part, de lui accorder un cadre contractuel aménagé. Ce cadre contractuel est basé sur une logique d'assistance technique qui laisse une grande marge de manœuvre sur la gestion du contenu fonctionnel du produit, mais

il prévoit en revanche une clause de «dégagement anticipé» : si, au terme d'une itération, David estime que le projet n'avance pas suffisamment vite ou qu'il prend une mauvaise tournure, il peut l'arrêter aussitôt et récupérer le produit – sources, tests, etc. – en l'état. Dans ce contexte, il revient donc à Leslie et son équipe d'apporter suffisamment de valeur ajoutée au produit à chaque itération pour que le projet dure aussi longtemps que possible.

Formation

Ayant reçu carte blanche pour mener le projet comme elle l'entend, Leslie doit faire face à deux tâches immédiates : en premier lieu, jeter les bases d'un plan de réalisation d'ensemble – qui sera bien compris comme provisoire et soumis à modifications, mais qui doit par ailleurs identifier les principaux risques «architecturaux» ; en second lieu, recueillir l'adhésion de son équipe à la méthode proposée et identifier d'éventuels risques en rapport avec les compétences individuelles.

Leslie et ses collègues ont parlé d'XP à plusieurs reprises, de façon relativement informelle. Yves et Sébastien se sont exercés, à temps perdu, à l'utilisation d'un outil de tests unitaires, et ont tiré profit des notions de remaniement dans leurs propres activités ; celles-ci consistaient pour l'essentiel à réaliser des pages HTML/JavaScript pour le site de l'entreprise, ce qui ne leur a pas tout à fait permis de les exploiter pleinement. Selon Leslie, une session «Heure Extrême» (voir chapitre 7) s'impose pour concrétiser certaines notions et garantir que l'équipe partage une même vision d'ensemble de la démarche. Cette formation réunira les développeurs et le client autour d'un pseudo-projet.

Au-delà de ses aspects informatifs, cette démarche permet également à Leslie de tuer dans l'œuf un conflit potentiel. Sébastien, de deux ans son aîné, remplit dans l'entreprise les fonctions de directeur technique, sans en posséder officiellement le titre ; nommé à l'origine Responsable d'exploitation, il a supervisé la mise en place de l'infrastructure réseau d'Advent et a assuré la coordination des tâches confiées à Yves et Benoît au cours des premiers mois d'activité. Ses qualités et son sens relationnel en font d'une certaine manière un chef naturel. La nomination de Leslie aux fonctions de chef de projet a de ce fait créé des tensions mineures, mais qu'elle souhaite résoudre avant qu'un malaise ne s'installe. Une conversation très ouverte autour de la répartition des rôles et des responsabilités en XP, à l'issue de la session, leur permet de trouver un terrain d'entente : Leslie parvient à rassurer Sébastien sur la compatibilité et la complémentarité de leurs ambitions.

Exploration du besoin

Dans le même temps, Leslie met en œuvre ses compétences d'architecte. Il ne s'agit pas, pour elle, de fixer les choix techniques mais de faire le point avec le client et l'équipe sur les difficultés techniques (réelles ou supposées) susceptibles d'hypothéquer la réussite du projet. À ce stade, les risques identifiés concernent le stockage des données : l'expérience de David avec Access le conduit à envisager comme évidente une solution qui s'appuierait sur une base de

données SQL ; Alex a assisté à une présentation sur la technologie EJB et pense qu'une compétence dans ce domaine pourrait être valorisée. L'utilisation d'une base de données objet est une troisième voie que Leslie suggère d'envisager. Elle soulève par ailleurs une problématique moins immédiatement perçue par ses collègues, celle qui touche au serveur d'application Web. Advent est stratégiquement engagée sur les plates-formes Java et notamment J2EE, mais cette orientation laisse ouverts d'autres choix : orientation XML, utilisation de JSP ou d'un des divers «modèles MVC» pour l'implémentation des pages Web dynamiques par exemple. Tous ces aspects devront être étudiés lors de la première itération du projet, qui restera en partie exploratoire.

Afin d'initialiser la planification, l'équipe travaille avec David pendant deux journées consécutives pour extraire du «cahier des charges» initial une série de… fiches bristol, chacune décrivant en une ou deux phrases une fonctionnalité importante du projet. Sur chaque fiche, deux annotations chiffrées sont apposées, l'une par les développeurs – un chiffre de 1 à 5 représentant la durée estimée pour réaliser cette fonctionnalité–, l'autre par David – une indication de P1 à P5 représentant l'importance relative de chaque fonctionnalité. Laborieusement au départ, pour l'essentiel parce qu'il est très difficile à David de ne pas répondre «mais *tout* est important pour moi dans ce projet», puis avec plus d'entrain, le projet est ainsi découpé en scénarios client.

Le projet vient de démarrer formellement au sens d'XP.

La théorie…

Il en va des conditions conjoncturelles comme des conditions structurelles : tous les projets XP ne naissent pas égaux en ce qui concerne l'avant-projet. Cependant, dès lors qu'une décision formelle est prise de fonctionner selon XP, il est souhaitable de distinguer nettement la «phase projet» de tout ce qui précède. La plupart des projets font de toute façon l'objet d'un lancement formel, ce qui peut être une bonne occasion pour marquer cette séparation. Il est souhaitable de communiquer cette impulsion à l'équipe entière.

Itération 0 : calibrage

La toute première itération du projet revêt une importance d'autant plus grande que l'équipe travaille ensemble pour la première fois et aborde un domaine fonctionnel nouveau. Personne ne connaît à ce stade la correspondance entre «une fiche de 1 jour théorique» et un nombre de jours calendaires ou ouvrés. Cette première itération permettra de mesurer dans des conditions réelles la capacité de l'équipe à produire de la valeur, capacité dont Leslie et Sébastien s'accordent à penser qu'elle peut dépendre de nombreux facteurs impossibles à estimer *a priori* : aux variations de compétences individuelles s'ajoutent la motivation de chacun, la bonne ou la mauvaise entente des membres de l'équipe, l'adéquation des conditions de travail à la démarche, la difficulté intrinsèque des technologies mises en œuvre…

Infrastructure

Le calendrier se prête à un démarrage optimal : l'itération 0 débute un lundi matin. La durée des itérations a été fixée par consensus à deux semaines ; plus courte que les trois semaines préconisées de façon classique pour les projets industriels, plus confortable que les itérations d'une semaine recommandées par les plus radicaux des chefs de file de la communauté XP.

Une première réunion de lancement permet à chacun de s'accorder sur les tâches prioritaires. Benoît soulève une question épineuse – les pratiques XP supposent un certain nombre de prérequis techniques, mais tout le monde n'est pas encore doté des outils adéquats ; leur mise en place prendra du temps, qui n'est pas encore prévu au planning. La solution est heureusement vite trouvée : il suffit d'écrire et d'estimer un scénario client «mise en place de l'équipe». Pour prendre en compte son effet sur le planning, on décalera une des fiches de l'itération 0 à la suivante.

La théorie…

L'équipe passe ainsi outre la vocation des scénarios client, qui est d'exprimer une réelle exigence du client. Injecter dans la planification des tâches qui ne correspondent pas à une création de «valeur client» est une dérive souvent constatée dans les équipes inexpérimentées, qui peuvent trouver «injuste» que leur vélocité soit affectée par des tâches annexes mais néanmoins nécessaires. Une équipe expérimentée accepterait plus facilement que, lors de la première itération, sa vélocité se révèle très faible. Il faut cependant noter que l'utilisation de ces «faux» scénarios client pour communiquer de l'information sur un retard déjà anticipé relève bien de l'esprit de franchise qui caractérise les relations au sein d'une équipe XP.

Leslie aide l'équipe à mettre en place l'infrastructure nécessaire. Elle commence par des aspects apparemment triviaux – en tout cas non techniques. Un tableau blanc pour servir de support aux réflexions sur la conception ; un aménagement des plans de travail plus resserré, réunissant tous les intervenants autour d'un seul bloc de bureaux, avec de larges espaces autour pour faire circuler les sièges sur roulettes ; un grand tableau adhésif en imitation liège, sur lequel les fiches bristol correspondant aux fonctionnalités des itérations en cours et à venir peuvent être fixées et repositionnées.

Les choses «sérieuses» commencent ensuite. Sébastien, qui a déjà pratiqué certains de ces outils, configure le serveur d'application dans un mode minimaliste ; ainsi préparé il peut accueillir le code Java créé par les développeurs, éventuellement servir des pages JSP. Assisté de Benoît, il met également en place le serveur SourceSafe, utilisé pour conserver un historique des versions successives des fichiers source, et en automatise la sauvegarde. Leslie et Yves mettent en place les outils de test ; JUnit servira pour les tests unitaires, HTTPUnit pour les tests de recette. L'environnement de compilation Ant permettra d'automatiser la production d'un «exécutable» (en fait, un module à intégrer dans le serveur d'application) à partir du code source.

Le mardi, avant l'heure du déjeuner, tout est en place… avec une demi-journée d'avance. Un premier succès savouré autour d'une bonne table.

De la valeur, tout de suite

Le plan de livraison, en définitive, est encore assez flou. Les fiches choisies pour l'itération 0 l'ont été strictement sur la base des priorités établies par David entre les différentes fonctionnalités du système, mais il s'inquiète de ne pas pouvoir « montrer quelque chose » rapidement. Après une brève conversation, il ressort que David souhaite pouvoir ouvrir son site au public le plus rapidement possible, même si les fonctionnalités propres à la gestion du métier du cabinet de recrutement ne sont pas immédiatement disponibles.

Un nouveau plan d'action est rapidement fixé. L'itération 0 aura pour objectif de fournir un prototype du site : une page d'accueil, deux brefs textes fournis par Sonata pour présenter l'entreprise, une offre d'emploi urgente, deux formulaires permettant respectivement aux entreprises et aux candidats de s'enregistrer pour recevoir plus d'informations par mail. Au cours de l'itération 1, le site sera étoffé avec un contenu plus riche et un moteur de recherche sur les offres, avec pour objectif de mettre en ligne à la fin de l'itération ou au cours de la suivante.

L'équipe est un peu surprise de ce changement de priorités mais l'accepte avec bonne volonté – après tout, il suffit de déplacer quelques fiches sur le tableau de planning. Sébastien se déclare satisfait, tant d'avoir ainsi la preuve de la capacité d'XP à réagir aux changements que d'avoir l'occasion, dès la première itération, d'explorer des choix techniques majeurs, puisque les fonctionnalités demandées couvrent à la fois la présentation de contenus dynamique et le stockage permanent des données stratégiques – les informations sur les candidats et les entreprises.

Premières « barres vertes »

Les premières tâches d'implémentation sont pour Leslie l'occasion de mesurer les compétences mais aussi d'appréhender les personnalités des membres de l'équipe. L'expérience professionnelle de Sébastien, dont elle a déjà apprécié la franchise et l'ouverture, concernait avant son recrutement à Advent le domaine de la sécurité sous Windows ; il a notamment pratiqué le C++. Yves est un programmeur plutôt rigoureux, avec une première expérience de Java, curieux d'apprendre de nouvelles techniques et ouvert au dialogue, même s'il n'en prend pas habituellement l'initiative. Benoît est plus réservé, parfois un peu brouillon. De toute l'équipe, c'est lui qui a le plus besoin d'assimiler de nouvelles notions techniques relatives à l'objet.

La pratique de la programmation en binôme est, au cours de ces deux premières semaines, plus opportuniste que systématique. Leslie et Sébastien, convaincus de son importance, ne cherchent cependant pas à l'imposer, préférant laisser chacun trouver son rythme. Pour

Sébastien, l'aspect crucial de la pratique consiste à s'assurer d'un travail méthodique et sans bavure ; elle lui permet notamment d'assurer un rôle de «pivot», en donnant à chacun des instructions claires sur le travail qu'il laisse ensuite réaliser de façon autonome. Ces aspects sont moins importants pour Leslie, qui y voit avant tout l'occasion de transmettre une véritable philosophie de l'objet ; écrire du code est pour elle une activité littéraire, qui nécessite de comprendre intuitivement les caractéristiques du langage et de les exploiter au mieux, et de prêter une attention particulière à «ce que dit le code».

L'implémentation des premières fonctionnalités s'effectue conformément aux principes d'XP. L'équipe s'assure d'avoir bien compris l'intention du client, puis aborde la réalisation en rédigeant en premier lieu un test unitaire, le code correspondant étant ensuite écrit le plus simplement possible. Et ainsi de suite : petit à petit, les tests, et parallèlement les fonctionnalités, s'accumulent.

Par ailleurs, des tâches plus expérimentales permettent de découvrir quelques technologies importantes ; un prototype démontre l'intérêt d'une base de données objet, Adret, par rapport aux solutions SQL plus connues, bien que David et Alex restent sceptiques. Un autre permet à Leslie de montrer par l'exemple ce qu'on peut attendre d'un modèle de conception MVC (Modèle – Vue – Contrôleur) pour l'implémentation de pages interactives complexes à base de formulaires HTML. Les technologies XML et XSL pour la présentation du contenu sont également adoptées.

La satisfaction des premiers tests unitaires procure aux développeurs un sentiment d'achèvement fréquemment renouvelé ; quelques frustrations également, lorsqu'un test qui devrait passer se montre récalcitrant. Mais dans l'ensemble l'impression de progrès est palpable, l'ambiance générale très positive.

Vers la fin de l'itération, une «démo» donne l'occasion à David de faire quelques remarques générales. Un seul «bug» se manifeste à cette occasion – relativement embarrassant cependant, les messages rédigés par les candidats arrivant à l'adresse e-mail «entreprises», et *vice versa*.

Le bilan de l'itération reste cependant positif. La vélocité mesurée est de 27 «jours théoriques» ; une déception, toutefois, du seul point de vue d'Alex, pour qui le calcul nominal – quatre développeurs travaillant à raison de dix jours par itérations, soit 40 jours théoriques – suggère une performance médiocre. L'équipe explique qu'il ne s'agit que d'une productivité théorique et que la vélocité est justement corrélée avec la productivité réelle – la seule intéressante, qui peut d'ailleurs très bien s'améliorer. En tout état de cause, le périmètre total du projet est estimé à 159 jours ; certaines des fonctionnalités inscrites étant classées en priorité «ce serait bien si…», un planning sur 5 itérations reste tout à fait faisable, et conforme au cadre contractuel établi.

La théorie…

Certains vétérans d'XP recommandent d'utiliser systématiquement une notion de «points» sans aucun rapport conceptuel avec des jours calendaires, plutôt que des «jours théoriques» comme l'équipe d'Advent. Les «jours théoriques» ont le mérite de la simplicité et d'un «parler clair» vis-à-vis du client mais ils risquent de remettre en cause le phénomène d'abstraction autorisé par une notion de «points», qui permet aux développeurs de faire des estimations plus librement et honnêtement. Inversement, un autre argument oppose que la notion de «points» doit en définitive correspondre (*via* un coefficient bien entendu non constant) à des jours calendaires, et que les estimations sont «senties» plus justement en réfléchissant en termes de temps.

Itération 1 : mise en production

«Et n'oubliez pas, on est en ligne le 15.» Une phrase qui pourrait être anodine, prononcée par Sébastien à mi-itération, une semaine seulement avant la date fatidique. Elle ne sera pas relevée immédiatement comme absurde, mais sera souvent évoquée, une fois cette «deadline» manquée, comme symbole de deux éléments symptomatiques des projets «classiques». D'une part, l'alchimie subtile de la prise de décisions : sans que rien d'officiel ne soit réellement dit à ce sujet, l'estimation approximative d'une première date de livraison est devenue une *deadline*, et va peu à peu se transmuer en un «engagement commercial impératif». D'autre part, le caractère totalement gratuit de cette décision ; l'infrastructure réseau étant déjà en place, une page «en cours de construction» étant déjà affichée depuis plusieurs semaines à l'adresse Web de Sonata, rien – aucune conférence de presse, aucun rendez-vous avec sa clientèle potentielle – n'oblige l'entreprise à respecter une date précise.

Tests de recette

Cet épiphénomène mis à part, le projet s'oriente objectivement de façon satisfaisante. L'itération 0 n'a pas à proprement parler produit de tests de recette ; on s'est accordé à considérer les tests unitaires qui ont piloté l'écriture des premiers composants comme suffisants pour «couvrir» la totalité des fonctionnalités correspondantes.

Dès le début de l'itération 1, Leslie cherche à affirmer la distinction entre tests unitaires et tests de recette. Les tests unitaires «voient le code du point de vue du code», pour reprendre une de ses expressions sans doute un peu obscure mais très juste : un test unitaire est d'autant plus efficace et informatif qu'il isole un composant, voire une partie d'un composant (une ou plusieurs méthodes). Les tests de recette adoptent un «point de vue» plus extérieur ; ils ont pour objectif de spécifier le comportement de l'ensemble du programme, tel que l'utilisateur peut le percevoir.

Concrètement, cela se traduit pour le projet C^{4}MS (Content/Candidate/Consultant/Company Management System) par l'utilisation du module HTTPUnit et d'un outil nommé Tidy

permettant de traiter les pages HTML émises par le serveur sous la forme de données XML. Le langage de requête XPath permet ensuite de traduire de façon relativement mécanique des assertions ou suppositions sur ce qu'une page donnée doit ou ne doit pas contenir :

```
// Le champ «nom» dans la table «infos perso» contient le nom du candidat
assertEquals("//*/table[@name='infos']/*/div[@name='nom']/text()","Durand");
```

Les noms mystérieux de ces outils recouvrent en fait une infrastructure relativement simple du point de vue technologique et qui, par ailleurs, permet aux développeurs qui s'essaient à écrire ces tests de recette de renforcer leurs compétences dans ces domaines. Outre ces aspects techniques, Leslie s'attache également à faire comprendre à David et Alex l'importance de ces tests en termes de pilotage du projet. «Les tests doivent vous rassurer quant au fait que le système fait bien ce qu'on a dit, et on peut même les considérer comme une recette automatisée, d'où leur nom. Alex, le but du jeu, c'est que si tous les tests passent David peut signer ta facture les yeux fermés. David, ça peut paraître risqué du point de vue d'une gestion de projet plus contractualisée, mais n'oublie pas que le deal, c'est que tu as le droit de nous spécifier un nouveau test si tu as le moindre doute sur la capacité du système à faire telle ou telle chose ; notre mission, c'est de faire passer tous les tests.»

XP stipule que les tests de recette sont «sous la responsabilité» du client, mais cela n'implique pas forcément qu'il les écrive lui-même. Comme il est essentiel que l'effet de confiance dont elle parle fonctionne, Leslie a choisi ce qui semble la solution la plus simple, faire écrire les tests par les développeurs eux-mêmes d'après une écoute attentive de «scénarios» de test proposés par David. Lorsqu'il est en difficulté pour exprimer avec assez de précision ces scénarios, l'équipe l'aide – le plus souvent avec patience – en posant des questions : «Imagine que je vienne te voir en te disant : "Ça y est, j'ai fini cette fonctionnalité", et que tu ne me croies pas. Qu'est-ce que tu regardes dans la page HTML pour pouvoir me dire : "Ne te fiche pas de moi, ce n'est pas vraiment fini" ?»

La théorie…

Dans certains contextes, le client fera appel à un ingénieur recruté pour l'occasion ou issu de la fonction qualité de l'entreprise, mais en tout cas dépendant de lui hiérarchiquement : c'est alors cet ingénieur qualité qui formalisera les tests de recette. Si cette solution est avantageuse, elle ne dispense cependant pas des conversations entre le client et les développeurs.

Difficultés de mise en pratique

Le site de Sonata – ou plus exactement sa version de développement, visible à ce stade seulement sur le serveur de test d'Advent – commence à prendre forme. Une interface simple d'administration des contenus permet maintenant aux consultants de créer de nouvelles pages d'information ou d'offres d'emploi, et d'accéder aux renseignements sur les candidats ou les entreprises. Elle se limite pour l'instant à un menu hiérarchique de navigation, quelques boutons et une simple «boîte de texte» pour la saisie ou la modification des données, présen-

tées sous leur forme XML brute. Ces limitations contraindront les consultants de Sonata à se former à l'utilisation de la notation XML mais, après une première conversation avec ces utilisateurs finaux, cela ne représente pas un obstacle majeur ; en l'état, ces fonctionnalités constituent déjà une réelle valeur pour Sonata.

Benoît s'est intéressé à la notion XP de métaphore, et a suggéré une réunion d'ensemble pour chercher à décrire le système d'une façon plus parlante. Yves appuie énergiquement cette demande ; il s'inquiète de ce que le code semble «partir dans tous les sens» sur le plan du nommage des classes et des méthodes et relève aussi l'incohérence de certains termes utilisés pour parler des diverses fonctions. L'interface d'administration est appelée «back office» par les techniciens et simplement «l'admin» par David, les données persistantes sont un «repository» pour Leslie et une «base de données» pour les autres, et ainsi de suite.

Malgré une séance de brainstorming qui tourne au marathon – et explore des terrains aussi variés que «une bibliothèque», «un terrain de foot», «un atome d'uranium», «le métro parisien» –, l'équipe n'arrive pas, en définitive, à tomber d'accord sur une vision d'ensemble qui reflète assez la structure, l'architecture ou la conception du système, et ne paraisse pas «trop ridicule pour pouvoir l'utiliser au moment de nommer des classes».

Yves est également le premier à s'inquiéter publiquement d'un manque de cohérence sur un autre plan, celui du formatage, et plus généralement de l'aspect visuel du code. Son style habituel, qu'il est le seul à pratiquer, a été influencé par un précédent projet dans la finance ; l'une des classes du système, parmi les premières entièrement codées par Yves qui prend depuis peu une nette assurance, en est presque une caricature :

```
/** CCandidate représente un candidat dans le système
* (c) Advent 2000
* @author YVO
*/
class CCandidate {
// Nom du candidat
String m_name;
// Age du candidat
int m_age;
// Sexe du candidat
boolean m_hf;
         /** Renvoie une formule de politesse pour les emails adressés au
        * candidat en concatenant Mr/Mmme selon le sexe + le nom
        String politenessFormula() {
        StringBuffer result = new StringBuffer;
        if (m_hf)
        result.append("Mr.");
        else result.append("Mme");
        result.append(" " + m_name);
        return result.toString() ;
}
```

```
/** Accesseur pour 'nom' */
String getName() {
```

Le même code après avoir subi de la part de Leslie un remaniement plutôt musclé va ressembler à ceci :

```
class Candidat
{
public static final int HOMME = 0;
public static final int FEMME = 1;
private String nom;
private intage;
private intsexe;
String salutation()
{
return sexe == HOMME ? "Mr." : "Mme" + " " + nom;
}
```

Leslie justifie ses choix principalement par un souci de lisibilité et de simplicité ; les divers «préfixes» utilisés pour distinguer un nom de classe ou de variable n'ont pas d'intérêt tant que classes et méthodes restent courtes et bien factorisées ; les commentaires ne s'imposent pas s'ils ne sont que des redites de ce que le code rend déjà explicite. L'usage systématique de méthodes d'accès aux variables des classes est contraire à la notion de simplicité XP – ne pas écrire du code qui ne soit pas nécessaire pour faire passer un test, ce qui n'est pas le cas ici.

Enfin – et c'est peut-être le plus grave pour Leslie –, il vaut mieux éviter d'écrire du code en «franglais» ; l'expression «formule de politesse» ne se traduit pas littéralement en anglais. Leslie – elle-même encore un peu hésitante en français – argumente qu'il vaut mieux écrire le code dans la langue qu'on maîtrise le mieux, et récuse l'idée selon laquelle un code écrit en anglais serait «plus professionnel». À moins d'exiger des développeurs qu'ils maîtrisent parfaitement la langue de Shakespeare, ce qui n'a pas été le cas lors du recrutement, invoquer le professionnalisme serait une aberration dans ce contexte.

Une conversation plutôt tendue avec Yves permet en définitive de trouver un compromis acceptable par tous. Advent n'a pas officiellement adopté de règles de codage, mais le langage Java constitue un choix stratégique. Tout en rappelant que la seule règle utile reste : «Le code qu'on écrit doit ressembler à celui qui est déjà écrit», Leslie suggère d'adopter, pour arbitrer d'éventuels conflits, les conventions de Sun, la nomenclature et les règles de formatage auxquelles répond le code source des classes de base de Java ; y compris lorsque celui-ci ne correspond pas à ses propres préférences, comme c'est le cas par exemple sur le placement des accolades.

L'équipe se promet par ailleurs de pratiquer plus assidûment la programmation en binôme, afin d'éviter à l'avenir que des divergences aussi importantes entre les styles individuels compromettent la cohérence du code.

Visibilité

Alors que cette deuxième itération touche à sa fin, l'équipe accumule les heures supplémentaires, «collectivement hypnotisée» – pour reprendre les termes de Benoît – par la date imposée de mise en ligne, allant jusqu'à se réunir le week-end du 16 pour terminer quelques-unes des tâches, certaines non prévues par le planning initial, restant à effectuer avant l'ouverture au public.

Malheureusement, l'essentiel de ce «temps volé» sera pour Leslie accaparé par une discussion avec Alex, qui se plaint de «manquer de visibilité» sur l'architecture générale du système. Agacée qu'on puisse parler de visibilité alors que l'équipe déploie depuis quatre semaines une énergie considérable à montrer régulièrement des indicateurs de progrès – plus de cent tests unitaires, une transparence complète vis-à-vis de David sur les problèmes rencontrés et les solutions trouvées, des démonstrations régulières –, elle parvient cependant à guider la conversation jusqu'au cœur du problème.

Alex voit le système encore embryonnaire que l'équipe met au point comme un capital technique pour Advent ; aux termes du contrat avec Sonata, Advent restera propriétaire des parties centrales du code source, ce qui n'est généralement pas le cas dans une prestation de ce type, et ce qui a permis à Advent de faire une offre commerciale compétitive. Pour Alex, cette qualification comme «technologie de base» implique une vision plus «industrielle» du système, ce qui devrait selon lui se manifester par une documentation de conception plus conséquente. Il s'inquiète également de ce que le système de persistance des données s'appuie toujours sur une solution «simple» au sens XP – les objets principaux sont encore stockés sous la forme de fichiers, aucune base de données n'est encore utilisée compte tenu des volumétries encore limitées. Sans s'engager à des actions précises, Leslie promet néanmoins d'apporter des réponses circonstanciées à ces interrogations.

Le site n'est finalement mis en ligne que le 19. La vélocité calculée pour l'itération 1 n'est que de 23 points ; dès le premier jour, des utilisateurs extérieurs font état de deux défauts non bloquants mais sérieux.

La théorie…

Tous les projets connaissent des moments de stress ; un projet XP ne fait pas exception. La difficulté consiste à ne pas abandonner définitivement toute discipline en revenant brusquement aux pratiques familières, plus sécurisantes, même si elles sont nettement moins efficaces.

Itération 2 : croissance

Au cours des jours suivants, le bon accueil fait au site de Sonata par ses premiers visiteurs et l'excellente stabilité du système (une fois les défauts constatés corrigés) redonnent confiance à l'équipe, qui se résout cependant à ne plus déroger à la règle du rythme durable. Le projet entre par ailleurs dans une seconde phase, plus sereine, avec pour objectif de fournir les

premières fonctionnalités destinées aux consultants de Sonata pour la gestion des entretiens avec les candidats.

Notions métier

L'itération en cours a pour objet de jeter les bases des fonctionnalités de gestion des entretiens. Plus que la précédente, elle donne donc l'occasion à l'équipe de se mettre sous la dent des notions fonctionnelles concrètes et de tirer parti de la proximité du client. Les efforts sincères fournis par chacun pour se faire comprendre de David lorsqu'il s'agit de questions techniques, et aussi de réellement le comprendre lorsqu'il s'agit de questions fonctionnelles, conduisent petit à petit à deux résultats concrets : d'une part, une infrastructure plus avancée de tests de recette, basée sur un langage de script rudimentaire, et, d'autre part, un «glossaire» évolutif du système résumant rapidement les principaux termes métier utilisés ou susceptibles d'être utilisés dans le code. Ce glossaire est la première «documentation» technique produite spontanément par l'équipe, qui va se révéler au fil des jours la plus utile.

Cette activité d'analyse conduit l'équipe à rencontrer non seulement David mais également, sur une suggestion de ce dernier, sa propre équipe de consultants recruteurs, dans leurs propres locaux, pour mieux comprendre comment fonctionne le «processus métier» que le système doit conduire à optimiser. David reste seul décisionnaire tant sur ce qui doit être réalisé et pour déterminer quand ces contacts avec les utilisateurs finaux sont payants. Cela permet par exemple à l'équipe d'identifier des règles fondamentales qui n'ont pas été explicitement formulées comme des scénarios clients mais qu'il est important que le système respecte : par exemple, il est absolument hors de question d'aborder pour un recrutement un candidat déjà en poste chez une entreprise cliente – alors même que le fichier des candidats «placés» par Sonata et suivis ultérieurement peut constituer, sous réserve du respect de cette règle, la partie la plus précieuse de la base de données.

C'est également en interrogeant Nicolas, directeur financier de Sonata, qui sera à ce titre particulièrement vigilant quant aux modules liés à la facturation, que certaines erreurs seront évitées… qui n'auraient pas manqué d'être signalées comme des «bogues». De même, les retours recueillis par l'équipe sur l'utilisation du système en conditions réelles permettent d'améliorer l'ergonomie de certaines fonctionnalités et de proposer certaines simplifications qui faciliteront la suite des travaux : ainsi l'un des formulaires pour l'évaluation post-entretien des candidats, alourdi de nombreux champs qui ne sont que très rarement renseignés lors d'entretiens réels, se voit simplifié par la fusion de ces derniers en un unique champ «notes».

Un seul véritable «bogue» – ou plus exactement défaut du système – est relevé lors d'un de ces rendez-vous, que Benoît s'empresse de corriger en regagnant Advent, avant de livrer une version corrigée… le tout en moins d'une heure.

Évolution de l'équipe

Frank rejoint l'équipe vers la fin de l'itération. Il serait plus exact de dire qu'il va y être progressivement intégré, sur une durée de plusieurs semaines : «pris en charge» gratuitement par Sonata à la suite d'une candidature spontanée, il a eu l'occasion de rencontrer les autres membres de l'équipe… autour d'un café mais également autour d'un clavier. Durant plusieurs séances d'une heure environ, en marge des horaires de travail de l'équipe, réparties entre un «essai» de programmation en binôme sur une tâche réelle et une discussion informelle, chacun a pu prendre la mesure de ses compétences – nettement suffisantes en objet, épicées d'un soupçon d'exotisme par la pratique de langages fonctionnels tels qu'Erlang – et surtout de sa personnalité – loquace et extraverti, sportif, offrant un certain contraste avec le reste de l'équipe.

Le «véritable» entretien d'embauche de Frank n'a lieu que comme une ultime étape, Alex entérinant – comme mis devant le fait accompli – la décision de l'équipe et expédiant les démarches administratives, y compris l'aval de son propre supérieur, sans délai supplémentaire. Ce déroulement peu orthodoxe n'a pas été officiellement décidé ou préalablement discuté ; tacitement encouragé, notamment par Leslie et David, il s'est presque imposé comme une évidence.

La fin prochaine de l'itération et l'arrivée de Frank donnent à Leslie l'occasion de proposer une réunion générale pour effectuer un bilan provisoire de la méthode. Le format est classique – les questions cruciales en sont : «Qu'est-ce qui a bien marché, qu'est-ce qui a moins bien marché, qu'est-ce qu'on pourrait faire différemment ?» Les différentes pratiques d'XP sont examinées une par une pour tenter d'évaluer le «degré de maîtrise» dont l'équipe a fait preuve pour chacune.

Sur une suggestion de Benoît, l'équipe tente également de faire le point sur les adaptations locales : tout ce que fait l'équipe qui n'est pas directement recommandé par XP. Il s'agit d'en déterminer les objectifs et les effets réels : par exemple, *quid* de cette habitude, devenue tradition, de tester collectivement de nouveaux outils pour la gestion du code, comme ce système d'analyse des dépendances «démoli» le vendredi passé pour son manque d'ergonomie ? Frank est chargé, en tant que «candide» par rapport à ces adaptations locales, de jouer un rôle d'observateur pendant l'itération à venir. Surtout, l'équipe décide de systématiser ces bilans sur la démarche, qui seront l'occasion de l'optimiser.

Du Zen et de l'écriture des tests unitaires

Le dernier jour de l'itération, l'un des consultants de Sonata appelle pour faire état d'une «régression» dans le module de gestion des entretiens – un défaut précédemment corrigé serait réapparu. Après examen, Sébastien constate qu'il s'agit en fait du «bogue» corrigé par Benoît à la suite de l'une de ses visites chez Sonata, lequel n'a en fait pas été résolu, malgré il est vrai une modification de deux lignes dans le code – qui se révèle en fait inefficace.

En comparant leurs expériences à la suite de cet incident, l'équipe s'accorde pour ne faire *aucune* exception à la règle XP qui veut qu'aucun code ne soit écrit – y compris pour la correction d'un défaut – sans que le test unitaire correspondant ne le soit au préalable. La règle est désormais formelle : même pour la correction d'un défaut ou d'un problème mineur, on écrira d'abord un test qui doit échouer avant la correction et passer après la correction. Les tests unitaires serviront à la fois de «rapports d'incidents» et de «rapports de correction».

Les tests unitaires fournissent également à l'équipe un atout précieux pour une autre catégorie de problèmes – les défauts, hélas assez nombreux, du serveur d'application. En appliquant la même démarche que pour leurs propres erreurs, Yves et Benoît sont maintenant à même de corriger de manière fiable et démontrable deux problèmes majeurs dans la gestion des cookies par le site de Sonata. Avec le temps, il semble de plus en plus à l'équipe que «rien ne résiste aux tests».

La théorie…

L'intégration de «l'esprit XP» à des aspects du fonctionnement de l'entreprise ou de la relation client autres que ceux qui en dépendent «officiellement» sera certainement un indicateur majeur du succès rencontré par la démarche, de même que la capacité de l'équipe non seulement à s'autodiscipliner dans la mise en œuvre des pratiques mais également à mettre au point des adaptations d'XP appropriées.

Itération 3 : stabilisation

Poursuivant cette alternance entre des thèmes fonctionnels et des thèmes techniques esquissée par les deux itérations précédentes, l'itération 3 a pour principal objectif de fournir une interface plus efficace et intuitive pour la gestion des contenus rédactionnels, des descriptifs de mission et des offres d'emplois diffusés sur le site de Sonata. Cette interface permettra à Sonata, jusqu'alors dans une certaine mesure tributaire d'Advent pour la mise à jour des contenus plus complexes, d'être autonome dans la gestion de son site Web.

Déploiement en continu, optimisation

La montée en puissance des tests unitaires et fonctionnels comme indicateur global de la qualité du système permet désormais à l'équipe d'automatiser entièrement le processus de déploiement. Le contraste est frappant par rapport à la procédure d'origine, instaurée par Sébastien. Celle-ci imposait de consigner dans un cahier prévu à cet effet toutes les étapes, mêmes triviales, de toute intervention technique effectuée sur les serveurs de production. Frappée au coin du bon sens, cette procédure, bien que pénible, garantissait que toutes les étapes du déploiement seraient reproductibles, et qu'en cas d'incident il était possible de tracer les différentes étapes une à une pour identifier la manœuvre fautive.

La nécessité est mère de l'invention et, dès les premiers jours, Leslie et Benoît, qui conçoivent tous deux la paresse comme une vertu, n'ont eu de cesse d'automatiser le plus possible ces procédures afin d'éviter le «pensum» que constitue la tenue du cahier d'intervention. Elles se réduisent désormais à un seul double-clic : en lançant un script créé sous Ant, initialement pour automatiser la compilation, depuis une machine de développement quelconque, on démarre sur le serveur «test» le processus suivant :

- L'intégralité du code source Java et des pages HTML ou JSP est rapatriée de SourceSafe.
- L'application serveur et les pages HTML sont entièrement recompilées.
- La totalité des tests unitaires et de recette est exécutée.
- Si une seule défaillance dans un test est constatée, un e-mail est expédié à toute l'équipe.
- Si tous les tests sont effectués avec succès, le code Java est «packagé» pour le serveur d'applications.
- Le fichier de déploiement (fichier WAR) est expédié sur le serveur «live».
- De façon optionnelle, des procédures de «migration» des données sont exécutées.
- Le serveur se met à jour automatiquement avec le nouveau code (une fonctionnalité propre au serveur d'applications utilisé, mais des plus appréciables).

La durée totale de l'opération est d'environ 20 minutes, pour l'essentiel le temps nécessaire pour l'exécution des 400 tests unitaires et de recette que comprend maintenant le système. Les nouvelles fonctionnalités sont donc disponibles pratiquement en temps réel pour être évaluées par les utilisateurs…

Loin de se satisfaire de ces 20 minutes – pourtant un temps record pour une recette –, l'équipe cherche au cours de l'itération à optimiser les performances du système : au cours du développement de nouvelles fonctionnalités, il n'est pas rare d'exécuter l'ensemble des tests toutes les X minutes, et la lenteur du jeu de tests total oblige les développeurs à ne s'intéresser qu'à un sous-ensemble : tests unitaires seuls ou tous les tests concernant la fonctionnalité en cours.

L'outil de «profiling» commercial OptimizeIt permet d'identifier les parties du code les plus consommatrices de temps ou de ressources : l'équipe constate ainsi que la base de données SQL est le principal goulet d'étranglement, suivi de très loin par les classes implémentant leur version du modèle MVC pour les pages Web interactives. Avec l'utilisation d'une base de données SQL «open source», plus rapide dans les situations de test que le serveur de production, on peut optimiser l'exécution des tests – quant au gain qu'il est possible d'attendre de l'optimisation des classes de l'équipe, il est relativement marginal et celles-ci sont en définitive laissées telles quelles.

L'équipe est désormais bien mieux rodée à l'utilisation d'un langage riche en métaphores, qui permet à tous de faire tendre les conversations sur des questions fonctionnelles plutôt que sur des détails techniques. Le vocabulaire partagé par tous, s'il n'est pas encore totalement

homogène faute d'Une Métaphore Parfaite qui en unifierait les différents aspects, permet néanmoins de dégager une vision qui soit commune et de la concrétiser sans efforts dans le code. En conséquence, l'implémentation de nouvelles fonctionnalités devient plus efficace ; avant même la fin de l'itération, il devient évident que la vélocité de l'équipe sera en définitive en nette hausse.

Seconds mandats

Sur le plan commercial, la formule contractuelle mise au point par Alex est un succès. Il est vrai que cette appréciation tient en grande partie à ce que le projet lui-même se solde par un bilan largement positif ; personne ne souhaite trop spéculer sur ce qui se serait passé si Sonata avait choisi de faire jouer l'une des clauses de désengagement anticipé prévues par le contrat, permettant essentiellement au client d'interrompre le contrat à la fin de chaque itération en conservant l'existant à ce jour.

Selon les modalités prévues par cette formule adaptée au déroulement d'un projet XP, David et Alex décident de « solder » les itérations précédentes – ce qui revient de la part de David à confirmer la recette des fonctionnalités livrées jusqu'à présent et en régler la facture – et de reconduire le contrat pour deux itérations supplémentaires assorties de clauses de désengagement plus favorables à Advent, en gage de confiance.

Sébastien et Leslie d'un côté, Alex de l'autre, « soldent » leurs divergences au cours d'une conversation quelque peu tendue. Des questions déjà débattues sur la « visibilité » fournie par le projet reviennent sur le tapis, Alex faisant valoir que des « innovations » comme la programmation en binôme ne sont pas vues du meilleur œil par la direction du groupe ; il menace de retirer la responsabilité du projet à Leslie si elle ne « fait pas un effort » pour imposer un suivi individuel plus précis aux développeurs, un effort de documentation non comptabilisé comme « scénarios client », entre autres demandes.

Mais ces exigences et ces menaces cessent d'elles-mêmes, Alex finissant par reconnaître qu'elles ne sont motivées que par un sentiment diffus qui tient à ce qu'il ne contrôle pas une équipe qui, au cours des semaines, a réellement pris corps. S'appuyant sur des résultats objectivement excellents, Leslie et Sébastien obtiennent en définitive qu'on reconnaisse comme principal facteur de succès du projet la cohésion de cette équipe et qu'elle puisse à la fois conserver son identité au cours de projets futurs, et propager dans le reste du groupe aussi bien cette culture que la démarche XP.

Leslie se verra affecter à un autre projet, et une autre équipe, à l'issue de la prochaine itération, avec pour mission de montrer que de tels résultats sont reproductibles ; la stabilité du reste de l'équipe étant garantie pour au moins un projet ultérieur.

Nouveaux horizons

Le bilan d'itération, au cours duquel ces diverses décisions sont annoncées, s'est transformé au fil du temps en un cocktail impromptu, une occasion de réfléchir et de discuter mais également de faire la fête avant le week-end. Les développeurs ont désormais confiance en leur propre maîtrise de la démarche et en leur capacité à travailler ensemble ; leurs regards critiques et leurs idées d'innovations portent essentiellement sur des aspects qui vont au-delà du projet en cours.

C'est ainsi que l'équipe décide – peut-être pour retrouver un peu de l'esprit «cow-boy» qui anime plus souvent les projets non XP, en contraste avec le sentiment général d'être devenus une «machine à implémenter des scénarios client» – de s'attribuer un petit budget temps au sein de chaque itération pour réaliser, collectivement ou individuellement, des expérimentations personnelles – avec ou sans tests, avec ou sans refactoring, bref de façon totalement libre pourvu qu'aucun code issu de ces recherches ne soit versé au référentiel du projet.

On réfléchit aussi à la constitution d'une bibliothèque commune, d'un abonnement à quelques magazines spécialisés, de formations techniques auxquelles l'équipe, ou ses membres individuellement, pourraient participer pour développer leurs compétences.

Mais l'heure n'est bientôt plus à la réflexion – le week-end approche, le soleil est chaud et les bières fraîches, et on se met à rire et à parler de choses moins sérieuses. L'histoire n'est pas finie pour nos «héros» mais nous ne les suivrons pas plus loin : la notion XP de rythme durable implique qu'une équipe XP efficace peut, une fois constituée, continuer indéfiniment.

12

Un projet industriel en XP

Rien de ce qu'il faut savoir ne peut être enseigné.

– Oscar Wilde

XP est une méthode volontairement iconoclaste sur certains sujets tels que la documentation, le cloisonnement et la hiérarchisation des programmeurs… mais il serait faux de penser que cette méthode est réservée aux seules start-ups ou aux PME qui ne posséderaient aucune culture méthodologique préalable. Tant par la discipline qui sous-tend la méthode que par la qualité du produit, de la planification et du «reporting» dont elle peut se prévaloir, ou encore par l'excellente gestion des ressources humaines qu'elle permet, l'Extreme Programming a de quoi séduire les grands groupes industriels.

Ce chapitre décrit une expérience réelle, le premier projet en France à mettre en œuvre XP de manière quasi complète dans un contexte industriel.

Le contexte

Ce retour d'expérience provient d'une entreprise française de taille moyenne (600 salariés dont environ 150 développeurs) qui réalise des produits et des systèmes dans le domaine des transports. Deux types de logiciels y sont développés : des logiciels temps-réel dur, souvent critiques pour la sécurité des personnes et des biens, parfois embarqués, et des logiciels de supervision (typiquement des postes de commande centralisés), pseudo-temps réel, et «seulement» critiques pour la réussite des missions.

Certifiée ISO9001 depuis 1994, l'entreprise utilise le cycle en V qui est le standard maison. Pourtant, depuis 1997, les projets de supervision sont passés à la programmation orientée

objet, et il y règne une certaine velléité à travailler de manière plus itérative, sans que cela se concrétise officiellement.

C'est à cette époque qu'un ingénieur logiciel de la société découvre l'Extreme Programming au travers des discussions sur le newsgroup *comp.object* et sur le Wiki Wiki Web de Ward Cunningham. C'est le coup de foudre. Avec assez d'expérience en tant que chef de projet pour avoir connu les problématiques de planning, pilotage et gestion d'équipes, mais pas trop pour avoir oublié ce qu'était la vie de programmeur, cet ingénieur trouve que la méthode est idéale des deux points de vue. Pourtant, il a aussi participé à la réussite de projets en pratiquant le cycle en V et une gestion de projet plus traditionnelle, mais XP lui donne l'impression d'un bond qualitatif.

Pour s'en faire une meilleure idée, il commence à apprendre et utiliser certaines pratiques, compatibles avec le projet sur lequel il travaille actuellement, en particulier les tests unitaires et la programmation en binôme. Cela ne fait que renforcer ses premières impressions et il continue à se familiariser avec la méthode.

L'opportunité

Fin 1999, on lui confie un nouveau projet pour un système de supervision. Il s'agit d'un contrat au forfait, dont la partie logicielle placée sous sa responsabilité représente 95 hommes-mois. C'est un système temps réel distribué comprenant un poste de commande centralisé, composé d'un serveur avec redondance à chaud[1], de cinq postes opérateurs et de douze postes distants, chacun responsable d'une zone sur le terrain. Ces différents calculateurs effectuent des acquisitions de données et des envois de commandes sur plusieurs équipements. Le système permet aux opérateurs du poste centralisé ou des postes distants de contrôler et commander les véhicules et les équipements, mais effectue aussi des traitements automatiques critiques à l'exploitation (calculs de parcours, informations voyageurs, etc.) Le tout doit fonctionner 24 heures sur 24, 7 jours sur 7, avec en moyenne 100 000 heures entre deux pannes totales du système.

Les délais annoncés et le périmètre prévu indiquent que l'équipe de développement devrait comporter entre six et huit développeurs. Une première analyse révèle qu'il s'agira d'un développement nouveau : il n'existe pas de projet analogue mettant en œuvre les technologies actuelles (C++, bibliothèques graphiques et middleware orientés objet). En outre, il n'y a à ce moment que deux développeurs disponibles ; il faudra donc compléter l'équipe en recrutant au moins quatre ingénieurs.

Le responsable logiciel pense que le projet est de nature à permettre d'essayer en «grandeur nature» la méthode XP. Encore faut-il en convaincre, en l'occurrence, la chef de projet, les directeurs et les développeurs eux-mêmes…

1. En anglais *hot-swap* – deux versions du logiciel tournent en permanence et l'une peut prendre le pas sur l'autre à tout moment sans opération préalable.

Susciter l'adhésion

La chef de projet, première consultée, perçoit rapidement les atouts que pourrait présenter XP, mais souligne les risques que présente la mise en œuvre d'une méthode nouvelle pour tout le monde. Par chance, elle a l'esprit ouvert et souhaite déléguer complètement aux responsables des différentes parties du projet, logicielles et matérielles, en leur laissant un maximum d'initiative. Elle donne son accord de principe et pose pour seule condition l'obtention de l'accord de toutes les parties concernées.

Le futur «coach XP» fait donc une présentation de la méthode XP au management (direction opérationnelle, direction qualité). L'accent est mis sur le contexte favorable (taille de l'équipe, délais courts, spécifications changeantes), et les atouts suivants sont soulignés :

- le cycle incrémental et itératif et ses principales conséquences : gestion des risques, réalisme des indicateurs d'avancement ;
- le travail en binôme qui constitue une «relecture de code» permanente ;
- la responsabilité collective du code et les autres pratiques collaboratives, qui réduisent les risques liés au turn-over et les goulets d'étranglement autour des individus qui sont supposés indispensables ;
- la forte culture de tests.

Les objectifs visés (réduction des coûts, amélioration de la qualité et gestion des risques) intéressent logiquement les décideurs, et les techniques mises en avant apparaissent toutes plus ou moins comme des solutions aux problèmes constatés sur tel ou tel projet récent. Le management autorise l'utilisation de la méthode XP, et jouera par la suite son rôle en exigeant tout au long du projet que la démonstration soit faite que la méthode tient ses promesses.

Le coach réunit alors ceux des développeurs qui sont déjà présents pour leur présenter plus en détail la méthode : les quatre valeurs et les pratiques canoniques. L'accueil est plutôt positif, bien que légèrement mitigé : une curiosité et une certaine attirance doublée d'un léger scepticisme. Seul Florent, un développeur dont l'expérience et les compétences sont d'ailleurs assez analogues à celles du coach, accueille tout cela avec enthousiasme, avec l'impression que toutes les pièces d'un puzzle sont enfin et soudain réunies.

Premiers pas

Au moment où la décision d'utiliser XP est prise et que les programmeurs constituent l'équipe, une première version des spécifications a déjà été rédigée par le coach sous forme de cas d'utilisation, utilisant le formalisme recommandé par Alistair Cockburn[1]. Ces cas d'utili-

1. *Writing Effective Use Cases*, Alistair COCKBURN, Addison-Wesley 2000 (traduit aux Editions Eyrolles sous le titre *Rédiger des cas d'utilisation efficaces*, 2001)

sation (80 environ) serviront de scénarios client. Mais avec des scénarios déjà prêts, une phase d'exploration ne présenterait qu'un intérêt technique et risquerait fort de déraper vers une phase de mise en place d'architecture. Le coach décide donc de procéder directement à une itération ayant pour but de livrer un ensemble de scénarios liés. Son choix s'est naturellement porté sur un ensemble cohérent fournissant une valeur métier tout à fait fondamentale, dont dépend le reste de l'application. Par bonheur, la solution technique, bien que complexe d'un point de vue algorithmique, est simple sur le plan technologique et peut être réalisée avec un minimum d'architecture et de la programmation en C++ pur.

Remarque

Au début, le coach joue donc lui-même le rôle de client (écriture des scénarios client, choix des priorités). Par la suite, il donnera libre choix au chef de projet et au responsable technique de projet pour fixer les priorités mais, compte tenu de son implication initiale dans la spécification des besoins, il restera vis-à-vis de l'équipe de programmeurs l'interlocuteur principal pour préciser les besoins. Il s'agit évidemment d'une « adaptation locale » des recommandations théoriques d'XP, mais bien comprise comme telle et dont les risques sont considérés comme maîtrisés.

Première itération

La durée des itérations est fixée à un mois. Pendant les deux premières semaines, l'équipe se réunit au moins une fois par jour, tantôt pour discuter de la conception, tantôt pour que soient établies en commun des règles de codage, ou encore pour assister à des topos du coach sur la méthode XP ou sur la conception objet. Outre cette forme de conception réalisée de manière collective et au fur et à mesure que les besoins quotidiens se présentent, les programmeurs se consacrent à l'apprentissage des deux pratiques fondamentales pour leur activité : la programmation en binôme et les tests unitaires.

Aménagement du bureau

Le coach avait beaucoup insisté en début de projet pour réunir dans un seul bureau toute l'équipe de programmeurs (désormais au nombre de cinq, outre le coach), et à la faveur de la réorganisation que cela demandait, il avait fait aménager le bureau à la façon XP : trois postes de développement regroupés sur une grande table centrale, avec deux chaises par poste ; une table de réunion centrale ; tout autour sont disposés des bureaux individuels, avec un poste bureautique pour deux bureaux. Autour de la table centrale de développement, les développeurs peuvent se voir et se parler sans se lever. Les bureaux individuels périphériques sont aussi orientés par rapport au centre, de façon que tout le monde puisse continuellement observer ce qui se passe, écouter et participer. Un mois plus tard, l'équipe est complétée par deux programmeurs supplémentaires, soit un poste de développement en plus et deux bureaux

individuels. À côté des postes de développement se trouve un poste d'intégration. La figure 12-1 illustre l'aménagement final du bureau.

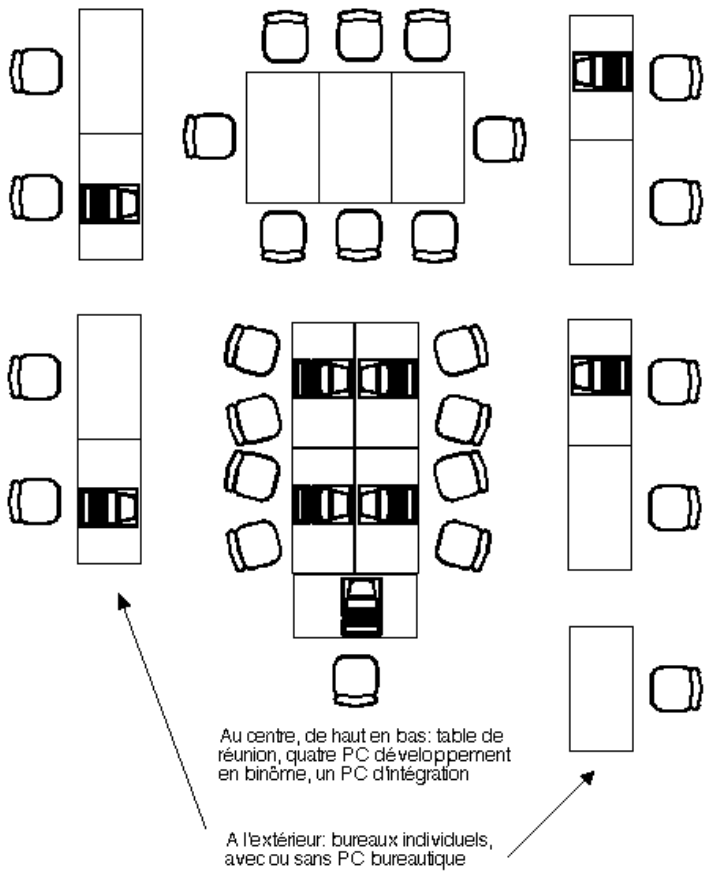

Figure 12-1. Aménagement XP du bureau

Premières impressions

Très rapidement, les programmeurs sont globalement satisfaits de ce qu'ils découvrent, et chacun s'enthousiasme d'un aspect particulier de la méthode : la programmation en binôme, les tests unitaires, le travail collectif, ou encore la communication permanente favorisée par la disposition des bureaux.

Témoignage de Michaël, développeur

Au tout début de la mise en pratique de la méthode XP, j'avais une certaine appréhension concernant notamment des aspects qui m'étaient jusque-là plutôt inconnus. Par exemple, mes expériences précédentes en développement logiciel ne m'avaient pas permis d'appréhender la planification d'un projet.

Grâce aux séances collectives de planification, au travail en binôme et aux réunions quotidiennes de conception, volontairement courtes, mes connaissances techniques ont très vite augmenté, me procurant ainsi une très grande satisfaction intellectuelle. La participation active de tous les développeurs lors des séances de planification a permis de confronter des points de vue, tous très intéressants car empreints d'expériences différentes, contribuant ainsi à une bonne définition des tâches et aussi à une connaissance très rapide de chacun des membres de l'équipe.

La cohésion de l'équipe et les connaissances de chaque personne augmentant avec le temps, chacun s'est enrichi intellectuellement, ce qui a eu pour effet de créer une très forte motivation dans le travail effectué.

En effet, les programmeurs moins expérimentés sont ravis d'être impliqués dans la conception et la planification. Florent, tout en appréciant aussi ce travail collectif, est tellement accoutumé à ce qu'on lui confie l'architecture de ce genre de projet qu'il doit se surveiller pour ne pas reprendre ses habitudes antérieures : prise en compte des fonctionnalités futures, mise en place de frameworks. Heureusement, il lui suffit de se rappeler qu'il ne travaille plus seul désormais…

Sur le plan technique, il n'y a aucune difficulté à mettre en place l'outil de tests unitaires en C++, CppUnit, et l'outil de gestion de configurations, CVS. Les méthodes associées, la programmation pilotée par les tests et l'intégration continue sont de nouvelles habitudes qui sont rapides à assimiler.

Corriger le tir

Lors de cette première itération, deux difficultés sont pourtant constatées.

D'abord, un membre de l'équipe, Jean-Baptiste, arrive sur le projet avec le seul module issu d'un projet précédent qui doit en principe être réutilisé : un framework de redondance à chaud. Le module doit être «porté» ou plutôt «importé» dans ce nouveau projet, et l'estimation de ce travail a été sous-estimée. Comme Jean-Baptiste a réalisé ce module, il travaille hélas tout seul à ce portage… et, plus le temps passe, plus il insiste pour rester seul, convaincu que la nécessité d'expliquer ce qu'il fait à une autre personne va le retarder encore plus.

Rétrospectivement, sachant qu'il lui a fallu l'itération entière pour terminer ce travail et qu'il aurait suffi de quelques jours à une autre personne pour se familiariser avec le module, il prend conscience d'avoir mis plus de temps que nécessaire à la réalisation de cette tâche ; d'avoir perdu l'opportunité de ne plus être le seul détenteur de la connaissance de ce module ; et

surtout de s'être mis un peu à l'écart des développements réalisés par le reste de l'équipe, même s'il continuait à participer aux séances collectives. Ironiquement, Jean-Baptiste est ensuite devenu l'un des plus ardents défenseurs de la programmation en binôme.

Témoignage de Jean-Baptiste, développeur

Le point XP qui me parut être le plus révolutionnaire fut la programmation en binôme. En partie parce que tous les autres points découlent beaucoup plus du bon sens ; se donner les moyens de les appliquer lorsqu'on les a identifiés devient vite une évidence.

Mais le binôme... Pas facile d'en percevoir de manière certaine l'apport de productivité. Encore aujourd'hui, bien qu'entièrement convaincu, je dois me surveiller pour pratiquer constamment la programmation par paire. Parce que c'est tentant, pour des petites modifications, l'ajout d'une fonctionnalité facile, de faire tout seul dans son coin un petit bout de code. Cependant, l'expérience et la comparaison m'ont démontré que, même pour un travail simple, une réflexion préliminaire à deux est profitable pour développer de manière plus intelligente et que le codage à deux évite des erreurs stupides toujours coûteuses en temps.

Lorsque la solution est plus délicate à développer, la confrontation des idées, le besoin constant de clarifier pour se faire comprendre, le recul du second quand le premier est au clavier, rendent la pratique de la programmation par paire une notion dont il sera difficile par la suite de se passer.

Quand bien même on ne serait pas persuadé du gain immédiat de productivité, les aspects à plus long terme peuvent justifier cette pratique : la programmation par binôme entraîne automatiquement une connaissance collective de la totalité du projet dont il résulte :

– la responsabilité collective du code ;

– des séances de planification où chacun a son mot à dire ;

– et surtout, une application simplifiée de l'adage « nul n'est irremplaçable dans son travail ». Finies les perturbations dues aux vacances, aux arrêt maladies, aux changements de poste et aux démissions.

Le binômage est vraiment une pratique qui gagne à être essayée, mais qu'il faut pratiquer assidûment pour s'en convaincre définitivement.

Le second problème constaté à la suite de cette première itération est que l'équipe n'a pas terminé les fonctionnalités planifiées. Il faudra finalement le double du temps prévu (une deuxième itération). La vélocité constatée est donc de moitié par rapport à celle qui avait été initialement supposée – ce qui constitue en soi une information utile.

Encore faut-il relativiser la gravité de ces constats, car le plus important est là : un véritable esprit d'équipe s'instaure, les progrès des programmeurs moins expérimentés sont immédiats, et l'un des problèmes de programmation les plus difficiles que l'équipe aura à résoudre a été éclairci, même s'il reste du travail. En outre, l'équipe peut tirer parti de cette expérience pour réviser ses estimations pour la suite du projet.

Montée en charge

Ces estimations révisées confirment d'ailleurs le scénario qui avait été initialement envisagé, à savoir qu'il faut compléter l'équipe avec deux développeurs supplémentaires. C'est d'autant plus vrai que (comme pour prouver que la responsabilité collective du code fait ses preuves), Fabrice se casse une jambe au ski et Jean-Baptiste tombe sérieusement malade : ils cumulent à eux deux près de 3 mois d'absence… En outre, l'un des premiers programmeurs à arriver sur le projet, Michel, est sans cesse et inopinément rappelé sur un ancien projet pour des périodes de quelques jours. Il apparaît rapidement que ce genre de présence en pointillé est incompatible avec la méthode XP. L'équipe doit être entièrement dédiée au projet. Le coach ne parvient pas à obtenir l'assurance que ce sera le cas pour Michel, et décide à regret qu'il vaut mieux qu'il quitte l'équipe. Michel est pourtant un « vétéran » de la société, et de loin le plus expérimenté dans le domaine métier, mais le bon fonctionnement de l'équipe passe en l'occurrence avant les qualités personnelles.

Il est intéressant de noter qu'un programmeur de plus de cinquante ans, aux habitudes bien ancrées et assez « forte tête », a apprécié cette nouvelle méthode, notamment la programmation dirigée par les tests dont il a immédiatement perçu l'intérêt.

S'il fallait recommencer

Rétrospectivement, compte tenu de ses compétences métier doublées de compétences en informatique, il apparaît que Michel aurait dû jouer le rôle de client et/ou testeur pour cette équipe. Ses absences occasionnelles auraient probablement été plus tolérables. Cette absence des rôles de client et testeur (du moins au sens XP) sera d'ailleurs une des plus grosses erreurs dont sera entaché le projet. Mais nous anticipons…

François, tout droit sorti de l'école, et Franck, déjà expérimenté, arrivent peu après dans l'équipe. Leur intégration se passe extrêmement bien (et rapidement). Au lieu de leur présenter une pile de documents à lire la première semaine, le coach fait présenter le projet oralement en 30 minutes par l'équipe au complet, et ils se mettent immédiatement au travail en binômant avec un développeur sur une tâche en cours. Franck est opérationnel en quelques heures, et François, qui a pourtant beaucoup à apprendre, en quelques jours.

Vitesse de croisière

Au terme de deux ou trois itérations, l'équipe est familiarisée avec l'ensemble des pratiques qu'elle va utiliser pendant le restant du projet. Un plan de livraison a été défini avec la chef de projet et le responsable technique. La première livraison, destinée à une équipe de validation indépendante, n'est prévue qu'au bout de six mois. Le coach insiste sur la nécessité d'effectuer cette validation en parallèle, au fur et à mesure de la progression de l'équipe logicielle, et sans attendre six mois. Mais les personnes responsables de cette validation ne peuvent être libérées à temps…

Le projet utilise des itérations d'un mois. Au début de chaque itération, une journée entière est consacrée à la (re)lecture des scénarios, à de la conception générale permettant de définir les tâches et les estimer. L'équipe se prête à un petit jeu permettant d'atteindre des estimations à la fois fiables et les plus fines possible. Une fois l'ensemble des tâches définies, deux groupes sont formés en cherchant à équilibrer les expériences respectives. Chaque groupe évalue d'abord indépendamment toutes les tâches, avant de comparer les résultats. Lorsqu'il y a une différence entre les estimations, un débat s'ensuit jusqu'à ce qu'un consensus soit atteint. Parfois, l'équipe ayant fait l'estimation la plus faible n'a pas pensé à une difficulté particulière relative à la tâche en question. D'autres fois, l'estimation plus faible s'explique par une idée permettant de faire le travail plus simplement… En cas de divergence inexplicable, l'estimation la plus faible est retenue. Le futur responsable de cette tâche pourra toujours demander à l'une des personnes qui a fait l'estimation optimiste de binômer avec lui !

Ces séances de planification et de conception, tout en étant très intensives et productives, se déroulent dans la bonne humeur. Les éclats de rire peuvent même perturber les équipes voisines, qui ne sont pas habituées à une telle ambiance.

Synthèse des pratiques et des non-pratiques

Le tableau 12-1 présente le positionnement de cette équipe par rapport aux 13 pratiques canoniques d'XP.

Client sur site	Il s'agit d'un développement logiciel au sein d'un développement système (comportant de la conception matérielle, et un logiciel embarqué, réalisé par une autre équipe). En tant que tel, le «client» de l'équipe logicielle est le responsable technique de projet. Ce dernier est effectivement proche des programmeurs (d'abord quelques bureaux plus loin, avant de s'installer dans le même bureau) ; il définit les priorités dans la planification des fonctionnalités. Hélas, pour l'essentiel parce que c'est le coach qui a rédigé les spécifications logicielles, l'équipe ne prend que très tard dans le projet l'habitude de consulter directement le responsable technique pour avoir des précisions fonctionnelles.
Planification itérative	Des cas d'utilisations avaient été écrits par le coach en amont du projet. Ils sont utilisés comme scénarios client. Un plan de livraison, établi par le chef de projet en concertation avec le coach, alloue les scénarios clients aux différentes itérations. Les itérations durent 1 mois et commencent par une séance d'une journée pendant laquelle les programmeurs et le coach décomposent le travail en tâches dont ils font l'estimation. Le plan de livraison est parfois revu en conséquence.
Conception simple	C'est une volonté bien présente. Par exemple, le projet marque une rupture avec les précédents en abandonnant un certain nombre d'outils et technologies au profit d'une «simple» programmation en C++. Les plus expérimentés sont parfois tentés de mettre en place des «frameworks» en prévision des besoins ultérieurs. Tout en acceptant d'en parler et d'imaginer des solutions, le coach propose d'en remettre à plus tard la mise en œuvre. Spontanément, les programmeurs résolvent donc leurs problèmes immédiats avec des solutions «provisoires» plus simples, qui deviendront bien souvent permanentes.

Tests unitaires	Le harnais de tests CppUnit est déployé avant que les premières lignes de code ne soient écrites, et l'écriture des tests unitaires avant l'écriture du code est la première technique XP que le coach présente. Elle est rapidement maîtrisée par les programmeurs, à la fois comme outil de conception détaillée (ils ne ressentent plus le besoin de rentrer à ce niveau de détail lors des réunions collectives de conception) et comme outil de non-régression. Plus tard, lorsque la validation du logiciel aura commencé, chaque défaut détecté sera d'abord reproduit par un test unitaire, avant même d'être corrigé.
Remaniement	Pendant plusieurs mois, le développement consiste plutôt à ajouter du code supplémentaire. Petit à petit, pourtant, de nouveaux besoins rendent inappropriés certains choix de conception initiaux. Dans un premier temps, les programmeurs pratiquent le remaniement au sens où ils s'autorisent à changer la conception existante, grâce à la présence des tests unitaires. Plus tard, le coach fait l'acquisition du livre de Martin Fowler et les programmeurs apprennent à travailler progressivement en gardant une maîtrise permanente du code remanié, et le «catalogue» des remaniements entre peu à peu dans leur vocabulaire.
Intégration continue	Dès le début, les programmeurs travaillent sur la totalité du code source et intègrent presque quotidiennement. Il n'y a jamais de phase d'intégration explicite.
Tests de recette	La validation du logiciel suit plutôt le cycle en V et ne commence qu'après 8 ou 9 mois de développement. En outre, les tests de recette ne sont pas tous automatisés, ni autovérifiants. L'interface graphique n'est testée que manuellement. Lorsque c'est possible, les programmeurs écrivent des tests «unitaires» qui sont presque fonctionnels, car ils testent des objets de très haut niveau qui représentent ce que fait une application complète.
Programmation en binôme	Les programmeurs travaillent à deux dès le début du projet. Plusieurs techniques de rotation sont essayées (binômes fixés pour une tâche, pour une itération, pour une demi-journée…). Il n'y a presque aucune exception à cette règle, et ce n'est guère que les jours où les programmeurs sont en nombre impair que l'un d'entre eux se retrouve seul. La seule dérive constatée est que Franck se retrouve seul plus souvent que les autres, car il a tendance à travailler sur des problèmes complexes d'architecture distribuée qu'il est seul à affectionner…
Règles de codage	Dès le début du développement, le coach propose des règles de codage. L'équipe lui fait confiance pour ce qui est de fixer les règles importantes pour la qualité du code C++, mais les conventions (nommage, mise en page…) le sont par consensus. Chaque fois qu'un développeur se trouve face à un choix nécessitant une convention, il en parle à l'équipe, et une nouvelle règle est définie. Au bout de quelques mois, l'équipe dispose d'un jeu complet et cohérent de règles C++, à tel point qu'il sert de base pour la définition d'un guide C++ pour l'entreprise entière. Bien que 60 développeurs soient concernés par ce guide, il est élaboré (en plusieurs mois) dans la concertation.
Métaphore	Les développeurs prennent l'habitude d'utiliser des métaphores pour décrire certaines parties de la conception (métaphores locales) mais, malgré plusieurs *brainstormings*, il semble difficile de trouver une métaphore générale.
Responsabilité collective du code	Cette pratique est complètement respectée : chaque binôme est libre de modifier toute partie du code qu'il souhaite. Les développeurs s'efforcent, au gré des tâches et itérations, de travailler sur toutes les parties du système.
Livraisons fréquentes	Compte tenu du décalage entre les développeurs et l'équipe indépendante de validation, la première livraison ne se fait pas avant huit ou neuf mois. Ensuite, une livraison est effectuée lors de chaque itération (3 ou 4 semaines).
Rythme durable	L'ensemble de l'équipe pratique une semaine de 39 heures pendant la durée du projet.

Reporting

Dans un premier temps, la gestion du projet déroute quelque peu les responsables (chef de projet, chef de service, directeurs). Le coach doit énormément communiquer pour en expliquer le fonctionnement (définition et attribution des tâches), les principes de planification (livraisons, itérations, scénarios client) et les indicateurs. À ce moment, la vélocité n'est pas encore la norme en XP, c'est son prédécesseur le *Load Factor*, qui est utilisé. Les développeurs estiment les tâches comme les scénarios client en jours idéaux[1], ensuite ils suivent le temps de travail et le temps absolu (calendaire) réellement écoulés pour chaque tâche. On obtient ainsi deux mesures : en premier lieu, la différence (en jours idéaux) entre l'estimation et ce qui est effectué, baptisée glissement technique ; en second lieu, le coefficient, appelé *Load Factor*, qui permet d'obtenir les jours réels à partir des jours idéaux effectués. Le glissement technique indique donc les erreurs d'estimation ou les difficultés imprévues. Le *Load Factor* représente l'efficacité de l'équipe (la proportion du temps passé à développer).

Les questionnements du management sont très bénéfiques et, petit à petit, le coach, avec un développeur désigné pour jouer le rôle de tracker, met en place un tableau de reporting fourni à chaque itération au management. Y figurent :

- le nombre total de scénarios client et le nombre de ceux qui sont terminés[2] ;
- le *load factor* de la dernière itération et le *load factor* moyen sur l'ensemble du projet ;
- le glissement technique (nombre de jours réels supplémentaires requis pour finir les tâches de la dernière itération, par rapport aux estimations ; autrement dit, quantité de travail prévu non terminé) ;
- la liste des scénarios client prévus pour la prochaine itération.

L'estimation en jours idéaux se révèle utile pour les développeurs, leur permettant de fournir des évaluations honnêtes, sans devoir compenser pour les perturbations, généralement indépendantes de leur volonté (sollicitations diverses) ou requises par la méthode XP (travail en binôme sur la tâche d'un autre programmeur, réunions d'équipe fréquentes…), qui sont prises en compte automatiquement grâce au *load factor*. Il y a certes une grande part de subjectivité entre ce qui est attribué à un glissement technique ou «passé» en *load factor* mais, comme les estimations se font en équipe et que le suivi des tâches fait aussi l'objet de discussions, l'équipe adopte naturellement des conventions pour différencier ce qui doit être comptabilisé comme glissement technique et ce qui relève plutôt du *load factor*. Cela permet de rendre les estimations de plus en plus fiables et précises.

1. Pour procéder à cette estimation en jours idéaux, un programmeur doit considérer qu'il travaille dans de bonnes conditions, avec le binôme de son choix, sans être lui-même sollicité par ailleurs (binômes ou réunions), et sans autres perturbations (environnement, matériel…).
2. En l'absence de tests de recette automatiques qui seraient disponibles au fur et à mesure du développement, le critère de complétion d'un scénario client devient hélas non pas sa recette, mais le fait que les développeurs estiment l'avoir terminé.

Ralentissement

La motivation des développeurs, leur maîtrise grandissante des techniques et pratiques XP ainsi que leurs progrès constants dans l'art de programmer et de concevoir font que le développement avance vite, très vite. Compte tenu de la progression constatée après quatre itérations, les estimations à long terme révisées par l'équipe permettent au coach de prendre la décision de ne pas monter encore en charge (alors que le budget initial et le plan de charge correspondant le prévoyaient).

Le code grossit donc rapidement et, à partir du mois de juin, après cinq itérations, les indicateurs, en particulier le *load factor*, témoignent d'un ralentissement. Quelques signes d'exaspération ponctuent parfois l'ambiance qui reste toutefois très bonne. Les pauses café deviennent plus fréquentes.

La cause de ce changement de rythme est rapidement montrée du doigt : le temps nécessaire à la génération du programme qui passe tous les tests unitaires a atteint 22 minutes ! Or, les méthodes de programmation en XP, que ce soit en développement ou en remaniement, impliquent de regénérer et relancer ce programme très fréquemment, parfois toutes les cinq minutes !

Lorsque le temps *d'exécution* des tests unitaires devient trop long, les programmeurs peuvent faire quelque chose (simplifier certains tests, optimiser le code…). En revanche, lorsqu'il s'agit du temps de *génération* de ces tests (compilation + édition des liens), la seule chose à faire est de réduire les dépendances entre classes et modules, dans l'espoir qu'une petite modification n'entraînera pas la recompilation de trop de fichiers.

Après avoir amélioré les choses de ce côté-là, les programmeurs font état que, comme l'outil de compilation commet parfois des erreurs dans ses calculs de dépendances, ils ont en fait tendance à tout supprimer et tout recompiler systématiquement.

Le coach propose de changer d'outil. Cela nécessite une semaine de travail pour un binôme (ou souvent un programmeur, ce travail étant souvent assez systématique), mais l'équipe ne regrette pas l'investissement : en recompilant juste ce qui est nécessaire, l'outil leur fait déjà gagner du temps.

Toutefois, le temps d'édition des liens, pour sa part, ne peut être diminué, et il est lui-même très élevé (plus de dix minutes). Comme l'exaspération commence à gagner, plusieurs programmeurs cherchent diverses solutions et tentent de diagnostiquer les problèmes. Il apparaît qu'en augmentant considérablement la mémoire de façon à dépasser la mémoire utilisée par le processus d'édition des liens, ce dernier cesse d'écrire sur le disque dur : le temps nécessaire est radicalement diminué.

Malheureusement, les habitudes de travail étant assez différentes dans l'entreprise, le coach doit lourdement insister pour obtenir cette mémoire supplémentaire. Une ironie, compte tenu des coûts relatifs de la mémoire vive par rapport à une équipe de programmeurs… mais c'est représentatif des difficultés qui relèvent de la culture d'entreprise auxquelles peut se trouver confrontée une première mise en œuvre de la méthode XP.

Mais on dispose enfin de la mémoire nécessaire, et l'équipe retrouve rapidement son efficacité et sa bonne humeur ; en remaniant un peu le code, en déployant un nouvel outil de génération et en augmentant la mémoire des machines, elle a réduit le temps de génération de l'exécutable de test à guère plus de deux minutes !

Désillusions

Environ huit mois après le début du développement, l'équipe qui doit faire la validation fonctionnelle du système (comprenant le logiciel développé par l'équipe XP, du matériel et un logiciel embarqué développé par une autre équipe) commence à travailler. Très vite, des défauts sont mis en évidence dans des fonctions que les développeurs pensaient avoir terminé et testé. Pour l'essentiel, deux sortes d'erreurs sont rencontrées :

- Les traitements qui correspondent à ce qu'avaient compris les développeurs, mais pas à ce qui était attendu par l'équipe système.
- Les défauts qui n'étaient pas révélés par les tests unitaires des développeurs, soit parce qu'ils n'étaient pas exhaustifs, soit parce que l'environnement d'exécution n'est pas le même (plate-forme de développement plutôt que plate-forme cible).

Évidemment, ces problèmes auraient été détectés immédiatement si les tests fonctionnels avaient pu être effectués en parallèle des développements. Le décalage de huit mois pose trois problèmes majeurs :

- Les développeurs passent beaucoup plus de temps à corriger un défaut sur des fonctions développées huit mois plus tôt qu'un défaut identifié dans les développements en cours.
- Les erreurs de compréhension ont parfois un impact qui s'est répercuté sur tous les développements effectués pendant ces huit mois.
- Pour mesurer l'avancement, l'équipe a été forcée de se baser sur le nombre de scénarios client «terminés» *selon les développeurs*. Les estimations de charges et de délais correspondantes sont aussi basées là-dessus. Une certaine charge a bien été prévue pour traiter les défauts détectés lors de la validation, mais cette charge a été évaluée «à la louche» et n'a jamais pu être vérifiée auparavant. Compte tenu de la lourdeur de certaines corrections (pour les deux raisons précédentes), les nouvelles évaluations sont assez alarmantes, ce qui ne fera que se confirmer avec le temps.

Fière de son travail, soucieuse aussi de démontrer l'efficacité de la méthode XP au monde extérieur, l'équipe vit difficilement cette période de désillusions. Toutefois, la manière dont l'équipe gère cette situation apporte deux satisfactions. D'abord, dès qu'un défaut est découvert par l'équipe de validation, les développeurs s'efforcent en premier lieu de le reproduire à l'aide d'un test unitaire. C'est une forme de diagnostic. Parfois, le fait de ne pas pouvoir le reproduire permet de découvrir qu'il ne s'agit pas d'un problème purement logiciel, mais d'un problème d'installation ou de paramétrage sur la plate-forme cible… Une fois le défaut reproduit par un test unitaire, la correction devient souvent triviale, ou au moins peut s'effectuer

dans un contexte bien maîtrisé – sachant de plus que le défaut ainsi circonscrit ne pourra réapparaître sans être immédiatement détecté. En second lieu, l'équipe de développement est très souple et réorganise fréquemment les priorités. Ces deux facteurs permettent à l'équipe d'être très réactive et d'absorber assez aisément les changements induits par ces problèmes.

L'audit ISO9001

L'entreprise où se déroule le projet est certifiée ISO9001 ; par conséquent, entre les missions de certification ou de renouvellement effectuées par des auditeurs tiers, des audits internes ont régulièrement lieu. Le projet comprenant cette équipe XP est désigné pour un audit interne. Bien que les auditeurs soient membres du personnel de l'entreprise, ils sont formés pour effectuer de véritables audits et ne font preuve d'aucune complaisance, même si l'objectif de l'audit est d'être constructif.

Après un entretien poussé avec la chef de projet, les deux auditeurs rencontrent le coach, accompagné de Jean-Baptiste, le développeur jouant aussi le rôle de tracker. D'emblée, ils semblent s'intéresser à la méthode XP («ça fait tellement longtemps qu'on parle de méthodes itératives sans en pratiquer réellement»). Bien que certaines pratiques soient à première vue très éloignées des méthodes généralement utilisées, ils en comprennent facilement le principe grâce aux explications du coach.

Les auditeurs relèvent plusieurs points forts :

- une très bonne traçabilité des activités de développement, au sens où les scénarios client conduisent à des tests unitaires qui conduisent à leur tour au code applicatif ;
- la qualité de la communication au sein de l'équipe ;
- la planification itérative, notamment la qualité et la finesse des estimations de charges des tâches, ainsi que le suivi de ces dernières ;
- l'intégration continue ;
- la qualité du reporting et des indicateurs.

En revanche, ils relèvent trois anomalies :

- L'accord de la direction qualité pour pratiquer une méthode non standard n'apparaît nulle part (il avait bien été obtenu, mais cela ne figurait dans aucun compte rendu, et aucune dérogation officielle n'avait été signée).
- Il n'y a aucune traçabilité entre les spécifications système et les scénarios client (les cas d'utilisation rédigés par le coach). Du coup, la traçabilité entre les scénarios client et les tests fonctionnels n'existe pas non plus…
- L'absence de traçabilité historique des décisions de conception et des changements (de spécifications ou de conception).

La première anomalie n'est donc qu'un «vice de forme». La seconde, loin d'être un défaut de la méthode XP elle-même, ne fait que souligner les problèmes créés par le fait que l'équipe «logiciel» pratiquait XP sans contact direct avec l'équipe «système et validation», équipe qui aurait dû être son client. La troisième est bien une caractéristique de la méthode XP, mais est résolue facilement en conservant un «livre de bord», simple cahier dans lequel l'équipe note les principaux événements au jour le jour.

L'heure des bilans

Au terme d'un an, l'équipe est devenue complètement autonome et n'a plus besoin du coach. Jean-Baptiste, un programmeur expérimenté et impliqué dans la mise en place de la méthode, notamment dans le reporting au management, est désormais prêt à jouer ce rôle.

Avant de quitter le projet, le coach effectue un bilan, au travers de revues avec le management et l'équipe de cette première expérience de la méthode XP.

Tenue des coûts et délais

Le coût du développement logiciel avait été évalué par le coach bien avant le passage à XP à 95 hommes-mois. Le coût final atteindra environ 110 hommes-mois. Quoi qu'il en soit, cela ne nous apprend pas grand-chose sur la productivité d'une équipe XP, ni sur sa capacité à évaluer les charges à long terme. Cela nous indique seulement que le coach a légèrement sous-estimé les charges.

Pendant les premiers mois, la progression fut très rapide, à tel point que le budget a même été revu à la baisse (diminution de douze hommes-mois). Mais c'était avant d'être confronté à la réalité des tests fonctionnels.

Une seule chose est objectivement vérifiée : la planification itérative impliquant fortement les développeurs permet des estimations précises des délais à moyen terme. Deux facteurs ont perturbé l'équipe dans sa capacité à faire des prévisions sur le long terme :

- le décalage par rapport aux tests fonctionnels, déjà évoqué ;
- l'absence d'implication du vrai client dans le projet et le fait qu'il s'agissait d'un projet au forfait : la possibilité d'ajuster le périmètre fonctionnel était très limitée, alors qu'il aurait été possible de se rapprocher du coût final en faisant abstraction de certaines fonctionnalités mineures.

La vision du management

Le management opérationnel (chef de projet, chef de service et directeur des opérations) a une perception positive des aspects suivants de la méthode :

- les indicateurs d'avancement ;
- la motivation des individus et l'esprit d'équipe ;
- la gestion des ressources humaines (diffusion des compétences et des connaissances).

En revanche, il met en garde contre un discours trop radical et souhaite qu'un effort soit fait pour que la méthode XP :

- soit en conformité avec les exigences documentaires (déclinées de façon classique sur le V système même si le cycle suivi n'est pas un V dans la méthode XP) ;
- réponde aux exigences du système qualité, soit par adaptation de la méthode, soit par adaptation du système (dans les deux cas, un effort minimal de rédaction sur la façon de procéder est nécessaire et doit être soumis à approbation).

Le directeur qualité est d'avis que les anomalies constatées lors de l'audit interne peuvent être corrigées et ne sont pas intrinsèques à la méthode XP, et souhaite que la méthode soit ajoutée au système qualité sous forme de processus de développement « optionnel ».

Témoignage du chef de service

L'amélioration de la productivité dans le domaine du logiciel est un souci constant du département « Supervision » ; en effet, cette activité représente plus de la moitié de la charge du département. Le choix d'expérimenter la méthode XP sur un projet de supervision en 1999 a été fait dans cet objectif.

Les résultat attendus en étaient :

– maîtrise des coûts et délais ;

– validation fiabilisée du logiciel (processus itératif) ;

– réactivité de l'équipe (équipe motivée, chacun connaît l'ensemble du logiciel) ;

– qualité du logiciel (grâce au travail en binôme et aux tests unitaires).

Suite à cette première expérience, le bilan est mitigé ; en particulier, on peut déplorer un dérapage des coûts et délais dès le début de la validation fonctionnelle et des tests sur site. On peut invoquer deux raisons à cette dérive. D'abord, une nouvelle méthode requiert un temps et un coût d'apprentissage non négligeables. Ensuite, l'activité « définition et validation du système » n'a pas été effectuée dans le respect de la méthode, ce qui a induit une mauvaise compréhension des besoins, et par conséquent des erreurs de conception.

Cependant, la réactivité de l'équipe face à ce constat a été excellente aussi bien en termes de proposition de modifications de conception qu'en termes de délai de réalisation.

D'autre part, l'équipe a été renouvelée à 80 % au cours du projet ; la méthode a démontré sur ce point une facilité d'intégration de nouveaux éléments.

Enfin, l'équipe a montré, pendant tout le projet, une cohésion de tous les instants et a manifesté une réelle motivation.

La vision des développeurs

L'équipe elle-même fait un bilan très positif sur la méthode XP. D'après les développeurs, cette méthode apporte :

- un gain de productivité ;
- une meilleure qualité du produit ;
- une plus grande motivation et en retour une satisfaction personnelle.

Outre la méthode XP elle-même, la politique de formation continue proposée par le coach (forums hebdomadaires et nombreux topos internes à l'équipe) a contribué à ce bilan.

De l'avis de l'équipe, les problèmes constatés sont dus à une application incomplète ou insuffisante de la méthode XP. En particulier, pour en tirer le maximum de bénéfices, la méthode doit être pratiquée aussi par les équipes système (dans le rôle de client) et de validation (dans le rôle de testeur).

Voyons dans le détail les remarques de l'équipe.

Cycle de vie itératif

Ce processus résout de nombreux problèmes classiques : spécifications inexactes ou changeantes, découverte trop tardive des risques et du coût de l'intégration et de la validation, mise en place trop tardive des outils d'intégration et de validation.

Sur le projet en question, le cycle itératif n'a hélas été pratiqué que par l'équipe logicielle, ce qui explique que certains problèmes aient été rencontrés en fin de projet :

- découverte extrêmement tardive de certains détails de spécifications relatives à des fonctions réalisées six mois plus tôt ;
- découverte tardive du coût de validation et de mise au point ;
- mises au point (corrections de défauts) de fonctions logicielles réalisées longtemps auparavant.

Méthode de tests unitaires

Cette méthode, qui consiste *systématiquement* et *avant de coder* à définir un jeu de tests unitaires auto-vérifiants pour toute classe, s'est révélée extrêmement bénéfique :

- La démarche permet de préciser le besoin et la conception détaillée.
- On procède à moins de développements superflus : on s'arrête dès que les tests passent.
- Il en résulte une amélioration radicale de la qualité du code (moins de défauts).
- Il devient possible de modifier la conception et le code sans risque de régression.

Parfois (rarement), la démarche a été ignorée (difficulté technique de certains tests, paresse ou pression des délais). À moyen terme, cela est presque toujours apparu comme une erreur. Si c'était à refaire l'équipe appliquerait encore plus rigoureusement la méthode.

Travail en binôme, propriété collective du code

Le travail en binôme aussi a fait l'unanimité :

- transfert rapide et permanent des connaissances, à la fois techniques et fonctionnelles ;
- plus grande rigueur dans le respect des méthodes, des règles de codage ;
- meilleur respect des spécifications ;
- meilleure qualité du code (revue de code permanente) ;
- grande homogénéité du code sur l'ensemble du projet ;
- plus grande efficacité et productivité.

Sur ce dernier point (la productivité), tous sont d'accord pour dire qu'en binôme, la productivité globale (prenant en compte l'amélioration de la qualité, le respect des spécifications, etc.) est plus importante. La majorité est même d'avis que la productivité immédiate, sur la seule activité de conception détaillée/codage, est plus importante.

Intégration continue

Cette pratique est perçue comme étant très bénéfique :

- les problèmes d'intégration sont connus immédiatement, et donc résolus rapidement puisqu'on sait exactement d'où vient le problème ;
- tout le monde travaille sur une version récente de l'ensemble du projet, et il est motivant de voir progresser les fonctionnalités ;
- la gestion de configuration est très simplifiée.

Le seul problème de cette méthode est qu'elle nécessite des recompilations fréquentes, ce qui fut pénalisant pendant la longue période où les machines n'étaient pas adaptées à cette pratique. Une fois ce problème résolu, cette façon de travailler se révèle excellente.

Moyens généraux (machines, bureau)

L'équipe est réunie dans un seul bureau, dont l'aménagement est inhabituel :

- un espace central où s'effectue le développement en binôme ;
- une table de réunion ;
- les bureaux individuels en périphérie.

L'équipe est unanime sur le fait que cet aménagement est *fondamental*. Il représente un facteur majeur de son efficacité, grâce à la communication permanente qu'il permet.

Méthode d'estimation, indicateurs

Le bilan de la méthode d'estimation et de suivi est positif, bien que plus mitigé :

- l'estimation à laquelle il est procédé collectivement est plus précise et responsabilise tout le monde ;
- il est utile de pouvoir distinguer les glissements techniques d'une perte d'efficacité ;
- il faudrait rendre la distinction plus objective, en dressant une liste de ce qui constitue un glissement technique et ce qui passe en *load factor*[1].

L'équipe conserve donc une impression positive de la méthode, et la plupart des développeurs en sont devenus de fervents défenseurs. L'impression générale est que les problèmes constatés auraient été résolus par une mise en œuvre encore plus complète de la méthode, notamment avec l'utilisation de tests de recette qui soient en phase avec le développement – y compris, en particulier, des tests automatiques d'IHM.

Témoignage de Florent, développeur

Quand je suis arrivé sur le projet, j'avais déjà dix ans d'expérience dans la programmation objet dont sept ans en C++, les quatre dernières années en tant que chef de projet, le tout sur des projets aussi variés que des programmes de recherche en intelligence artificielle, des études de faisabilité en recherche opérationnelle ou des projets industriels dans le domaine militaire.

C'est dans ce dernier type de projet que j'ai rencontré le plus de difficultés à satisfaire les exigences du contrat. Je dis ici volontairement du contrat et non pas du client car il n'est pas rare de voir des situations aberrantes où le besoin du client évolue au fil du projet mais où ces nouvelles exigences ne peuvent pas être prises en compte du fait de la rigidité du contrat.

La contrainte la plus pesante est le cycle de développement en V qui impose notamment de faire la conception en début du projet, souvent seul. Quand vient l'heure de monter en charge, les développeurs et le concepteur soulèvent des anomalies dans la conception et font évoluer le code sans toujours avoir le temps et l'énergie nécessaire de mettre à jour les documents de conception en parallèle. Par ailleurs, les développeurs peuvent ressentir une sorte de frustration à ne pas pouvoir sortir du cadre rigide d'une conception qui n'est peut-être plus adaptée et à laquelle ils n'ont souvent pas participé. De plus, la plupart du temps, la décomposition en modules qui est faite lors de cette conception préliminaire est accompagnée d'une décomposition correspondante de l'équipe. Ce cloisonnement est souvent un frein à une intégration réussie et crée un manque d'homogénéité du code.

1. Avec l'utilisation désormais recommandée de la vélocité (voir chapitre 6), ce reproche est moins pertinent.

Le dernier point critiquable du cycle en V est l'arrivée très tardive des tests qui en réduit fortement l'utilité.

Ce premier projet XP auquel j'ai participé m'a permis de trouver une réponse concrète à bon nombre de mes expériences infructueuses. Ainsi, dès les premiers jours, je participais à des réunions de conception ponctuelles sur des tâches clairement identifiées dans le cadre d'un incrément de taille raisonnable. En parallèle, je réfléchissais avec le coach à certains aspects d'architecture globale, mais sans les figer.

Par la suite, j'ai rapidement pu transférer bon nombre de mes connaissances par le biais des séances de conception collective et aussi grâce à la programmation en binôme. En outre, et c'est moins évident à imaginer, j'ai moi-même énormément appris à travers les questions parfois naïves, simplistes mais souvent simplificatrices des développeurs moins expérimentés. Apprendre aux autres, c'est d'abord apprendre soi-même, se remettre en question, formaliser son travail et le rationaliser.

Tout cela fonctionne bien à la condition que les membres de l'équipe possèdent un esprit de groupe, et de communication. Travailler dans un projet XP est donc également une très bonne expérience humaine. Le travail en binôme pousse notamment à donner le meilleur de soi-même vis-à-vis de l'autre, à respecter les règles de codages, à rechercher de bonnes solutions, à mettre en œuvre des « design patterns » quand c'est opportun, enfin à rechercher la satisfaction personnelle du travail bien fait.

L'outil qui me semble aujourd'hui le plus puissant de la méthode est la systématisation des tests unitaires qui donne confiance en notre code, et qui permet d'envisager sereinement son remaniement.

Épilogue

Après le départ du coach, l'équipe a continué à appliquer la méthode XP jusqu'à la fin du projet, y compris quand il a fallu prendre en compte un avenant pour une extension du système, qui a pris un an de plus. À cette occasion, un important turn-over s'est produit, ne laissant en place parmi les développeurs de l'équipe initiale que Jean-Baptiste et Michaël. La diffusion des connaissances et la rapidité d'intégration des nouveaux venus ont énormément facilité cette transition.

De son côté, le coach a démarré un deuxième projet, toujours en XP, en appliquant les leçons que l'on avait pu tirer du premier. François, qui avait été développeur sur le premier projet, y joue le rôle de client, et est donc responsable à la fois de la définition des scénarios client et des tests de recette. Il est intégré à l'équipe et développe les tests de recette en parallèle des développements. Au bout de quelque temps, il parvient même à fournir les tests de recette avant que les développeurs ne commencent les développements !

Les développeurs, au nombre de sept, ne proviennent pas du premier projet et découvrent pour la plupart XP. Ils apprennent facilement la méthode et l'apprécient énormément. Encore une fois, la montée en charge est progressive, et l'intégration et la formation des derniers arrivés se font très rapidement.

Dès le début, l'application est conçue pour améliorer la testabilité dans les deux domaines qui avaient posé des problèmes lors du premier projet : les tests de l'interface graphique et les tests des applications distribuées (multi-processus et multi-machines).

Les itérations sont plus courtes (trois semaines pendant un an, puis deux semaines).

D'autres projets, sans adopter complètement la méthode XP, ont retenu certaines pratiques, notamment les tests unitaires, le fait de réunir l'équipe dans un seul bureau et de ne pas la diviser, par exemple, en équipe serveur et équipe interface graphique.

Enfin, la direction qualité a demandé au coach d'étudier l'intégration de la méthode XP dans le système qualité de l'entreprise.

Conclusion

Dans la mesure où ce livre a pour vocation d'enseigner l'esprit et la pratique de la méthode XP et de contribuer à son adoption par les entreprises, ce retour d'expérience est instructif à plusieurs titres. On notera tout d'abord que la méthode XP ne nécessite pas d'investissement particulier : bien qu'il ne faille pas négliger le temps nécessaire à une compréhension profonde de la méthode XP et *a fortiori* à sa réelle intégration au sein d'une entreprise, qui a une culture méthodologique antérieure, il n'est pas difficile de réussir son premier projet XP. La simplicité et la rigueur des pratiques permettent de se lancer dans un premier projet sans avoir à se poser trop de questions, et certaines d'entre elles, notamment la programmation en binôme et l'accent mis sur les tests automatiques, apportent de telles améliorations en termes d'efficacité et de qualité par rapport à la plupart des méthodes courantes que, même avec des imperfections, un premier projet a de fortes chances de faire ses preuves.

Comme toujours, l'ingrédient humain est primordial. À ce sujet, on relèvera que, contrairement à ce que laissent entendre certaines idées reçues, ce n'est pas forcément le management qu'il sera le plus dur de convaincre du bien-fondé d'XP. On l'a vu dans ce récit, les responsables sont en principe sensibles aux promesses de la méthode, car qui peut prétendre aujourd'hui ne pas travailler dans un contexte où les besoins sont imprécis et changeants, les délais courts et les impératifs de qualité et d'efficacité très forts ? En donnant sa chance à XP, tout en restant exigeants sur les résultats concrets, les décideurs jouent parfaitement leur rôle (cf. chapitre 2) et contribueront à la réussite du projet.

Ce qui est plus surprenant, c'est la réticence que rencontre parfois XP de la part des développeurs *n'ayant jamais pratiqué la méthode*, et ce en dépit des nombreux aspects séduisants de la méthode du point de vue des programmeurs (le droit à faire un travail de qualité, la forte implication de tous dans la planification et la conception). Plutôt que de spéculer sur d'éventuelles explications à ce type de comportement, il est plus utile de souligner à quel point les développeurs qui ont réellement pratiqué la méthode en deviennent généralement de fervents supporters.

Bien que nous soyons aujourd'hui rodés dans cet exercice particulier qui consiste à présenter la méthode, par écrit ou oralement, en trois jours ou cinq minutes, nous n'avons jamais réussi à attirer aussi bien l'attention qu'en conviant les gens à venir voir travailler une équipe, ou discuter avec les participants. L'esprit d'équipe qui s'en dégage, la décontraction et la bonne entente doublée de professionnalisme et d'un attachement réel à la réussite du projet, la ferveur avec laquelle ils défendent leurs pratiques, sont autant de preuves que l'Extreme Programming est une alchimie qui permet aux qualités individuelles de s'exprimer et se développer au sein d'une équipe qui travaille ensemble, simplement et efficacement.

Annexes

A1

Glossaire

	Définition	Terme anglais
Agile	Le terme « agile » qualifie une famille de méthodes mettant l'accent sur les individus, l'acceptation du changement, la collaboration continue entre maîtrise d'œuvre et maîtrise d'ouvrage et le pragmatisme – un cycle itératif court visant à bénéficier immédiatement d'un logiciel utilisable, et la prise en compte de besoins changeants. Ces valeurs et objectifs ont été définis par un collège de dix-sept praticiens qui a pris pour nom l'Alliance agile. XP est la plus populaire et la plus pratiquée des méthodes agiles.	**Agile**
Assistance technique	Mode contractuel de réalisation d'un projet dans le cadre duquel le prestataire s'engage sur les moyens fournis. Les coûts sont fonction du temps passé par l'équipe pour mener à bien le projet.	**Time & materials**
Assurance qualité	Activité visant à vérifier l'adéquation d'un processus ou d'un produit par rapport à des exigences (besoins, normes…).	**Quality Assurance**
Binôme	Association de deux programmeurs sur une même machine pour l'écriture du code. Les binômes sont renouvelés régulièrement.	**Pair**
C++	Langage de programmation compilé, à typage statique, permettant la programmation procédurale ou orientée objet et la programmation générique.	
Carottage	Exploration d'un problème (fonctionnel ou technique) *via* une solution minimaliste (prototype) isolée du contexte habituel du projet, visant à obtenir un feedback rapide pour en confirmer la validité ou se faire une idée de la difficulté.	**Spike**
Client	L'un des rôles XP, chargé d'une part d'exprimer le besoin en rédigeant des scénarios client et en définissant des tests de recette correspondants, et d'autre part de définir les priorités lors des séances de planification. Souvent associé à la maîtrise d'ouvrage.	Customer

	Définition	Terme anglais
CMM (Capability Maturity Model)	Modèle d'évaluation et d'amélioration de la maturité d'une organisation au travers de sa capacité à réussir le développement de systèmes et de produits. Définit cinq niveaux de maturité, exigeant chacun certaines activités et processus clés.	CMM
Coach	L'un des rôles XP, chargé de coordonner les autres acteurs du projet, et plus généralement d'améliorer en permanence la façon dont XP est mis en œuvre dans le projet.	Coach
Communication	L'une des quatre valeurs fondamentales. La communication franche, ouverte et honnête est un antidote à la plupart des dérapages possibles sur un projet – de délais, de qualité…	Communication
Conception	Étape consistant à modéliser (ou plus généralement réfléchir à) ce qui va être codé. Dans un cadre XP, conception et programmation sont indissociables ; la conception ne produit par ailleurs pas nécessairement un autre «document» que le code source lui-même.	Design
Courage	L'une des quatre valeurs fondamentales d'XP, qui rappelle la nécessité d'affronter les difficultés en y faisant face plutôt que de les ignorer ou de repousser à plus tard leur prise en compte.	Courage
CVS (Concurrent Version System)	Outil (logiciel libre) de gestion des versions, adapté à la méthode XP car il permet la modification simultanée d'un même fichier par plusieurs personnes, tout en fournissant une aide à la fusion de ces travaux parallèles.	CVS
Cycle en V	Manière d'organiser les activités de développement (d'un système ou d'un logiciel) par une succession de phases séquentielles, d'abord «descendantes» (spécification, conception, réalisation) puis «remontantes» (tests unitaires, intégration, validation).	Waterfall model
Estimation	Évaluation de l'importance (en termes de temps de réalisation) d'un scénario client ou d'une tâche. En XP, l'estimation est basée sur une comparaison avec un scénario ou une tâche analogue d'une itération précédente, et s'exprime en unités abstraites, indirectement liées au temps calendaire.	Estimate
Feedback	L'une des quatre valeurs fondamentales. Quand vous vous posez une question, ne spéculez pas sur la réponse : agissez, expérimentez ou questionnez ; vous l'obtiendrez sous la forme de «feedback».	Feedback
Forfait	Réalisation d'un projet dans un cadre contractuel où le prestataire s'engage à l'avance sur le contenu du produit, ainsi que sur les coûts et les délais du projet.	Fixed Time, Fixed Price
Intégration continue	Démarche consistant à intégrer aussi fréquemment que possible (au moins une fois par jour) les développements unitaires sous forme d'une application globale, prête au déploiement et susceptible de passer les tests de recette.	Continuous integration
Itération	Période de deux ou trois semaines rythmant le travail de toute l'équipe. Chaque itération commence par une séance de planification, puis consiste à réaliser, recetter et livrer les scénarios clients sélectionnés.	Iteration

	Définition	Terme anglais
Java	Langage de programmation partiellement compilé, puis interprété par une machine virtuelle, à typage statique, orienté objet.	
Livraison	Fourniture au client, pour utilisation immédiate, d'une version du logiciel en état opérationnel et vérifiant tous les tests de recettes exigés pour cette version.	Release
Maîtrise d'œuvre	Partie contractante qui prend en charge la responsabilité de réaliser le projet ou d'organiser sa réalisation.	
Maîtrise d'ouvrage	Partie contractante à qui reviendra le produit réalisé, et qui confie à la maîtrise d'œuvre la réalisation du projet. Généralement désignée par le terme «client».	
Manager	L'un des rôles XP, correspondant le plus souvent au supérieur hiérarchique des programmeurs. Il est chargé des aspects administratifs du projet, et c'est à lui que l'équipe rend des comptes.	Manager
Métaphore	L'une des 13 pratiques. La description de haut niveau, en termes imagés et compréhensibles par toute l'équipe, non-développeurs compris, de l'architecture technique et fonctionnelle du projet. Par extension, désigne toute description de ce type, y compris d'une partie du projet, et aussi «l'exercice» même qui consiste à inventer des métaphores.	Metaphor
Plan d'itération	Plan précis définissant, pour une itération, les scénarios client qui ont le plus de valeur pour le client et qu'on prévoit de pouvoir implémenter au cours de cette itération, si la vélocité reste constante.	Iteration plan
Plan de livraison	Plan définissant, sur N itérations, une livraison du logiciel correspondant en gros à une «version mineure» (1.1, 1.2, etc.) comportant un lot de fonctionnalités délimité approximativement.	Release plan
Planification itérative	L'une des 13 pratiques. Consiste à établir, au terme d'une séance de planification, un plan de livraison et/ou un plan d'itération comprenant un certain nombre de scénarios client. Ces plans sont revus à chaque itération.	Planning Game
Pratique	Les 13 pratiques d'XP, également appelées «études», sont des recommandations concrètes, chacune bénéfique en elle-même, sur la façon de conduire ou de réaliser un projet ; il est conseillé de les maîtriser d'abord en les appliquant consciencieusement, puis de les adapter.	Practice
Programmeur	L'un des rôles XP, chargé des estimations de charges et de risques, de la décomposition des scénarios client en tâches, de l'écriture de tests unitaires, de la programmation et de la conception.	Programmer
Prototype	*Voir* carottage.	Prototype
Python	Langage de programmation interprété, disponible en logiciel libre, à typage dynamique, permettant la programmation procédurale ou orientée objet.	

	Définition	Terme anglais
Remaniement	L'une des 13 pratiques. Consiste à revenir sur le code en permanence pour le rendre plus simple et plus clair, et faciliter ainsi l'ajout de nouveau code ; cela permet de faire émerger la conception de manière progressive tout au long du développement. Le terme s'applique à la fois à l'activité («faire du remaniement») et à une technique particulière parmi un catalogue («appliquer le remaniement *renommer une classe*», par exemple).	Refactoring
RUP (Rational Unified Process)	Processus ou méthode, défini par la société Rational, permettant d'organiser les activités de développement d'un logiciel, en s'appuyant notamment sur UML.	
Scénario client	Représentation d'un besoin exprimé par le client, sous la forme de quelques phrases inscrites sur une fiche bristol. Il s'agit de l'unité principale d'estimation, de planification et de suivi des développements.	User story
Séance de planification	Séance collective au cours de laquelle le client et les programmeurs établissent ou révisent de concert les plans de livraison et d'itération, en fonction de la vélocité de la précédente itération, des estimations faites par les programmeurs et des priorités définies par le client.	Planning Game
Simplicité	L'une des quatre valeurs fondamentales d'XP qui privilégie un code épuré et simple à un code générique et complexe. Pour une équipe de développement, la simplicité consiste notamment à ne pas en faire plus que ce qui est strictement nécessaire dans l'immédiat, à éviter toute duplication dans le code, à faire le moins de classes et de méthodes possibles tout en conservant un code qui exprime clairement la conception. La simplicité se traduit aussi par l'utilisation d'outils simples (comme des fiches cartonnées...).	Simplicity
Smalltalk	Langage de programmation interprété par une machine virtuelle, à typage statique, orienté objet.	
Squeak	Langage de programmation, environnement de développement (bibliothèques) et d'exécution (machines virtuelles), sur-ensemble du langage Smalltalk.	
Test de recette	Test permettant d'assurer de manière automatique et contractuelle la conformité du logiciel avec les exigences du client.	Acceptance Test
Test unitaire	Test codé par les programmeurs dans le même langage que celui utilisé pour le reste de l'application. Chaque classe du logiciel en développement possède un test unitaire correspondant, qui exerce une ou plusieurs instances de la classe testée et en vérifie automatiquement le comportement au moyen d'assertions.	Unit Test
Testeur	Rôle XP (parfois joué par le client lui-même) consistant à développer les tests de recette définis par le client et à communiquer les indicateurs de progression associés.	

	Définition	Terme anglais
Tracker	L'un des rôles XP, consistant à se rendre tous les deux ou trois jours auprès de chaque programmeur pour faire un point sur l'avancement des travaux, relever les éventuels problèmes rencontrés, et le cas échéant faire part de ces problèmes au coach afin qu'il trouve une solution pour aider le programmeur en question.	Tracker
UML (Unified Modeling Language)	Notation graphique, normalisée par l'OMG (Object Management Group), permettant de représenter diverses facettes d'un système, en particulier d'un logiciel.	
Vélocité	Indicateur permettant de faire la correspondance entre des durées estimées – mesurées en nombre de «points» abstraits – et des périodes calendaires. À un instant donné, la vélocité de l'équipe correspond au nombre de points de scénarios client réalisés à l'itération précédente.	Velocity
Wiki Wiki Web	Site Web (*http://c2.com/cgi/wiki*) créé par Ward Cunningham, dont les pages sont ajoutées et modifiées directement et simplement par les visiteurs, et qui traite en particulier de développement logiciel, *patterns* et Extreme Programming.	
xUnit	Famille des outils (harnais) de tests unitaires, déclinés pour divers langages de programmation (SUnit pour Smalltalk ou Squeak, JUnit pour Java, CppUnit pour C++…).	

A2

Bibliographie

Livres XP

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

Kent Beck présente dans ce premier ouvrage consacré à l'Extreme Programming les principes et les fondements de la méthode. Cet ouvrage permet de comprendre les tenants et les aboutissants de l'Extreme Programming, même si les aspects pratiques de sa mise en œuvre y sont abordés de manière lapidaire.

Ron JEFFRIES, Ann ANDERSON et Chet HENDRICKSON, *Extreme Programming Installed*, Addison-Wesley, 2000.

Cet ouvrage donne une description concrète et vivante des pratiques XP et complète le premier livre, *Extreme Programming Explained*.

Kent BECK et Martin FOWLER, *Planning Extreme Programming*, Addison-Wesley, 2000.

Ce livre est focalisé sur les pratiques de gestion de projet XP. Il reprend la description de ces pratiques et les complète par de nombreux conseils de mise en œuvre. Il s'adresse principalement aux coachs, ou à des décideurs envisageant de proposer XP à leurs équipes.

Martin FOWLER, *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 1999.

Un ouvrage complètement dédié à la pratique de remaniement (refactoring), qui est si indispensable à XP. Martin Fowler y décrit les principes de cette démarche et dresse une liste d'environ 70 remaniements, en fournissant pour chacun d'entre eux une description précise des étapes à suivre pour modifier le code en minimisant les risques d'erreur. Comme les Design Patterns, cette liste de remaniements établit un vocabulaire commun qui facilite la communication entre les développeurs.

Andrew HUNT et Dave THOMAS, *The Pragmatic Programmer: from journeyman to master*, Addison-Wesley, 2000.

Les pratiques d'XP sont adossées à une théorie ou tout du moins à un modèle du processus de développement. Bien que la philosophie à laquelle adhèrent les auteurs, fort proche de l'esprit d'XP, transparaisse tout au long de ce livre, ils s'efforcent cependant d'éviter toute approche méthodologique, tout discours de la méthode : le lecteur trouvera avant tout dans ce livre une somme considérable de conseils pratiques – de ceux qui font regretter aux programmeurs chevronnés qu'ils ne les aient pas connus dès le début.

Business Interactif, *Extreme Programming – Méthodes agiles (Livre blanc)*, *www.businessinteractif.fr*, 2001.

Ce livre blanc présente un panorama des principales méthodes agiles (dont l'Extreme Programming) : caractéristiques, positionnement des méthodes les unes par rapport aux autres. RAD et UP sont également positionnés par rapport à l'agilité.

Livres sur la gestion de projet

Tom DEMARCO, Timothy LISTER, *Peopleware: Productive Projects and Teams*, Dorset House Publishing, 1987.

Tom DeMarco et Timothy Lister se font les avocats d'environnements de travail plus humains et insistent sur l'influence importante de la cohésion de l'équipe sur sa productivité. Un ouvrage désormais classique.

Frederick P. BROOKS, *The Mythical Man-Month*, Addison-Wesley, 1995.

Quelques vérités fondamentales sur la gestion de projets, parmi lesquelles on retiendra notamment que les hommes ne sont pas interchangeables avec les mois. Publié en 1975, cet ouvrage a été réédité en 1995 et reste en tous points d'actualité…

Gerald WEINBERG, *Quality Software Management 1-4*, Dorset House, 1997.

Weinberg expose les trois compétences fondamentales en matière de gestion de projet : *primo*, savoir les concevoir comme des systèmes complets et complexes, *secundo*, savoir observer ce qui se passe au sein de ces systèmes et *tertio*, savoir réagir de façon appropriée à nos observations et à nos connaissances sur ces systèmes. Référence indispensable en la matière.

Ouvrages généraux

Peter SENGE, *La Cinquième Discipline*, Éditions générales First, 1992.

Une introduction à l'approche systémique dans l'entreprise et à d'autres techniques destinées à permettre la constitution d'équipes vouées à l'apprentissage permanent : la maîtrise personnelle des méthodes, la déconstruction des modèles mentaux, le partage de visions et l'apprentissage en équipe.

Christopher ALEXANDER, *The Timeless Way of Building*, Oxford University Press, 1979.

Les travaux de l'architecte Christopher Alexander ont inspiré bien des développeurs, et donné naissance à la communauté des patterns (dont le résultat le plus célèbre est le livre *Design Patterns*). Dans ce livre, Alexander défend une approche de la construction qui replace les habitants eux-mêmes au centre du processus de construction, à travers une démarche progressive (pour ne pas dire itérative) qui permet à ces derniers de décider des évolutions susceptibles d'améliorer leur cadre de vie – le parallèle avec la notion de « Client XP » est difficile à éviter.

Christopher ALEXANDER & *al*., *The Oregon Experiment*, Oxford University Press, 1975.

Ce petit ouvrage décrit le retour d'expérience de Christopher Alexander sur un projet d'amélioration de l'université d'Orégon. Il y décrit les principes de son approche, qui sont très proches de la voie prônée par XP : construction progressive et itérative, implication des utilisateurs, émergence du tout à partir d'actes locaux, amélioration permanente (remaniement ?) de l'ensemble.

Robert PIRSIG, *Traité du zen et de l'entretien des motocyclettes*, Éditions du Seuil, 1998.

On pourra lire dans ce roman une réflexion philosophique sur la notion de Qualité dont on ne saurait se passer.

Sites Internet

http://www.extremeprogramming.org

http://www.xprogramming.com

http://c2.com/cgi/wiki?ExtremeProgrammingRoadmap

http://www.agilealliance.org

http://www.martinfowler.com

http://www.objectmentor.com

http://www.xp123.com

http://www.xp-france.org

http://www.junit.org

http://www.pairprogramming.com

comp.software.extreme-programming

http://www.design-up.com

Organismes de formation

ObjectMentor (États-Unis) – *http://www.objectmentor.com*

Connextra (Royaume-Uni) – *http://www.connextra.com/products/xp_training.htm*

Exoftware (Irlande) – *http://www.exoftware.com*

XPTools (Suède.)– *http://www.xptools.com*

TWAM (France) – *http://www.twam.com/cursus.html*

Oza (France) – *http://www.oza.fr*

A3

Exemples de code

Nous donnons ici les exemples de code complets des chapitres 3 et 4.

Remarque

Pour compiler et tester ces exemples, il faut bien entendu créer au préalable un répertoire nommé `données` contenant les fichiers `interface.cfg` et `systeme.cfg` appropriés. Cette étape est laissée en exercice pour le lecteur.

Exemple du chapitre 3, avec son test

Code non remanié

```
import java.io.*;
import java.util.*;

public class Config {

   public static void main(String[] args) throws Throwable {
      Vector attenduUtilisateur = new Vector();
         attenduUtilisateur.add("un");
         attenduUtilisateur.add("deux");
         attenduUtilisateur.add("trois");

      Vector attenduSysteme = new Vector();
         attenduSysteme.add("aaa");
         attenduSysteme.add("bbb");
```

```
      attenduSysteme.add("ccc");

   Config test = new Config();
   test.lireConfigurations();

   if (        attenduUtilisateur.equals(test.configUtilisateur)
        &&  attenduSysteme.equals(test.configSysteme)) {
      System.out.println("ok...");
   } else {
      System.out.println("PAS BON !");
      System.out.println("user:"+test.configUtilisateur+" !=
      "+attenduUtilisateur);
      System.out.println("syst:"+test.configSysteme+" != "+attenduSysteme);
   }
}

private Vector configUtilisateur;
private Vector configSysteme;

public void lireConfigurations() throws IOException {

   Vector lignesUtilisateur = lireConfigurationUtilisateur();
   Vector lignesSysteme = lireConfigurationSysteme();

   this.configUtilisateur = lignesUtilisateur;
   this.configSysteme = lignesSysteme;
}

public Vector lireConfigurationUtilisateur() throws IOException {
   File configUtilisateur = new File("donnees/interface.cfg");
   Vector lignesUtilisateur = new Vector();
   BufferedReader lecture = new BufferedReader(new
   FileReader(configUtilisateur));
   String ligneCourante;
   do {
      ligneCourante = lecture.readLine();
      if (ligneCourante != null) {
      lignesUtilisateur.add(ligneCourante);
      }
   }
   while (ligneCourante != null);
   lecture.close();
   return lignesUtilisateur;
}

public Vector lireConfigurationSysteme() throws IOException {
   File configSysteme = new File("donnees/systeme.cfg");
```

```
      Vector lignesSysteme = new Vector();
      BufferedReader lecture = new BufferedReader(new FileReader(configSysteme));
      String ligneCourante;
      do {
         ligneCourante = lecture.readLine();
         if (ligneCourante != null) {
         lignesSysteme.add(ligneCourante);
         }
      }
      while (ligneCourante != null);
      lecture.close();
      return lignesSysteme;
   }

}
```

Code après remaniement

Le même code, après les remaniements expliqués au chapitre 3.

```
import java.io.*;
import java.util.*;

public class Config {

   public static void main(String[] args) throws Throwable {
      Vector attenduUtilisateur = new Vector();
      attenduUtilisateur.add("un");
      attenduUtilisateur.add("deux");
      attenduUtilisateur.add("trois");

      Vector attenduSysteme = new Vector();
      attenduSysteme.add("aaa");
      attenduSysteme.add("bbb");
      attenduSysteme.add("ccc");

      Config test = new Config();
      test.lireConfigurations();

      if (        attenduUtilisateur.equals(test.configUtilisateur)
           &&  attenduSysteme.equals(test.configSysteme)) {
         System.out.println("ok...");
      } else {
         System.out.println("PAS BON !");
         System.out.println("user:"+test.configUtilisateur+" !=
```

```
            "+attenduUtilisateur);
            System.out.println("syst:"+test.configSysteme+" != "+attenduSysteme);
        }
    }

    private Vector configUtilisateur;
    private Vector configSysteme;

    public void lireConfigurations() throws IOException {
        this.configUtilisateur = lireConfiguration("interface");
        this.configSysteme = lireConfiguration("systeme");
    }

    public String repertoireBase() {
        return ".";
    }

    public Vector lireConfiguration(String zone) throws IOException {
        String nom = nomFichierConfiguration(zone);
        LecteurConfiguration lecteur = new LecteurConfiguration();
        lecteur.traiterFichier(nom);
        new AfficheurConfiguration().traiterFichier(nom);
        return lecteur.resultat();
    }

    private String nomFichierConfiguration(String zone) {
        String nom = repertoireBase();
        if (!repertoireBase().endsWith("/")) nom = nom + "/";
        nom = nom + "donnees/"+zone+".cfg";
        return nom;
    }

}

abstract class TraitementConfiguration {

    public void traiterFichier(String nomFichier) throws IOException {
        File fichier = new File(nomFichier);
        BufferedReader lecture = new BufferedReader(new FileReader(fichier));
        String ligneCourante;
        do {
            ligneCourante = lecture.readLine();
            if (ligneCourante != null) {
            traiterLigne(ligneCourante);
            }
        }
        while (ligneCourante != null);
```

```
      lecture.close();
   }

   public abstract void traiterLigne(String ligneCourante);

}

class LecteurConfiguration extends TraitementConfiguration {

   Vector lignes = new Vector();

   public void traiterLigne(String ligneCourante) {
      lignes.add(ligneCourante);
   }

   public Vector resultat() {
      return lignes;
   }

}

class AfficheurConfiguration extends TraitementConfiguration {

   public void traiterLigne(String ligneCourante) {
      System.out.println(ligneCourante);
   }

}
```

Exemple du chapitre 4

```
import junit.framework.*; // Ne sera pas répété par la suite

public class FibTest extends TestCase {
   // Junit a besoin de la ligne suivante
   public FibTest(String name) {super(name);}
   public void testFib0() {
      assertEquals(0,new Fib().fib(0));
   }
   public void testFib1() {
      assertEquals(1,new Fib().fib(1));
   }
   public void testFib2() {
      assertEquals(1,new Fib().fib(2));
```

```
   }
   public void testFib3() {
      assertEquals(2,new Fib().fib(3));
   }
   public void testFib4() {
      assertEquals(3,new Fib().fib(4));
   }
   public void testFib5() {
      assertEquals(5,new Fib().fib(5));
   }
   public void testFib6() {
      assertEquals(8,new Fib().fib(6));
   }
}

class Fib {
   public int fib(int terme) {
      if (terme <= 1) return terme;
      return fib(terme-1)+fib(terme-2);
   }
}
```

A4

Aide-mémoire XP

Les treize pratiques de l'Extreme Programming

1. Tests unitaires
2. Tests de recette
3. Planification itérative
4. Client sur site
5. Programmation en binôme
6. Intégration continue
7. Remaniement (refactoring)
8. Livraisons fréquentes
9. Conception simple
10. Métaphore
11. Responsabilité collective du code
12. Règles de codage
13. Rythme durable

Charte des droits respectifs des développeurs et des clients

Un client a le droit :

- de bénéficier d'un plan d'ensemble, montrant ce qui peut être accompli, pour quand, à quel coût ;
- d'obtenir le plus de valeur possible de chaque semaine de programmation ;
- de constater des progrès sur une application qui marche, comme doivent le prouver les tests répétables qu'il spécifie ;
- de changer d'avis, de substituer des fonctionnalités, et de changer ses priorités sans en payer un prix exorbitant ;
- d'être informé des modifications au calendrier de réalisation, assez tôt pour avoir le choix de réduire le périmètre fonctionnel, et retrouver ainsi la date de livraison initiale ;
- d'annuler le projet à tout moment et de disposer d'une application utile et utilisable en contrepartie de son investissement à ce jour.

Un développeur a le droit :

- de savoir ce qui est demandé, avec des priorités clairement déclarées ;
- de fournir un travail de qualité en toute occasion ;
- de demander et recevoir de l'aide de la part de ses pairs, de ses clients ;
- d'émettre et de réviser ses propres estimations de coûts ;
- d'accepter des responsabilités, qui ne peuvent lui être imposées ;
- de bénéficier d'un rythme de travail durable.

L'agilité en développement logiciel

L'alliance agile préconise de renverser certaines priorités. L'importance est ainsi donnée :

- aux *individus* et aux *interactions* plus qu'aux processus et aux outils ;
- à des *logiciels immédiatement disponibles* plus qu'à une documentation exhaustive ;
- à la *collaboration avec le client* plus qu'à la négociation contractuelle ;
- à la *réactivité face au changement* plus qu'au respect d'un plan.

Il va sans dire que la valeur des seconds termes n'est pas mise en doute mais que, tout simplement, préférence est donnée aux premiers : individus et interactions, fonctionnement immédiat des logiciels, collaboration et réactivité.

Index