Livre blanc Xebia

Evitez le surendettement :

Maîtrisez votre dette technique

Lorsque, sur un projet informatique, il devient difficile d'ajouter de nouvelles fonctionnalités et que la maintenance devient un véritable calvaire, c'est le signe que la dette technique s'est accumulée en excès et qu'il est temps de la rembourser.

Dans la majorité des cas, cette dette trouve son origine dans des compromis faits sur les choix de conception et de réalisation, souvent pour des raisons de temps.

La dette doit être gérée de manière permanente sur les projets : cela passe par de bonnes pratiques de développement telles que le refactoring systématique, le pair programming, les tests unitaires, etc, mais également par la mise en place d'outils d'intégration continue et de mesures de la qualité du code qui nous permettront de diminuer efficacement cette dette.

http://blog.xebia.fr
http://www.xebia.fr

Xebia IT Architects SAS
10/12 Avenue de l'Arche
92419 Courbevoie Cedex
Tél : 01 46 91 76 16
Mail : info@xebia.fr

Préambule

Le 4 Juin 1996, à 9h35 le vol 501 de la fusée Ariane 5 effectue son premier décollage. Quelques secondes plus tard, le système de guidage inertiel reçoit trop d'informations et se met hors service, car reconnu défaillant. L'ordinateur de bord est alors notifié qu'un dysfonctionnement est en cours et compromet les informations concernant la trajectoire de la fusée. Cette modification de la trajectoire entraîne l'arrachage d'un moteur d'appoint, déclenchant l'auto destruction de la fusée. Des analyses plus approfondies ont démontré que le système de guidage inertiel est lui même la cause de cet échec. Conçu à l'époque pour Ariane 4, il n'était plus nécessaire pour Ariane 5. Maintenu actif pour des raisons de commodité, ce système s'est avéré être à l'origine d'un des bugs informatiques les plus coûteux de l'histoire.

Au-delà du caractère spectaculaire de cet exemple, il est intéressant de noter que l'origine du dysfonctionnement réside dans un module développé pour une version antérieure de la fusée et devenu obsolète. Ce vestige de code, maintenu dans l'application sans être nécessaire pour son fonctionnement, est l'une des formes de ce que Ward Cunningham[1] désigne sous le terme de *dette technique*.

Pour introduire la notion de *dette technique*, nous pouvons recourir à l'analogie de la dette financière : un emprunt se compose du capital et des intérêts associés. Nos mensualités servent à rembourser une part du capital et une part des intérêts. Si les remboursements sur le capital ne sont pas réguliers, les intérêts, directement calculés sur ce dernier demeureront importants. Un emprunteur averti cherchera à rembourser au plus vite le capital pour diminuer les intérêts. En transposant cet exemple sur les projets logiciels, le code endetté de nos applications correspond au capital et les bugs et les maintenances représentent les intérêts. Dès lors que nous ajoutons de nouvelles fonctionnalités, le capital augmente et génère davantage d'intérêts, c'est à dire de nombreux bugs et une maintenance accrue.

La réduction ou ne serait-ce que le maintien à un niveau acceptable de la *dette technique* passe par une gestion active et permanente sur le projet dont l'objectif est de permettre de rembourser les intérêts régulièrement et au plus tôt.

A travers ce document, nous découvrirons en quoi la dette technique ralentit la productivité des équipes et nuit aux projets. Nous mettrons en évidence ses mécanismes sous jacents et les leviers d'actions dont nous disposons. Enfin, nous montrerons comment elle se gère au quotidien, par l'instauration de bonnes pratiques de développement et la mise en place d'outils, pour enfin aborder quelques stratégies complémentaires, mais essentielles pour venir à bout de la dette technique.

1 Ward Cunningham : http://fr.wikipedia.org/wiki/Ward_Cunningham

Table des matières

Identifier la dette technique

Définition

Terme inventé par Ward Cunningham, la dette technique représente des parties de code non utilisées ou dans lesquelles il est difficile d'effectuer des modifications et évolutions. La mise en place d'un projet logiciel est similaire aux différentes étapes de la construction d'une maison. Dès lors que les dessins d'architecture sont validés, la construction peut commencer : les fondations, les murs, le toit, etc. Il en est de même pour les logiciels : nous retrouvons les schémas d'architecture, le socle de développement, l'implémentation de base, etc.

Etudions de plus près la construction d'une maison : elle nécessite des matériaux comme du béton, des tuyaux, des fils, ainsi que de la main d'œuvre. Admettons que nous ne disposons pas des budgets nécessaires et que le temps presse : un hiver rude arrive et nous devons absolument loger sous un toit. Nous prenons alors le parti de trouver des matériaux et de la main d'œuvre moins chers. Cette décision nous permet d'être plus réactif, et également de réaliser des économies. Imaginons maintenant que quelques années passent et que nous avons un problème d'électricité. Nous cherchons à joindre la personne ayant effectué l'installation, mais nous n'y parvenons pas. Nous décidons alors de contacter un autre électricien; malheureusement ce dernier nous annonce et nous prouve que l'installation n'est pas aux normes et que des interventions significatives sur le circuit électrique sont requises. Nous devons alors financer ces réparations, assimilables à de la dette que nous avons contractée. Ces réparations entraînent un surcoût qui, potentiellement, excèdera de loin les économies réalisées lors de la construction initiale.

Il en va de même avec les projets logiciels : si pour des raisons de coûts et de délais la qualité des implémentations est revue à la baisse, la dette technique s'accumulera et nous devrons la rembourser tôt ou tard. Plus nous repoussons l'échéance et plus les remboursements à effectuer seront importants, c'est à dire que les modifications et corrections seront plus difficiles à mettre en place. Cela se traduira par des coûts supplémentaires, souvent largement supérieurs aux économies réalisées initialement.

Il existe plusieurs types de dette :

- La **dette naturelle** :

  Tout logiciel qui grossit est soumis aux lois de l'entropie. Sa complexité croît et ses coûts de maintenance et d'évolution croissent en conséquence. A mesure que le projet avance, les différents développeurs introduisent du « bruit » dans le code issu d'erreurs de conception, d'implémentations non conformes aux normes. La **dette naturelle** est quasiment inévitable sur un projet qui regroupe souvent des développeurs d'expériences et de sensibilités différentes. Les erreurs les plus courantes : duplication de code, méthodes excessivement longues, code non utilisé, commentaires incompréhensibles, etc, sont désignées par Martin Fowler sous l'appellation de Code Smell[1]. Nous reviendrons sur l'identification et la diminution de ces *Code Smells* dans la deuxième partie de ce document.

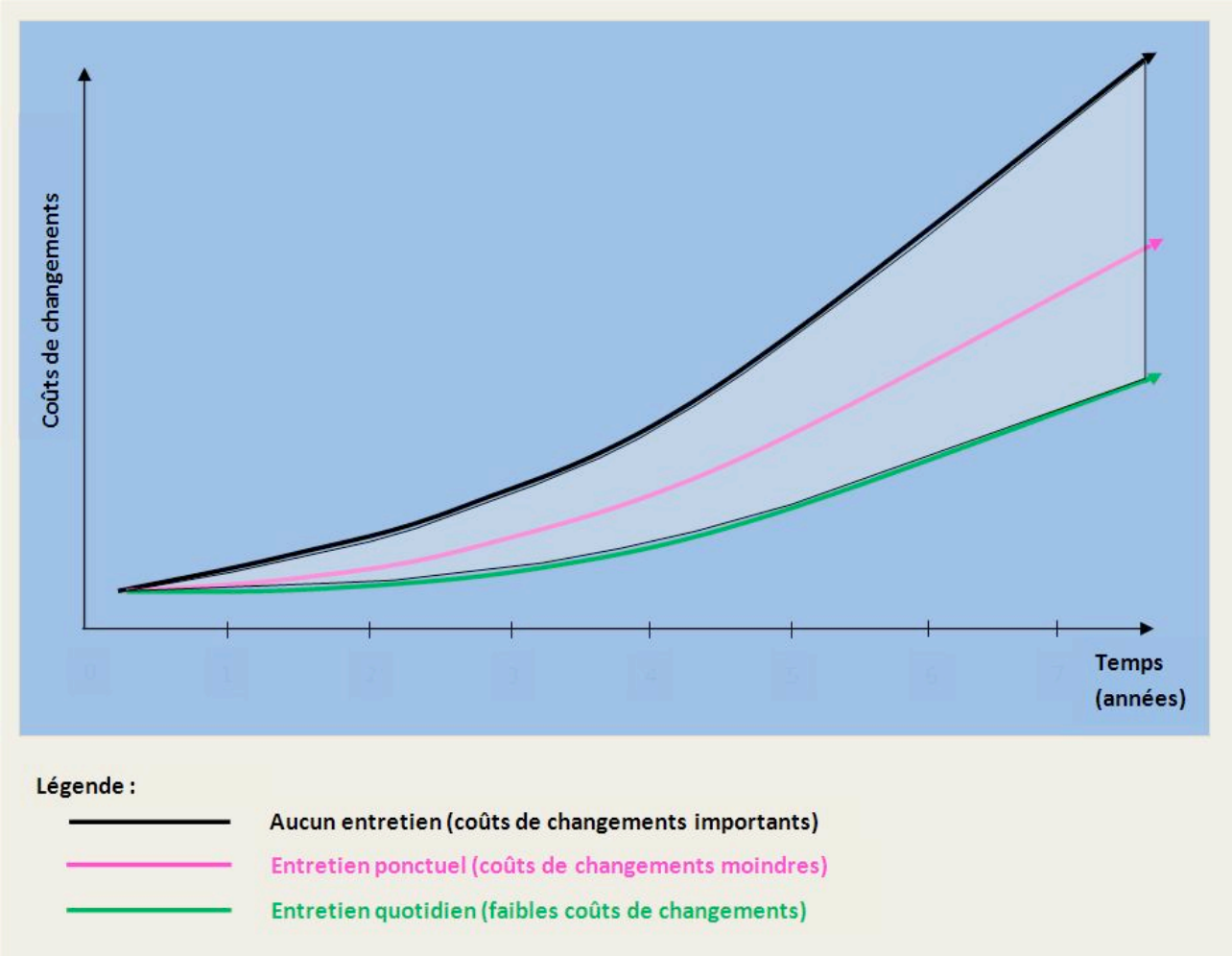
- La **dette intentionnelle** :

  Reprenons notre exemple de la dette financière : nous souhaitons financer un autre achat rapidement, nous effectuons alors une augmentation rapide du capital, et donc des intérêts associés qu'il nous faut rembourser au plus vite. Il arrive que sur les projets, pour des raisons de coûts ou de choix stratégique - proposer une fonctionnalité avant la concurrence aux clients par exemple – nous précipitons la livraison du logiciel. Nous nous retrouvons avec une augmentation rapide du volume de code, *le capital*, mais aussi des bugs, *les intérêts*, et donc de la dette technique. La **dette intentionnelle** est le résultat de décisions de livrer une ou des fonctionnalité(s) rapidement. Cette dernière représente ici un investissement et est effectivement justifiable. Mais nous devons prendre conscience qu'il est nécessaire de la résorber au plus tôt.

[1] *Code Smell : http://www.codinghorror.com/blog/2006/05/code-smells.html*

Les conséquences de la dette sur les projets logiciels

Le graphique suivant illustre l'augmentation de la dette technique à mesure qu'un projet grossit :

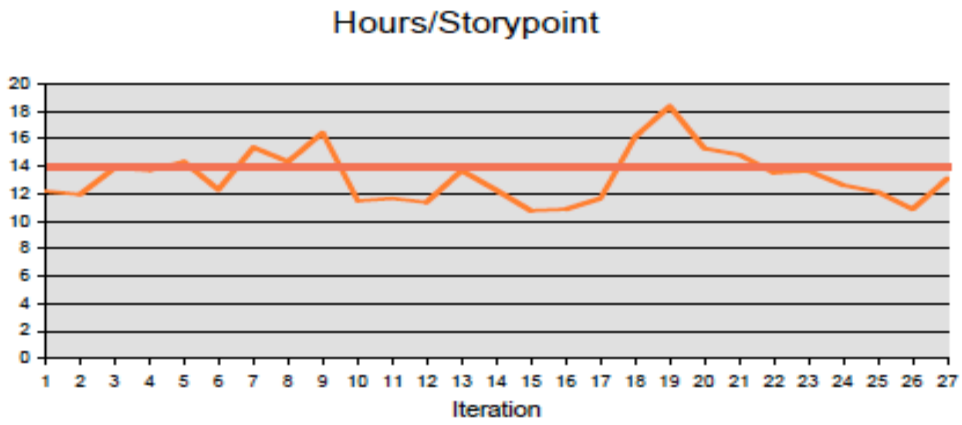


Evolution des coûts de changements

Avant toute chose, précisons qu'il n'est pas intuitif d'effectuer un lien de cause à effet entre les développements et la dette technique. Ce graphique illustre les conséquences de l'absence de gestion de la dette. Un constat saute aux yeux : le volume de code d'une application croit avec le temps. Les changements ou évolutions deviennent alors difficiles à mettre en place (courbe du haut), entraînant des coûts importants. Un « entretien » ponctuel de l'application (courbe du milieu) permet de les atténuer, tandis qu'un entretien quotidien (courbe du bas) les réduit. La zone se situant entre la courbe du haut et celle du bas permet d'identifier le surcoût induit par l'absence de gestion de la dette technique. Ce dernier devient de plus en plus important si nous n'effectuons aucun entretien.

Ce graphique illustre les conséquences de la dette technique mais ne représente en aucun cas une preuve tangible des coûts associés. Il existe cependant un outil permettant de mesurer la dette technique. Cet outil, issu des méthodes agiles, est **Scrum**, et plus précisément un de ses indicateurs : la **vélocité**. Elle représente la capacité d'une équipe à délivrer un certain nombre de fonctionnalités, exprimées en *Story Points*, pendant une période définie que nous appellerons *itération* par la suite. Les *Story Point*s représentent une unité de mesure relative pour connaître la complexité des fonctionnalités : plus elles sont difficiles ou longues à implémenter, plus les *Story Points* associées sont élevées.

Afin d'illustrer notre propos, étudions le graphique suivant :



Cette mesure, issue du retour d'expérience du projet Pro Rail (Fully Distributed Scrum: The Secret Sauce for Hyperproductive Offshored Development Teams[1]) présente l'évolution du nombre d'heures par *Story Point* au cours des *itérations*. Nous notons que l'équipe possède une référence de 14 heures par *Story Point*. En regardant plus en détail, nous remarquons que les *itérations* 9, et 19 sont marquées par une augmentation du temps passé sur les *Story Points.* Ceci a pour conséquence une **baisse de la vélocité** - toutes les fonctionnalités prévues n'ont pas été implémentées dans ces *itérations*. Le volume de code et sa complexité augmentant, le nombre de *Code Smell*, de bugs et de maintenances suit irrémédiablement. L'équipe se retrouve alors à gérer la dette, tout en continuant de nouvelles implémentations. Nous nous retrouvons dans la situation où le temps mis pour réaliser une fonctionnalité augmente. Le surplus de temps passé sur les *Story Point*s est le résultat d'une dette accumulée et non gérée. Lors des *itérations* 10 et 20 cette dernière a été en partie remboursée : le nombre d'heures par *Story Point* recolle à la référence et permet même dans certains cas de développer plus rapidement, ce que nous observons au début de l'*itération* 10.

A ce stade de nos explications, deux questions se profilent :

- Qu'entendons nous par gérer la dette technique ?
- Comment gérer la dette technique de sorte que le temps passé à implémenter nos *Story Point*s reste proche de notre référence ?

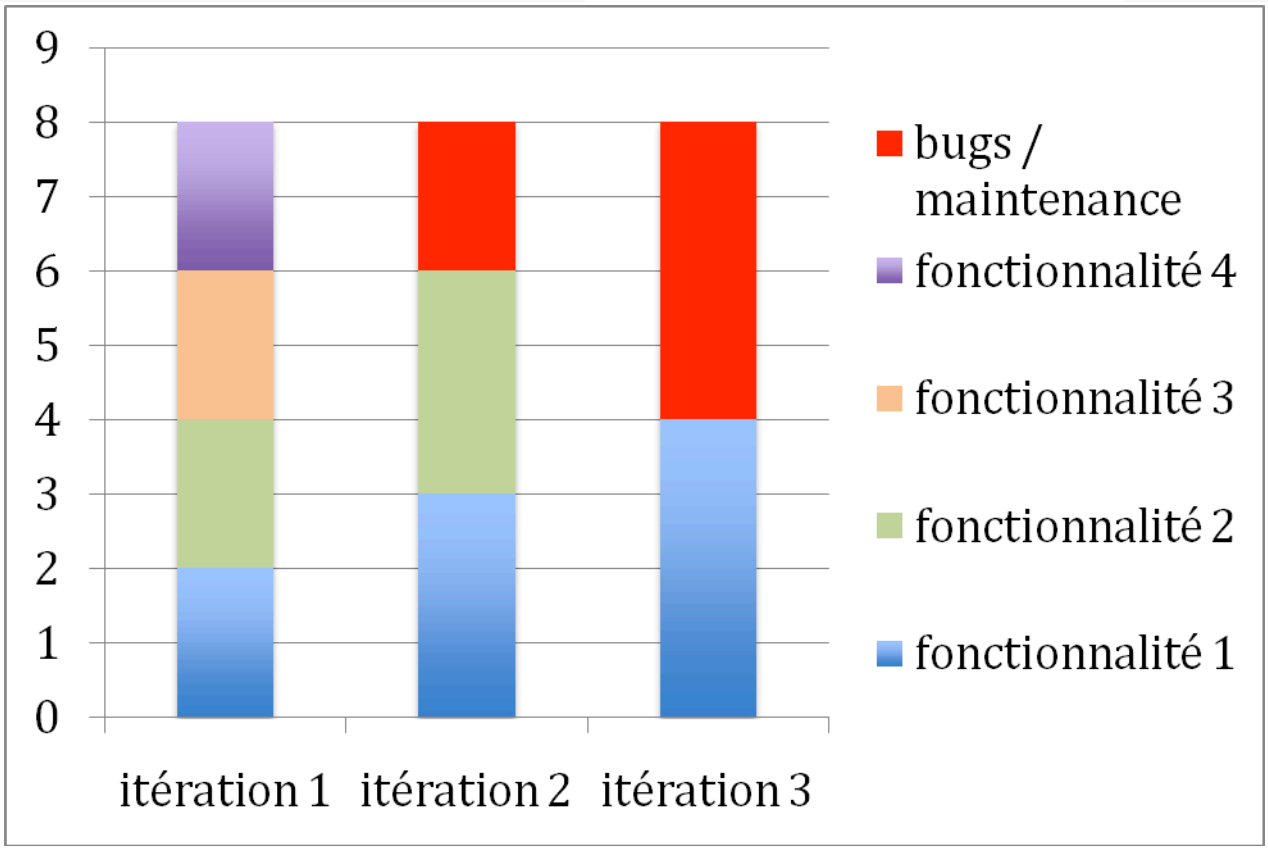
Ce sont à ces questions que nous tâcherons de répondre dans la seconde partie.

[1] *Fully Distributed Scrum: The Secret Sauce for Hyperproductive Offshored Development Teams :* *http://blog.xebia.com/2008/08/21/agile2008-fully-distributed-scrum/*

Gérer sa dette technique

Les mécanismes sous jacents de la dette technique

Les études et la pratique montrent que les projets logiciels évoluent beaucoup après leur première mise en production. Les équipes poursuivent continuellement les développements et les corrections d'anomalies.



Evolution des fonctionnalités par unité de temps

Ce graphique, simplifié pour l'étude, montre le développement de fonctionnalités - en unités de temps - par *itération*. Dans cet exemple, développer une fonctionnalité prend 2 unités de temps. Nous allons considérer qu'une *itération* comprend 8 unités de temps. De manière accélérée, nous voyons qu'au fur et à mesure des *itérations*, nous consacrons de plus en plus de temps aux corrections de bugs et à des opérations de maintenance : le nombre de fonctionnalités livrées par *itération* baisse constamment. Nous remarquons aussi que le développement de ces dernières nécessite plus de temps.

L'objectif de cette seconde partie est d'analyser en profondeur pourquoi le temps de réalisation d'une fonctionnalité augmente, et plus précisément d'étudier les mécanismes sous jacents créant ce surplus de temps. En premier lieu, intéressons nous aux principaux acteurs des projets logiciels : les équipes de développements et les responsables de projets.

Plaçons-nous dans une situation idéale : un nouveau projet sur lequel une équipe, sensibilisée au problème de la dette technique, décide d'adopter de bonnes pratiques de développement *(sur lesquelles nous reviendrons)* afin de la contrôler dès le départ. Cette situation, bien qu'envisageable, se rencontre malheureusement rarement. La plupart des équipes sont pilotées par des délais courts et de la pression, les obligeant à prendre une succession de décisions sans en mesurer nécessairement toutes les conséquences. Par exemple, développer moins de tests sur une fonctionnalité afin de livrer cette dernière plus rapidement est un cas de figure qui se présente fréquemment. Dans beaucoup de projets la phrase : « l'ajout de la fonctionnalité me prend beaucoup de temps, car le code n'est pas compréhensible et je n'ai pas le temps de tout refaire » est malheureusement courante. La plupart du temps la prise de conscience de la dette se fait dans un mode *catastrophe* : le volume « endetté » arrive à un point tel que le temps mis pour développer de nouvelles fonctionnalités est en constante augmentation, et nous ne pouvons plus l'ignorer.

De par sa nature, la résorption de la dette est rarement budgétée. Interviennent alors les responsables de projet, chargés de contrôler les cordons de la bourse. Leur expliquer la dette technique et ses conséquences peut se révéler une tâche ardue.

Notre principal objectif est de trouver un argumentaire pour démontrer à tous les acteurs des projets logiciels que ce mode dit de *catastrophe* cause une accumulation de la dette technique, entraînant des coûts supplémentaires et des pertes non négligeables. Pour cela nous choisissons de mener notre réflexion sur trois axes :

- La **maintenance**.
- L'**évolutivité**.
- La **fiabilité**.

Ces notions vont nous permettre de mettre en évidence les conséquences de ce mode *catastrophe* sur les projets logiciels.

La maintenance

En premier lieu, attardons nous sur la définition du terme *maintenance* trouvée sur Wikipédia :

> *« Selon la définition de l'AFNOR[1], la **maintenance** vise à maintenir ou à rétablir un bien dans un état spécifié afin que celui-ci soit en mesure d'assurer un service déterminé. »*

La *maintenance* fait référence à des équipements dont certaines parties sont à remplacer ou réparer. Selon les secteurs d'activités les types de maintenances diffèrent : par exemple, nous parlons de maintenance des machines dans l'industrie, souvent associée à la notion de dépannage. Dans le monde informatique, nous utilisons le terme de maintenance logicielle. Il s'agit en fait d'un abus de langage. Les notions de dépannage et réparation se retrouvent sous des formes différentes : maintenir un logiciel consiste à corriger des bugs et à effectuer des évolutions. Voici la définition de la maintenance logicielle que nous trouvons sur Wikipédia :

> *« En informatique logicielle, on divise la maintenance en plusieurs types :*
>
> - *la maintenance corrective[2] : elle consiste à corriger les défauts de fonctionnement ou les non-conformités d'un logiciel,*
> - *la maintenance adaptative[3] : sans changer la fonctionnalité du logiciel, elle consiste à adapter l'application afin que celle-ci continue de fonctionner sur des versions plus récentes des logiciels de base, voire à faire migrer l'application sur de nouveaux logiciels de base (un logiciel de base étant un logiciel requis pour l'exécution d'une application; exemples : système d'exploitation, système de gestion de base de données).*
>
> *On parle également de maintenance évolutive[4] : cela consiste à faire évoluer l'application en l'enrichissant de fonctions ou de modules supplémentaires, ou en remplaçant une fonction existante par une autre, voire en proposant une approche différente. Mais au sens de l'AFNOR, ce n'est même plus de la maintenance, puisque la maintenance consiste précisément à assurer qu'un bien continue de remplir sa fonction correctement, non à l'améliorer. »*

En d'autres termes nous constatons que pour les projets logiciels, le terme maintenance se découpe en trois parties. La *maintenance adaptative,* que nous ne développerons pas ici, concerne des problématiques d'exploitation. La *maintenance corrective* correspond à la correction de bugs et *la maintenance évolutive* correspond aux évolutions de fonctionnalités existantes. Ces deux dernières touchent directement les équipes de développement et nous intéressent plus particulièrement.

[1] *AFNOR : http://fr.wikipedia.org/wiki/AFNOR*

[2] *Maintenance corrective : http://fr.wikipedia.org/wiki/Maintenance_corrective*

[3] *Maintenance adaptative : http://fr.wikipedia.org/wiki/Maintenance_adaptative*

[4] *Maintenance évolutive : http://fr.wikipedia.org/wiki/Maintenance_%C3%A9volutive*

L'évolutivité

Comme nous venons de le voir, la définition de la *maintenance évolutive* est la suivante :

> *« Cela consiste à faire évoluer l'application en l'enrichissant de fonctions ou de modules supplémentaires, ou en remplaçant une fonction existante par une autre, voire en proposant une approche différente. »*

Cette définition montre que le terme *maintenance évolutive* est en contradiction avec la définition de l'AFNOR : « *ce n'est même plus de la maintenance, puisque la maintenance consiste précisément à assurer qu'un bien continue de remplir sa fonction correctement, non à l'améliorer* ».

Ainsi nous faisons le choix de ne pas utiliser le terme de *maintenance évolutive*, et de lui préférer le terme d'*évolutivité*. Ainsi, dans la suite du document nous parlerons de maintenance pour la *maintenance corrective*, et d'*évolutivité* qui sont des notions bien distinctes.

La fiabilité

Voici la définition que nous trouvons sur Wikipédia :

> *« Un système est **fiable** lorsque la probabilité de remplir sa mission sur une durée donnée correspond à celle spécifiée dans le cahier des charges. »*

Autrement dit, concernant les projets logiciels, lorsqu'une fonctionnalité ne fournit pas le comportement attendu, comme l'impossibilité de se connecter à une application, nous considérons que cette dernière est non fiable et que des actions de *maintenance* sont nécessaires.

Un manque de *fiabilité* peut avoir des conséquences néfastes sur les projets logiciels. Par exemple, des applications présentant des erreurs récurrentes. Ces dernières engendrent des demandes et plaintes de clients auprès des services de support logiciel. Ainsi, une *fiabilité* mal maîtrisée peut, à terme, entraîner un manque de compétitivité, une mauvaise image de marque avec à la clé des manques de revenus.

Trouver le bon compromis

Le temps consacré à la *maintenance* est du temps en moins sur les développements de nouvelles fonctionnalités. De même, un manque d'*évolutivité* entraîne des interventions sur le code plus longues et difficiles. Ce temps en moins ou rallongé pour les développements est péniblement rattrapé en introduisant une qualité moindre et donc une faible *fiabilité*.

L'idéal serait d'avoir des projets logiciels avec très peu de *maintenance*, des *évolutions* faciles et rapides à mettre en place, et une *fiabilité* excellente. A moins de nous trouver dans des secteurs particuliers tels que l'aéronautique, où les enjeux économiques et humains sont très importants, impliquant des niveaux d'exigences et de *fiabilité* très élevés, cela est rarement le cas. Les applications sur lesquelles nous intervenons auront toujours de la *maintenance*, des *évolutions* et des problèmes de *fiabilité*.

L'objectif est de minimiser la *maintenance* et de maximiser l'*évolutivité* ainsi que la *fiabilité*. Il n'y a malheureusement pas de réponse universelle, et prendre la décision de baisser le niveau de qualité des fonctionnalités afin de les livrer dans les délais exigés existera toujours. En revanche nous espérons que ce chapitre permettra aux responsables et équipes de développements de trouver de meilleurs compromis afin de garder un niveau suffisant de *fiabilité*. S'engager à livrer un nombre restreint de fonctionnalités lors d'une *itération* tout en gardant un niveau de qualité suffisant nous semble être un bon compromis.

Mesurer la maintenance, l'évolutivité et la fiabilité

Les projets logiciels sont soumis aux lois de l'entropie : le volume de code augmente entraînant de la complexité, donc de la *maintenance* et des évolutions plus difficiles à mettre en place. Ces dernières font parties des principales sources de la dette technique sur les projets logiciels, avec, pour conséquence l'augmentation du temps pour implémenter des fonctionnalités. A ce stade de notre raisonnement, nous pouvons nous poser la question suivante : pourquoi une *maintenance corrective* ou une *évolution* de fonctionnalité peut prendre beaucoup de temps ? La réponse dépend de la **tolérance aux changements**.

La *tolérance aux changements* est une expression traduisant le fait qu'intervenir sur des morceaux de code peut s'avérer plus ou moins difficile. La manière d'implémenter une fonctionnalité est différente selon les personnes : nous appréhendons et modélisons des solutions selon nos expériences. Dans une situation idéale, nous nous retrouvons avec du code compréhensible sur lequel nous pouvons effectuer des modifications, et des évolutions rapidement ; la *tolérance aux changements* est bonne. Dans le cas contraire nous sommes confrontés à du code difficile à comprendre, engendrant des *maintenances correctives* et des interventions plus longues à effectuer; la *tolérance aux changements* est mauvaise.

Nous constatons ainsi que cette dernière est fortement liée au code existant. Se posent alors deux questions :

- Comment pouvons-nous l'évaluer ?
- Comment produire du code ayant une bonne *tolérance aux changements* ?

Cela passe par la mise en place de bonnes pratiques de développement.

Les bonnes pratiques de développements et les outils

La tolérance au changement

Nous avons vu que les *maintenances correctives* et *l'évolutivité* dépendent de la *tolérance aux changements*. Selon nous cette dernière résulte de trois principaux attributs :

- La **conception**.
- La **lisibilité du code**.
- Les **tests**.

La conception

La *conception* est la manière de percevoir et de modéliser des idées et plus particulièrement des processus. Dans notre métier, la *conception* est intimement liée aux paradigmes des langages de programmation : *Orienté Objet*, *procédural*, *fonctionnel, etc*. Suivant ces derniers, la modélisation d'un processus prendra des formes différentes.

Intéressons nous plus particulièrement à la programmation *Orienté Objet* : de nombreuses pratiques existent et parmi elles nous retrouvons, par exemple : le principe de Liskov, le principe de ségrégation des interfaces, le principe d'injection de dépendances, etc. Citons également le *Domain Driven Design* (DDD) qui fournit des principes de *conception*. Bien appliqués, ces principes nous permettent de tirer profit de la programmation *Orientée Objet* et d'influer sur les notions de *rigidité* et de *fragilité* du code (Agile Software Development : Principles, Patterns, and Practices, Robert C. Martin).

- La **rigidité** d'un code correspond à la latitude laissée aux modifications et évolutions. Plus les interventions sont difficiles, plus nous considérons que le code est rigide : ce que nous pensons pouvoir modifier rapidement de manière ciblée se révèle être un travail titanesque, nous contraignant à étendre nos modifications sur plusieurs modules.

- La **fragilité** d'un code représente la difficulté à modifier ce dernier sans engendrer d'erreurs dans d'autres portions de code n'ayant aucun rapport avec la modification apportée. Par exemple, si nous modifions un service de connexion à une application et que le service de commande d'articles est impacté, cela signifie que le code de l'application est fragile.

La programmation *Orientée Objet* nous offre la possibilité de mettre en place du code peu rigide et peu fragile qui se traduit par un découpage cohérent et clair de l'application, nous permettant d'intervenir plus efficacement sur les modifications et les évolutions.

La lisibilité du code

La *lisibilité du code* fait référence à sa compréhension par des personnes autres que l'auteur. Dans les faits, nous sommes confrontés à du code qui n'est pas toujours clair ; les noms donnés aux méthodes ne reflètent pas forcément les traitements exécutés. Par exemple une méthode de valorisation de produits financiers peut très bien s'appeler valorisation de carottes. Les langages de programmation sont des interfaces pour dialoguer avec une machine : cette dernière réalise une tâche demandée, peu importe son nom. Autant pour une machine le nom d'un traitement n'est pas un problème, autant pour une personne cela peut l'être. Si une personne se retrouve confrontée à la méthode valorisation de carottes, sa compréhension demandera un certain effort. Bien entendu nous prenons un exemple simple pour illustrer notre propos, nous vous laissons imaginer ce que cela peut donner sur un ensemble de fonctionnalités. Un code lisible est plus facilement compréhensible, permettant des interventions plus rapides.

Les tests

Les *tests* sont les garants de la qualité de l'application. Effectuer des améliorations sur du code non testé peut avoir des impacts négatifs non souhaités sur les résultats attendus. Mettre en place des tests de non régression pour les ajouts ou modifications de codes nous donne l'avantage de savoir si ces interventions modifient les résultats escomptés. Si ces derniers diffèrent, nous savons que nos modifications n'ont pas correctement été mises en place et nous devons revoir notre copie. L'écriture d'un test prend autant, voire plus de temps que le développement de la fonctionnalité. Envisager tous les cas de figures et les implémenter est couteux. Mais ce temps n'est pas négociable : les tests sont indispensables. Ils nous garantissent des fonctionnalités fiables avec les comportements attendus, et nous permettent de réduire les bugs.

Une pratique spécifique existe : le *Test Driven Development* (*TDD*). Le *TDD* consiste à coder les tests avant d'effectuer l'implémentation. En pratique nous exécutons le premier test, le résultat est un échec. Nous implémentons alors la méthode répondant à ce dernier. Nous recommençons alors le processus pour les autres tests. Obscure pour certains, difficilement concevable pour d'autres, cette pratique présente l'avantage d'aller à l'essentiel pendant les implémentations, et de mieux cibler les développements avec moins de *Code Smell*.

Les bonnes pratiques

A travers les notions de *conception*, de *lisibilité du code* et de *tests*, nous pouvons juger de la *tolérance aux changements* d'un code. Dans l'optique d'améliorer cette dernière, un ensemble de bonnes pratiques de développements peuvent être mises en place :

- Le **refactoring**.
- Le **pair programming**.
- La **revue de code**.

Une partie d'entre elles sont issues de l'Extreme Programming (XP)[1] et visent à améliorer au quotidien la *conception* et la *lisibilité du code*.

Le refactoring

Le *refactoring* consiste à revoir, de manière continue, la *conception* et la *lisibilité du code* en vue de son amélioration. Lorsque nous achevons le développement d'une fonctionnalité, revenir sur le code pour éliminer des *Code Smell* et améliorer son design nous semble être une bonne approche. Nous « refactorons » régulièrement du code existant, et modifier ou améliorer ce dernier se révèle souvent très difficile. Dans cette situation la présence de tests se révèle un atout indispensable : intervenir sur du code existant devient alors possible, avec l'assurance de ne pas introduire de régressions.

Tous les projets nécessitent du *refactoring* plus ou moins profond, avec plus ou moins d'impacts : des *refactorings* ciblés peuvent se faire pendant les développements, tandis que des *refactorings* susceptibles d'impacter le cœur de l'application nécessitent des stratégies plus élaborées. Nous reviendrons sur ces stratégies dans la prochaine partie de ce document. Un certain nombre de méthodes permettent d'identifier et d'isoler des parties de code à désendetter, nous donnant ainsi l'avantage, non négligeable, de maîtriser notre dette.

[1] *Extreme Programming (XP) : http://fr.wikipedia.org/wiki/Extreme_programming*

Le pair programming

Le *pair programming* consiste à développer à deux derrière un seul écran. Souvent critiqué, le *pair programming* fait débat : quel(s) intérêt(s) avons nous à développer une fonctionnalité à deux, alors qu'un seul développeur parviendra également à la mettre en place ? Si nous raisonnons simplement, nous pouvons effectivement faire le constat suivant : deux personnes travaillent sur une même tâche, alors qu'elles pourraient chacune réaliser des développements différents. De ce fait, les implémentations sont moins rapides. Si nous poussons notre réflexion, une fonctionnalité développée par une personne représente un certain coût. Dans la pratique du *pair programming*, le développement d'une fonctionnalité a nécessité deux personnes : mathématiquement le coût de cette dernière est double.

En pratique, le constat est tout autre ; certes le coût de la fonctionnalité est plus élevé, mais les bénéfices sont importants. Tout d'abord en terme de qualité : notre code contient moins de *Code Smell* et est plus *lisible*. Le second développeur garde un œil attentif sur le développement en cours et peut intervenir immédiatement lorsqu'une objection ou une incompréhension surgit. Ensuite en terme de maintenance : une fonctionnalité développée par une seule personne a plus de chance de contenir des bugs contrairement à la même développée par deux personnes. Les échanges et connaissances de chacun des développeurs permettent d'aboutir à des implémentations plus efficaces : deux cerveaux valent mieux qu'un ! Enfin en terme de partage de la connaissance, travailler avec une personne ayant de solides connaissances permet à son binôme de s'approprier son savoir. Par exemple, des profils juniors travaillant avec des profils plus expérimentés gagneront en compétences tant au niveau des connaissances fonctionnelles qu'au niveau des méthodes et des pratiques de développements.

La revue de code

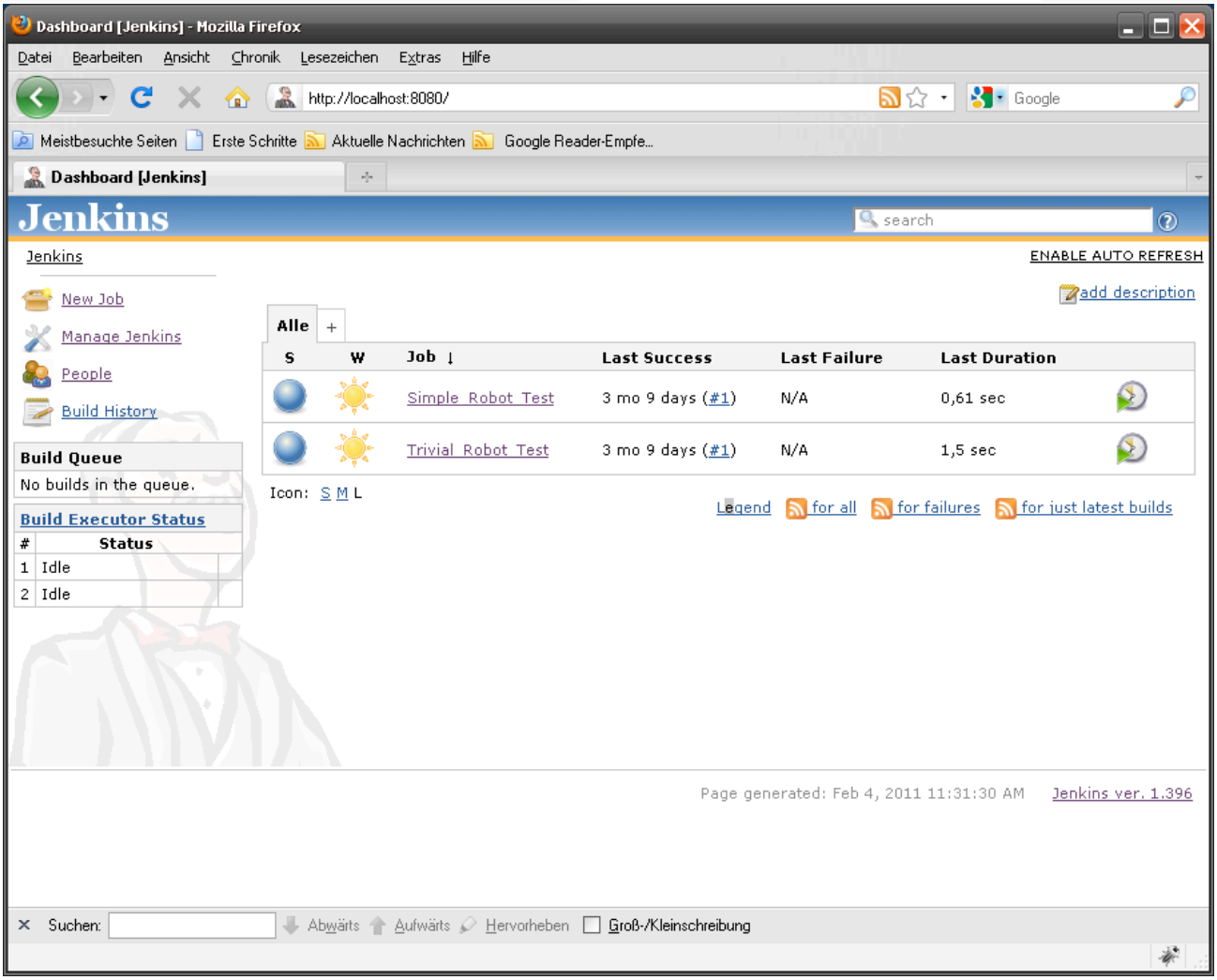
La *revue de code* consiste à discuter et confronter les choix effectués, aussi bien pour la *conception* que pour l'implémentation d'une fonctionnalité. Etroitement liée au *pair programming*, la *revue de code* en diffère dans sa mise en pratique : en premier lieu, elle intervient généralement à la fin du développement d'une fonctionnalité ou bien à la fin d'une *itération*. En second lieu elle s'effectue avec, de préférence, tous les membres de l'équipe. La *revue de code* peut être un substitut à la pratique du *pair programming*, dans certaines situations où nous ne pouvons pas le mettre en place. Imaginons que nous sommes dans une équipe constituée d'un nombre impair de développeurs. Une personne sera alors amenée à développer seule. Pour pallier à cette situation, la *revue de code* devient alors indispensable : expliquer ses choix de *conception* et d'implémentation se fait devant tous les membres de l'équipe, et les éventuelles corrections ou améliorations ont lieu immédiatement, ainsi que le partage de la connaissance. La *revue de code* est aussi un bon complément de la pratique du *pair programming* : imaginons que deux développeurs travaillant en *pair* mettent en place un algorithme générique et qu'il est prévu que ce dernier soit la base de futurs développements. La *revue de code* est alors un bon moyen pour présenter et partager les détails d'implémentations aux autres membres de l'équipe.

> Ces bonnes pratiques ne sont pas les seuls « outils » que nous avons à notre disposition. Tout comme la pratique de l'escalade, le fait de posséder de bonnes techniques ne nous suffira pas à grimper efficacement. De bons outils sont également nécessaires pour nous aider à atteindre notre objectif d'amoindrissement de la dette.

Les outils

Les plateformes d'intégration continue

Une plateforme d'*intégration continue* est chargée de construire les projets, c'est- à-dire compiler, déployer et exécuter les tests régulièrement. Ce type d'outil permet de connaître à tout moment l'état de santé de nos applications, en détectant les éventuelles régressions. Parmi ces outils, nous retrouvons Jenkins :



Interface de l'outil Jenkins

Jenkins nous indique, à travers des indicateurs de couleurs, les projets en bonne et mauvaise santé. Par mauvaise santé, nous entendons des erreurs de compilations, des échecs de tests, etc. Nous noterons que la mise en œuvre d'une plateforme d'*intégration continue* ne se suffit pas à elle même : elle doit s'accompagner de la mise en place de bonnes pratiques. Parmi ces dernières nous avons la notion de *commits* réguliers. Un *commit* est l'action de placer des fichiers modifiés dans un gestionnaire de version, qui est généralement une machine dédiée à la sauvegarde de notre code source. A chaque *commit*, l'outil d'*intégration continue* va démarrer une compilation et l'exécution des tests. Cela nous permet d'avoir des retours réguliers et rapides : échecs de tests, erreurs de compilations, etc, et d'intervenir sur les parties modifiées immédiatement. Conséquence de ces *commits* réguliers, les phases de compilations et d'exécutions des tests doivent être courtes : si ces dernières sont longues, il y a de fortes chances pour que nous ayons commencé à travailler sur de nouvelles fonctionnalités avant d'avoir le retour de *l'intégration continue*.

Une autre bonne pratique concerne la gestion des erreurs : lorsque des erreurs de compilation ou d'exécution de tests apparaissent, la priorité des personnes ayant effectuées les modifications doit être de corriger ces erreurs afin de remettre le projet en bon état. Ce mode de fonctionnement est indispensable : imaginons que d'autres développeurs veuillent récupérer du code du gestionnaire de version, ou bien déposer leurs propres modifications. Etant donné que l'*intégration continue* nous a notifié que le projet n'est pas stable, les autres développeurs ne peuvent pas effectuer leurs actions, et doivent patienter jusqu'au retour d'un état stable du projet.

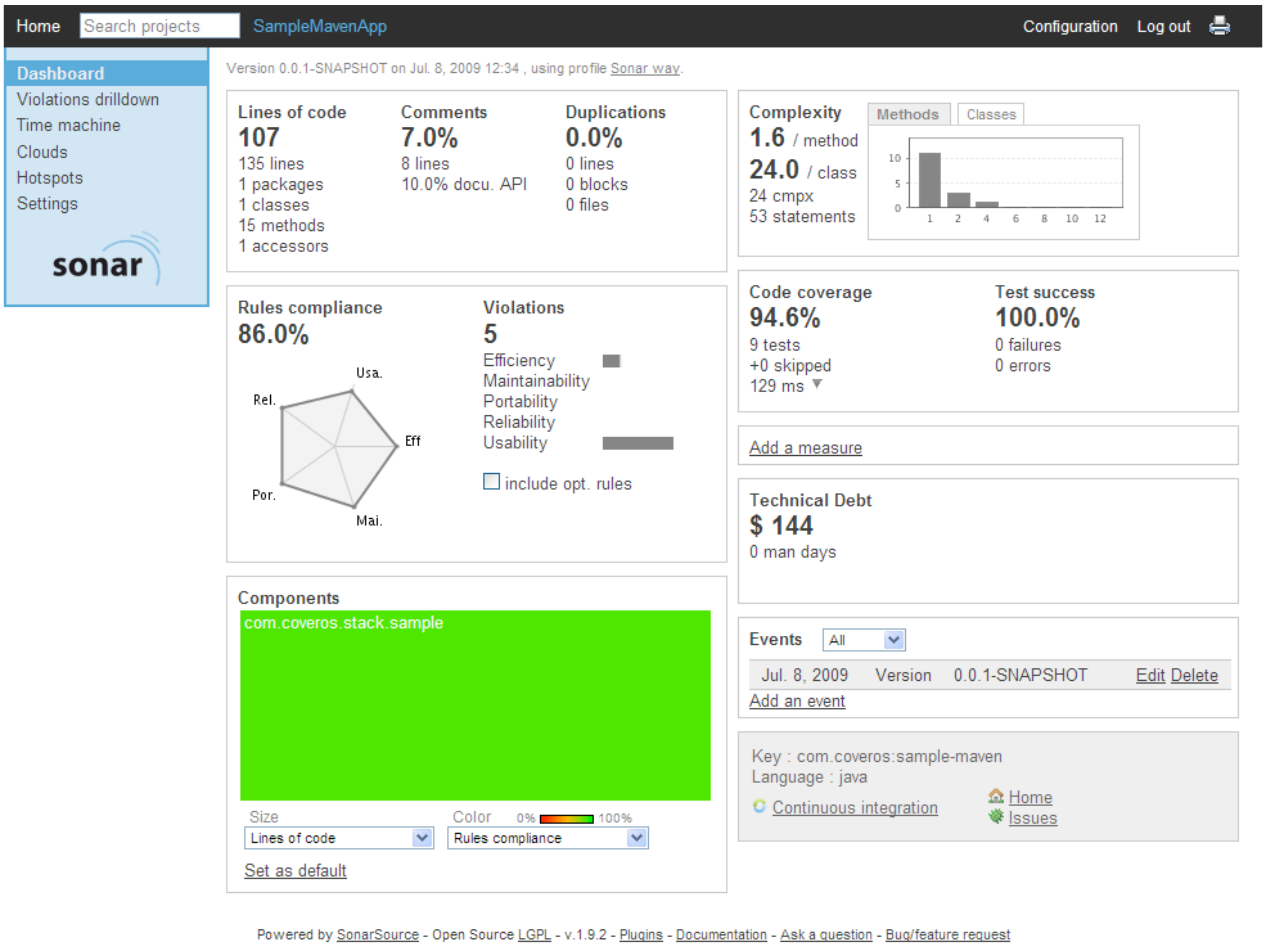
> Un outil d'*intégration continue* est incontournable : utilisé convenablement avec des *commits* réguliers, des compilations et exécutions de tests rapides, une gestion des erreurs, etc, elle représente notre principal garant de la *fiabilité* de nos applications en nous protégeant contre d'éventuelles régressions.

Les outils de mesure de la qualité

La qualité reste une notion assez floue concernant les projets logiciels. Dans les précédentes parties, nous avons expliqué qu'avoir une application avec peu de *maintenances*, une *évolutivité* et une *fiabilité* importantes représente un gage de qualité. Mais mesurer ces notions, en terme de coûts et de temps reste très difficile. Pour approfondir le sujet, nous vous conseillons le livre blanc de *Frédéric Dubois* et *Séven Le Mesle* sur la *qualité logicielle*[1]. Cette réflexion nous amène à considérer que nous n'allons pas « mesurer la qualité », mais choisir un certain nombre d'indicateurs nous permettant d'atteindre un faible niveau de *maintenance*, ainsi qu'une meilleure *évolutivité* et *fiabilité*.

Dans ce domaine, l'outil le plus connu et sans doute le plus utilisé est Sonar. Cet outil met à disposition un ensemble d'indicateurs, nous permettant de dresser un carnet de santé de nos projets. Beaucoup de ces indicateurs représentent de la dette technique contractée sur nos applications. Par exemple, les portions de code non utilisées, les erreurs mal gérées, la couverture des tests qui est un ratio entre la quantité de codes et les fonctionnalités testées, et bien d'autres.

[1] *Livre blanc de Frédéric Dubois et Séven Le Mesle sur la qualité logicielle :* http://blog.xebia.fr/2010/12/21/livre-blanc-qualite-logicielle/



Interface de l'outil Sonar

Comme nous pouvons le voir sur cette illustration, nous avons même une partie concernant la dette technique. Définie selon des critères propres à l'outil, cette dernière, exprimée en dollars, représente le montant que nous coûtent les erreurs liées aux indicateurs que nous avons choisis de mettre en place. Par exemple une portion de code non utilisée aura un certain coût, et l'outil est capable d'associer un montant à cet indicateur. Pour aller plus loin, nous vous invitons à consulter le site de Sonar afin de connaître les possibilités que vous offre cet outil.

Sonar nous aide à identifier et surtout à mesurer la dette de nos applications. Par exemple, certains projets exigent que la couverture des tests soit au minimum de 85%. De même des erreurs bloquantes et critiques, pouvant entraîner de gros problèmes d'exécutions ou des crashs de nos applications, sont à corriger très rapidement. Associé à un outil d'*intégration continue*, Sonar nous apporte une garantie de qualité supplémentaire. Il est par exemple possible de définir certaines règles au sein de l'équipe : admettons que nous voulons que la couverture des tests soit de 70%, mais que suite à de récentes modifications cette dernière soit descendue à 65%. Un mécanisme de notification nous avertira de cette baisse, et la priorité de l'équipe sera de remettre ce niveau de couverture à 70%.

Le principal avantage de cet outil réside dans la pertinence de ses mesures et la définition de règles : par exemple nous décidons de ne pas avoir plus de 10% de code non utilisé dans l'application. L'évolution des indicateurs nous permet de déterminer si la dette technique s'accumule ou se résorbe.

L'utilisation de bonnes pratiques et d'outils, au quotidien, nous permet d'obtenir une meilleure *tolérance aux changements* avec les bénéfices précédemment cités : des applications avec peu de *maintenances*, ainsi qu'une *évolutivité* et une *fiabilité* renforcées. Mais les résultats de l'application de ces bonnes pratiques ne sont pas immédiats. Leur mise en place peut nécessiter un certain temps, et les résultats escomptés s'inscrivent sur du long terme.

A ce stade de notre réflexion, deux questions subsistent :

- A quel moment devons nous gérer la dette technique ?
- Comment pouvons nous appliquer ces bonnes pratiques de développement quotidiennement à la gestion de la dette technique ?

Ce sont à ces questions que nous nous efforcerons de répondre dans la prochaine partie.

Mise en pratique

Prenons l'exemple de la construction et de la rénovation d'une maison. Dans le premier cas nous considérons que nous avons en notre possession les plans et que nous maîtrisons les savoir-faire nécessaires comme monter des parpaings, faire de la plomberie, de l'électricité, etc. En appliquant nos connaissances dès le début, notre construction se déroule sans problèmes majeurs et respecte les normes imposées, nous prémunissant d'éventuels problèmes. Dans le deuxième cas nous achetons une vieille maison et nous souhaitons effectuer des travaux avec en notre possession les savoir-faire adéquats. Restaurer une maison n'est pas trivial : la mise en pratique de nos connaissances nécessite des stratégies. Par exemple rénover un mur porteur ne peut pas se faire de n'importe quelle manière : ce dernier est une pièce maîtresse de la structure d'une maison. Il supporte le poids du plafond, de la toiture, etc. Si nous ne rénovons pas ce dernier correctement ou bien que pendant sa restauration nous effectuons de mauvaises manipulations, il est possible que notre habitation ne tienne pas longtemps debout. D'où la nécessité d'élaborer des stratégies tout en appliquant nos savoir-faire.

Il en va de même pour les projets informatiques. Nous avons un premier cas de figure, idéal, où un nouveau projet démarre. Les membres de l'équipe connaissent les bonnes pratiques de développements et les appliquent dès le début. Le projet bénéficiera immédiatement des résultats escomptés. Le second cas de figure nous intéresse plus : nous passons beaucoup plus de temps à modifier ou à faire évoluer des projets qu'à en créer. Mais l'application des bonnes pratiques de développement n'est pas triviale : par exemple effectuer des *refactorings* sur du code existant peut engendrer des changements de comportement ou bien des modifications de résultats. Ces *refactorings* nécessitent des stratégies.

A quel moment devons nous gérer la dette technique ?

La gestion de la dette technique est un travail quotidien. L'intégrer dans les cycles de développement au travers des bonnes pratiques que sont le *pair programming*, le *refactoring*, les *tests unitaires*, etc, nous donne l'avantage de contrôler l'accumulation de la dette. Attention cependant, l'application des bonnes pratiques est une stratégie à long terme : les résultats ne seront visibles qu'au bout de plusieurs *itérations*. Par exemple, travailler en *pair programming*, effectuer des *revues de code*, etc, sont de nouvelles disciplines que nous devons acquérir et leur mise en place demande du temps. Bien entendu il n'est pas nécessaire de maîtriser ces bonnes pratiques pour avoir des résultats, mais ces derniers seront maximaux si nous les maîtrisons.

Les stratégies de refactoring et de test

Les notions de refactoring et de test nous semblent être les bases pour réduire la dette technique efficacement : nous les retrouvons dans les autres bonnes pratiques. Le *pair programming* ou les *revues de code*, les discussions sur la *conception*, nous amènent souvent à refactorer le code. Ensuite, ces deux notions sont fortement liées : il n'est pas concevable de refactorer une portion de code sans la présence de tests. Sans ces derniers, nous n'avons aucunes garanties de la *fiabilité* concernant d'éventuelles régressions introduites.

Lorsque nous prenons la décision de refactorer des portions de code, il nous faut distinguer trois types de refactoring que nous qualifierons de *mineurs*, *majeurs* et *de longue durée*.

Refactorings mineurs

Les refactorings mineurs correspondent à des modifications n'ayant pas ou peu de conséquences sur les autres portions de codes de l'application. Cette catégorie regroupe entre autres les renommages, les duplications, les factorisations, ou encore la documentation du code.

Il est recommandé d'intégrer ces refactorings dans les phases de développement. Par exemple pendant un développement nous réalisons que deux traitements utilisent des portions de code identiques : le refactoring consiste à mutualiser ce code à un seul endroit et de faire référence à ce dernier à chaque fois que nous aurons besoin d'effectuer le même traitement. Deux arguments plaident en faveur de l'adoption d'une stratégie au fil de l'eau pour les refactorings mineurs :

- Cela permet, d'aboutir à du code plus lisible, compréhensible et évolutif (et donc d'introduire moins de dette).
- Le support de la majorité des techniques de *refactorings mineurs* par les environnements de développements nous apporte une garantie sur les modifications, se traduisant par peu ou pas d'erreurs.

Refactorings majeurs

Les *refactorings majeurs* concernent des modifications de code pouvant grandement impacter d'autres portions de codes, avec potentiellement des conséquences sur le comportement de l'application. Dans cette catégorie, nous retrouvons les changements de système comme par exemple une migration de bases de données, de technologie d'affichage d'une page Web, ou bien encore le design et la conception du cœur de l'application.

Seulement nous ne pouvons pas prendre le risque de mettre en péril une application : les objectifs que nous cherchons à atteindre en terme de *maintenance*, *évolutivité* et *fiabilité* pourraient ne pas être au rendez vous. Des stratégies plus fines sont de rigueurs. Prenons l'exemple d'une application dont le cœur de métier concerne la gestion de produits financiers. Un certain nombre d'algorithmes mathématiques permettent de calculer des formules financières complexes. Certains de ces algorithmes sont « génériques » et utilisés pour les calculs de plus d'une dizaine de produits financiers. Imaginons que des experts financiers découvrent une incohérence dans les résultats de ces calculs. Après quelques recherches nous nous apercevons de la cause de ces erreurs : un des algorithmes ne supporte pas un produit financier particulier. Ce dernier doit alors être, en partie, revu. La question qui va alors se poser est de savoir quelle stratégie nous allons mettre en place pour ne pas avoir de régressions sur les résultats existants tout en ajoutant un nouveau comportement.

- Quand nous disposons de tests :

  Deux possibilités s'offrent à nous :

  - Soit nous effectuons les modifications adéquates dans l'algorithme, et nous ajoutons des tests.
  - Soit nous suivons une approche TDD (*Test Driven Development*) : nous ajoutons des tests, et nous effectuons nos modifications sur l'algorithme afin de pouvoir exécuter ces derniers.

  Choisir la première ou la seconde possibilité est une question d'habitude et de style. Dans notre situation, retenons que l'avantage d'avoir des tests existants et d'en ajouter est de pouvoir détecter les éventuelles régressions sur les calculs existants et de garantir la *fiabilité* de nos modifications.

- Quand nous ne disposons pas de tests :

  Apporter les modifications sur l'algorithme n'est pas envisageable : étant donné qu'aucun test n'est présent, rien ne nous garantit que les résultats des calculs existants ne sont pas modifiés. Dans ce contexte, une stratégie consiste à ajouter ou compléter des tests sur les calculs existants : nous nous assurons que l'ensemble des résultats de calculs soient fiables. Nous pouvons ensuite sereinement ajouter nos modifications et nos tests concernant le support du nouveau produit financier. Attention cependant, un manque de connaissances fonctionnelles ou un manque de documentation peut rendre difficile l'ajout de tests sur des portions de codes assez anciennes. Cela s'accentue selon le niveau de *tolérance aux changements*. Au sujet de l'ajout de tests sur du code existant, nous vous recommandons la lecture du livre *Working with Legacy Code*, afin de mieux appréhender ces problématiques.

- Quand il est nécessaire de réécrire le code de zéro :

  Lorsque le code n'est pas compréhensible et qu'intervenir dessus s'avère être un travail colossal, il est préférable d'en revoir la logique. En appliquant les *tests, le pair programming, la revue de code, la documentation,* etc, nous pouvons repartir sur une base de code saine. Nous bénéficions ainsi du partage de la connaissance au sein de l'équipe, et surtout d'avoir du code avec une *maintenance* faible ainsi qu'une meilleure *évolutivité* et *fiabilité*.

Refactorings de longue durée

Un dernier cas de figure concerne les *refactorings* de longues durées : plusieurs semaines, voire des mois. Par exemple nous décidons de modifier une technologie dans notre application, nécessitant la migration du code existant. Comment faire en sorte de continuer à travailler sur les développements de nouvelles fonctionnalités, tout en effectuant ces migrations ? Une réponse possible se situe au niveau de l'utilisation d'un outil de gestion de versions. Un tel outil nous permet de sauvegarder différentes versions de notre applications : par exemple nous pouvons sauvegarder une version de Juin 2010, une autre de Décembre 2010 et en cas de besoin, récupérer l'une ou l'autre. Lors des développements en cours nous travaillons, en général, sur la version courante de l'application qui est communément appelée copie principale. Les outils de gestion de versions nous permettent de travailler sur des copies différentes, et une fois que les développements sont terminés, de réinjecter les modifications sur la copie principale. Nous pouvons effectuer nos migrations et lorsque nous jugeons ces dernières fiables, nous les intégrons dans la copie principale. L'avantage de cette manière de procéder est que pendant ces refactorings, la version courante de l'application reste stable pour effectuer d'autres développements.

> Plusieurs stratégies existent. En fonction de la taille et de l'importance des refactorings, une stratégie sera plus efficace qu'une autre et vice versa. Gardons également en mémoire que des refactorings ne peuvent s'effectuer sans tests au préalable. Les stratégies, conjuguées aux bonnes pratiques, nous permettent d'atteindre l'objectif de mieux maîtriser la dette technique : réduire la *maintenance*, augmenter *l'évolutivité* et la *fiabilité* de nos applications.

Conclusion

L'évolution naturelle des projets logiciels engendre nécessairement une augmentation de leur complexité. Cette augmentation de complexité se traduit par des coûts croissants, et parfois cachés, de réalisation. C'est ce phénomène que traduit la notion de **dette technique**. Lorsque cette dernière devient trop importante, les conséquences se font lourdement sentir sur un projet :

- Augmentation drastique du nombre de maintenances correctives.
- Difficulté croissante à mettre en place des évolutions.
- Manque de fiabilité.

Comprendre ces mécanismes est indispensable pour pouvoir gérer la dette technique efficacement. Ces trois notions nous permettent d'expliquer en grande partie les retards de développements, et donc les coûts supplémentaires dépassant de loin les budgets initiaux. Sur du long terme, cela peut entraîner un manque de compétitivité avec des potentielles pertes de revenus.

Il devient alors nécessaire d'agir sur ces symptômes.

Ces derniers sont directement liés à la notion de **tolérance aux changements**, que nous pouvons évaluer au travers de trois critères :

- La conception du code.
- La lisibilité du code.
- Les tests.

Pour ce faire, il convient de mettre en œuvre un ensemble de bonnes pratiques parmi lesquelles :

- Le refactoring.
- Le pair programming.
- La revue de code.

L'utilisation d'outils d'intégration continue et de mesure de la qualité nous facilitera cette tâche.

Ces disciplines s'acquièrent avec du temps et de l'expérience, et avoir des ressources avec ces compétences constitue des atouts indéniables sur les projets logiciels.

Associés à des stratégies adaptées telles que des refactorings impactant le cœur d'une application, ces bonnes pratiques et outils constituent nos principaux atouts dans l'amélioration des critères de *maintenabilité*, *évolutivité* et *fiabilité*, nous permettant ainsi de maîtriser notre **dette technique**.

L'apparition et l'accumulation de **dette technique** sur un projet est inévitable et nous pouvons même la qualifier de « toxique » : la supprimer totalement est impossible. Cependant il est indispensable de la gérer afin de la maintenir à un niveau acceptable qui nous garantit une application **maintenable**, **évolutive**, et **fiable**. Mais ce gage de qualité a un prix que nous ne devons pas non plus négliger. Cela demande des efforts et une rigueur constante. Dans cette optique, un compromis subtil est nécessaire - tout au long de la vie d'un projet - entre la quantité de fonctionnalités livrées et la qualité des réalisations.

Annexes

L'auteur : Nicolas Jozwiak

Nicolas est un ingénieur d'études confirmé disposant de 5 ans d'expérience en conception et développement sur les technologies Java/JEE. Son parcours chez un éditeur avant son entrée chez Xebia lui a notamment permis de développer de solides compétences dans le domaine de la qualité et de l'industrialisation (tests, intégration continue, gestion de configuration, contrôle qualité).

Bénéficiant d'une expérience très solide de mise en place des méthodes agiles et d'accompagnement d'équipes sur le terrain, il s'attache à mettre à profit quotidiennement son expérience qui est reconnue pour son approche pragmatique, proactive et pédagogique.

Remerciements

Je tiens tout particulièrement à remercier Guillaume Bodet, Directeur de la technologie de Finance Active, qui a contribué à la qualité de ce livre blanc de part son expérience ainsi que ses précieux conseils sur le sujet de la dette technique.

Je remercie également Frédéric Dubois et Christophe Heubès, de Xebia, qui ont bien voulu relire ce livre blanc et me faire profiter de leurs expériences.

Je tiens aussi à remercier toutes les personnes qui ont contribué par leurs remarques et suggestions à la qualité globale de ce livre blanc.

Références

- Kyle Brown - Paying back technical debt, 2010
  http://www.ibm.com/developerworks/websphere/techjournal/1001_col_brown/1001_col_brown.html
- Martin Fowler – Technical Debt, 2009
  http://martinfowler.com/bliki/TechnicalDebt.html
- Steve McConnell – Technical Debt, 2007
  http://blogs.construx.com/blogs/stevemcc/archive/2007/11/01/technical-debt-2.aspx
- Tom Brazier - Managing Technical Debt, 2007
  http://accu.org/index.php/journals/1301
- Kane Mar – Technical Debt and Design Death, 2006
  http://www.scrumalliance.org/articles/14-technical-debt-and-design-death
- Principle of OOD
  http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod
- Working Effectively with Legacy Code, Michael Feathers, 2004