Livre blanc qualité logicielle,

Pragmatisme et productivité

Xebia

Xebia
SOFTWARE DEVELOPMENT DONE RIGHT

Préface

« A l'instar des guildes néerlandaises de la fin du moyen âge, notre industrie s'organise autour de communautés d'intérêt, toutes porteuses de valeurs et de bonnes pratiques. Ces regroupements de professionnels ont tous pour vocation de faire progresser la collectivité et accompagner ainsi le long, très long, trop long même, mouvement de maturation de l'industrie logicielle.

Le logiciel n'a jamais été aussi important pour les entreprises : Amazon, FaceBook, Apple, eBay pour ne citer qu'elles, sont des entreprises qui ont assis leur domination grâce à l'avantage concurrentiel que leur procuraient les logiciels qui sous tendent leurs stratégies. Elles ont compris, bien avant les autres, l'importance de la qualité logicielle. Toute défaillance engendrant une perte directe de chiffre d'affaires.

Chez Amazon par exemple, on a habitude de dire « You Build it, You Run it » car là bas, l'exploitabilité des applications est une exigence dès la conception.

Après la naissance de groupes d'experts Java s'attachant à la connaissance des technologies et d'agilistes intéressés par les méthodologies, nous assistons depuis quelques temps à la naissance d'un groupe particulier de professionnels du développement logiciel, les Software Crafstmen qui prônent, pour leur part, l'adoption d'une attitude particulière vis-à-vis de la qualité.

Ces Craftsmen, ou artisans en français, sont des praticiens aguerris. Humbles, ils sont mus par le désir permanent de s'améliorer et souhaitent transmettre avec générosité, leur amour de la qualité logicielle aux moins expérimentés.

Ce livre Blanc a été écrit par deux Craftsmen »

Luc Legardeur
Président
Xebia IT Architects SAS

cc BY NC ND

Xebia
SOFTWARE DEVELOPMENT DONE RIGHT

Table des matières

Xebia
SOFTWARE DEVELOPMENT DONE RIGHT

1 La qualité logicielle est une notion évanescente

Objet du désir, elle est régulièrement invoquée sur le mode incantatoire au démarrage d'un projet de développement logiciel. Assurance qualité, direction qualité, responsable qualité, qualimétrie, processus unifié, modèle de maturité… nombreux sont les dispositifs visant à garantir que le résultat du développement sera source de fierté pour ceux qui l'ont conçu, de contentement pour ceux qui l'ont financé, de soulagement pour ceux qui devront le maintenir et de satisfaction pour ceux qui l'utilisent et l'exploitent.

Mais aussi nombreux que soient ces dispositifs, l'état des lieux en la matière en ce début des années 2010 est pour le moins mitigé. Xebia, pour avoir eu l'occasion d'auditer des dizaines d'applications, est bien placée pour le savoir : le logiciel de qualité reste l'exception. Pire : une rareté, pour ne pas dire une curiosité.

Cette évanescence de la qualité logicielle peut sembler un phénomène mystérieux. Après tout, on pourrait s'attendre, après un demi-siècle d'expérimentations et d'innovations innombrables, que le développement logiciel ait atteint une forme de maturité. La réalité est toute autre, et le développement logiciel souffre aujourd'hui comme hier d'une image délétère : lancer un projet reste pour beaucoup une opération pénible, risquée, coûteuse, incertaine et aux résultats trop souvent décevants.

Cette situation n'est pas, selon nous, une fatalité. Nous pensons qu'au prix d'un renforcement de la discipline de développement, et de la mise en place rigoureuse de certaines pratiques d'ingénierie, le niveau de la qualité logicielle peut être largement renforcé, et maintenu. Ces pratiques n'ont rien de révolutionnaire – la plupart sont décrites depuis des décennies, et appliquées avec succès par un nombre croissant d'équipes de développement : tests automatisés, développement incrémental, développement piloté par les tests, intégration continue, conception simple, refactoring, programmation en binôme... Ces pratiques ont toutes quelque chose en commun : elles aboutissent chacune à fournir dès que possible un retour sur le produit du développement. En d'autres termes, elles permettent de confronter minute après minute, heure après heure, jour après jour, semaine après semaine, le logiciel aux différentes exigences et contraintes qui le sous-tendent – et de le corriger en conséquence.

Ces pratiques, relèvent également d'une approche de la qualité sensiblement différente de celle qui a prévalu au cours des trois dernières décennies. La qualité n'est pas ici fonction du processus de fabrication du logiciel, mais bel et bien de la discipline et de la compétence de ceux qui sont chargés de le concevoir et de le programmer – en premier lieu les développeurs et les testeurs. Elle est le fruit de cette discipline, et non d'un dispositif de contrôle externe. Elle est donc, en dernière analyse, l'affaire des hommes et de femmes engagés dans le projet.

Ce document est articulé en deux parties – la seconde nettement plus longue que la première. Dans un premier temps, nous chercherons à définir ce qu'est la qualité logicielle, et à analyser les raisons de son évanescence. Ensuite, nous vous décrirons un ensemble de pratiques d'ingénierie qui, selon nous, et appliquées de façon systématique, permettent d'écrire, à moindre coût, des logiciels de très haute qualité.

2 Qualité, vous avez dit qualité ?

La qualité logicielle est une notion plurielle.

Difficile de décrire simplement ce qu'est un logiciel de qualité. Selon le point de vue que l'on adopte, la notion de qualité peut prendre des formes diverses : un utilisateur final se basera sur son expérience directe, sur l'ergonomie, la productivité pour accomplir certaines tâches, la stabilité, la fiabilité, les performances. De son côté, un développeur appréciera la rapidité de développement, la pertinence de la conception, la facilité de maintenance, la testabilité, la compréhensibilité du code source. Un exploitant s'attachera davantage à la facilité d'installation et de mise à jour, à la simplicité du diagnostic et de la supervision. L'architecte sera plus sensible à l'interopérabilité, à la portabilité, à l'intégration harmonieuse de la solution dans le système d'information. Un DSI, enfin, estimera les coûts de développement puis de maintenance et d'évolution. Bref : il existe beaucoup de critères possibles pour évaluer la qualité d'une application allant du plus concret au plus subjectif.

Pour faire le tri dans cette jungle de préoccupations, la norme ISO/CEI 9126 définit un vocabulaire visant à classifier l'ensemble des attributs d'un logiciel relevant de la qualité. La norme se présente sous la forme d'une arborescence de caractéristiques et sous-caractéristiques qui, chacune, décrivent une aspect de la qualité logicielle.

Selon cette norme, les 6 grandes caractéristiques de la qualité logicielle sont les suivantes :

- **Capacité fonctionnelle** : Le logiciel répond-il aux besoins de ses utilisateurs ?
- **Fiabilité** : Le logiciel est-il en mesure d'assurer un niveau de qualité de service suffisant pour satisfaire les besoins opérationnels de ses utilisateurs ?
- **Facilité d'utilisation** : Le logiciel peut-il être utilisé à moindre effort ?
- **Maintenabilité** : Est-il facile d'adapter le logiciel à de nouveaux besoins ou à de nouvelles contraintes ?
- **Rendement / Scalabilité** : Les ressources matérielles nécessaires à l'exécution du logiciel sont-elles en rapport avec sa rentabilité ?
- **Portabilité** : Le logiciel peut-il être transféré facilement d'une plate-forme ou d'un environnement à un autre ?

A ces caractéristiques, nous sommes tentés d'ajouter une septième, l'exploitabilité, particulièrement importante pour les applications serveur : le logiciel est-il facile à installer, à mettre à jour, à diagnostiquer et à superviser ?

Parmi tous ces aspect de la qualité logicielle, nous avons choisi, dans ce document, de nous concentrer sur ceux qui, selon nous, présentent la plus grande universalité : la maintenabilité et la fiabilité. Il n'est pas pour nous question, bien sûr, de minimiser les autres aspects. La capacité fonctionnelle, en particulier, ainsi que la facilité d'utilisation, présentent des défis singuliers, difficiles à relever. Mais ces défis présentent des visages divers selon le type de logiciel que l'on considère – un site web grand public, une application de gestion traditionnelle, un logiciel de courtage en temps réel, un gestionnaire d'imprimantes… Et les défauts dans ces domaines sont le plus souvent possibles à corriger a posteriori – fût-ce à un prix élevé -, pour peu que le logiciel soit maintenable. Quant aux autres aspects – rendement, performance, portabilité, exploitabilité – ils relèvent davantage de choix techniques adaptés aux contraintes d'exploitation du logiciel ; en 2010, et pour la plupart des applications, ces choix sont relativement simples à opérer, pour peu que l'on mobilise les bons experts.

Nous avons donc choisi de nous concentrer sur deux aspects de la qualité logicielle : la maintenabilité et la fiabilité.

Ce choix est piloté par deux motivations.

Tout d'abord, nous estimons que les défauts de fiabilité et de maintenabilité sont les plus coûteux pour l'industrie informatique dans son ensemble. Un manque de fiabilité se traduira par des erreurs de fonctionnement – des bugs -, dont les conséquences opérationnelles peuvent être catastrophiques. Un logiciel peu maintenable, quant à lui, verra ses coûts d'évolution croître avec le temps, potentiellement jusqu'à la calcification complète. Ce défaut de réactivité peut être mortifère, si l'on admet que la production logicielle a acquis une dimension stratégique pour de nombreuses organisations, et que ce statut impose d'ajuster en permanence les capacités fonctionnelles des logiciels à des exigences opérationnelles en évolution permanente.

Seconde motivation, nous pensons qu'il est possible de prendre des mesures universelles, indépendantes du domaine auquel appartient le logiciel, et permettant d'améliorer fortement fiabilité et maintenabilité. Ces mesures sont relativement simples à prendre, peu coûteuses, et leur retour sur investissement potentiellement spectaculaire – la seconde partie de ce document leur sera consacrée.

2.1 Fiabilité : le coût visible de la non-qualité

Eté 2009. Un petit bug dans la gestion des arrondis pour le calcul du nombre de trimestres cotisés à l'assurance vieillesse est découvert. Ce bug, introduit dans le système en 1984, a conduit à affecter un trimestre de trop à près de 8 millions de salariés, et à surestimer en conséquence le montant de leur pension. 25 ans plus tard, le coût total pour l'Assurance Sociale approche les 2,5 milliards d'euros.

Septembre 1999. Après plusieurs mois de voyage, la sonde Mars Climate Orbiter s'écrase piteusement sur la planète Rouge plutôt que d'adopter la trajectoire orbitale prévue. Après investigation, il apparaît que le système de correction de trajectoire, développé par une équipe du Jet Propulsion Laboratory, exploitaient des données codées selon le système métrique international – centimètres, mètres, kilomètres - tandis que le logiciel embarqué dans les capteurs, développé par une équipe du Lockheed Martin Astronautics, lui fournissait des données codées selon le système métrique impérial – pouces, pieds, miles…

Mai 1996. La First National Bank of Chicago met à jour son système de gestion des transactions en provenance des ATM. Objet de la mise à jour ? De nouveau codes d'erreur, réputés plus complets. Mais certains de ces codes sont mal interprétés par certains ATM, avec pour conséquence que les comptes courants de 813 clients de la banque sont crédités chacun de plus de 900 millions de dollars. Montant total de l'erreur ? Plus de 750 milliards de dollars…

Ces quelques exemples, tirés de l'histoire informatique, illustrent de façon spectaculaire les conséquences potentielles d'un manque de fiabilité. Ils ont pour point commun que quelques tests bien ciblés – unitaires dans le premier cas, d'intégration dans le second, de non régression dans le troisième – auraient suffi pour les détecter, et les corriger pour un coût infiniment inférieur à celui de leurs conséquences.

Le coût du manque de fiabilité des logiciels fait, depuis des décennies, l'objet d'analyses très nombreuses. Ce coût est fondamentalement dicté par les pertes opérationnelles – directes ou indirectes - dues à un dysfonctionnement, et les frais engendrés par sa correction. Hormis chez les éditeurs de logiciel, et dans certains secteurs hautement risqués – aéronautique, aérospatiale, armement, informatique médicale, système bancaire, pour ne citer qu'eux - son impact opérationnel est cependant longtemps resté limité : une source d'agacement récurrent, d'énervement parfois, rarement de catastrophe financière. Mais avec la généralisation d'une informatique stratégique, devenue un élément clé de la croissance et de la performance des entreprises, la fiabilité est devenue un sujet majeur dans de nombreuses DSI.

Il existe des moyens simples de réduire les défauts de fonctionnement d'un logiciel. Le principal est bien sûr le renforcement des pratiques de test. D'autres sont la revue de code systématique ou l'utilisation d'outils d'analyse statique – nous reviendrons en détail sur ces pratiques ultérieurement.

La question que l'on peut légitimement se poser est donc la suivante : si le coût du manque de fiabilité des logiciels est connu, et si la solution l'est aussi, pourquoi les pratiques de tests sont-elles encore à ce point défaillantes dans de nombreuses organisations ?

La réponse tient, selon nous, principalement à la combinaison de deux phénomènes : la préférence pour le présent, et le rendement marginal décroissant des tests.

La préférence pour le présent s'explique facilement. Tester davantage a un coût. Ce coût est d'autant plus important que le système est complexe, et il peut être mesuré au préalable. Le coût des dysfonctionnements quant à lui ne peut être mesuré qu'à postériori – c'est d'ailleurs la raison pour laquelle c'est l'un des rares domaines de la qualité qui fait l'objet d'analyses financières.

Pour reprendre l'un des exemple ci-dessus, il va de soit qu'écrire un test unitaire validant la règle d'arrondi du nombre de trimestres à prendre en compte pour le calcul de la pension de retraite aurait représenté une charge pratiquement nulle – surtout si on la met en regard de la perte engendrée par l'erreur. Mais pour que ce test ait eu une chance d'être écrit, il aurait fallu une politique de tests unitaires exhaustive, couvrant l'intégralité des fonctions du système. Et une telle politique impliquait un surcoût immédiat, alors difficile à mettre en regard de gains futurs.

La plupart des projets informatiques sont soumis à de fortes contraintes de budget et de délais. La préférence pour le présent consiste bien souvent à préserver des objectifs à court terme – livrer à temps – au détriment de risques à moyen et long terme. Et les tests, charge immédiate visant à préserver des intérêts futurs, fournissent une variable d'ajustement commode.

Cette préférence pour le présent est complétée et renforcée par ce que l'on qualifiera de rendement marginal décroissant des tests.

Pour comprendre ce second phénomène, nous nous appuierons sur le modèle statistique fournit par COQUALMO[1].

Le modèle COQUALMO, développé en complément de COCOMO II[2], utilise la notion de densité de défauts délivrés pour évaluer le niveau de fiabilité d'un logiciel. Cette densité se mesure en nombre de défauts par millier de lignes de code, et qui seront détectés après la mise en production. Le modèle identifie 6 profils de maturité en fonction de la densité de défauts :

1 COQUALMO : COnstructive QUALity Model, modèle d'estimation du nombre de défauts résiduels dans un logiciel.

2 COCOMO : COnstructive Cost Model, modèle statistique d'estimation des coûts basé sur 63 projets de 2k à 100k lignes.

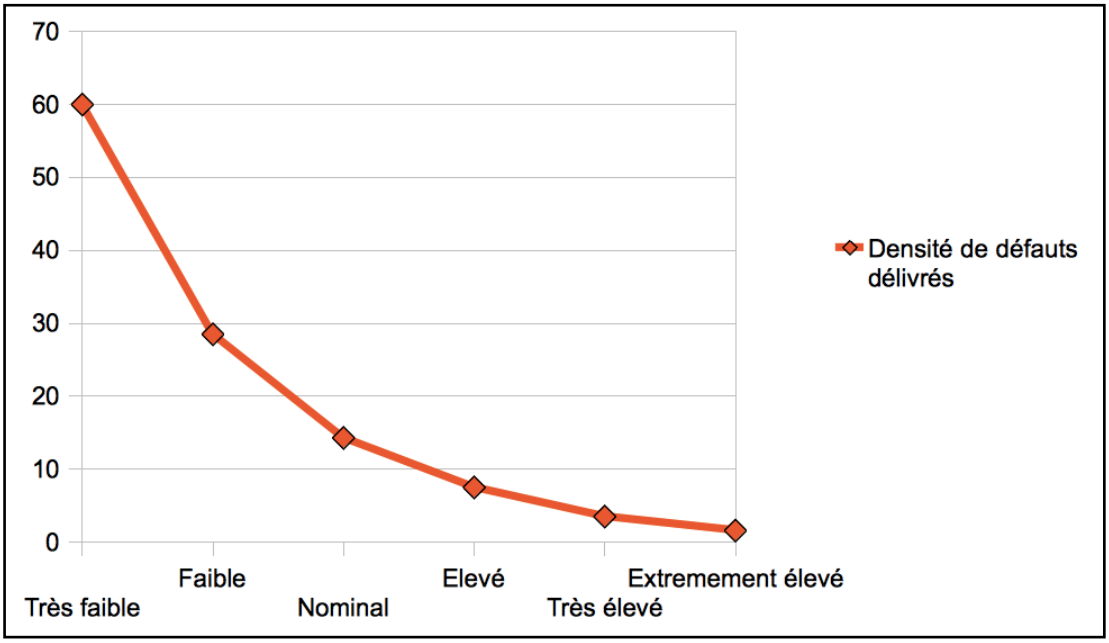


Illustration 1 : Estimation de la densité de défauts délivrés en utilisant COQUALMO
Source : Cost/Benefit-Aspects of Software Quality Assurance - Marc Giombetti

Ce modèle statistique a pour principale vertu de mettre en exergue un phénomène classique en micro-économie : les rendements marginaux décroissants. Si l'on admet que l'investissement nécessaire pour passer d'un niveau de maturité au suivant est du même ordre, alors le retour sur investissement, exprimé en densité de défauts, est décroissant : l'investissement consenti pour passer du niveau « très faible » au niveau « faible » permet de diviser la densité de défauts par trois ; l'impact du même investissement pour passer au niveau suivant est presque deux fois moindre ; pour atteindre le « zéro défaut », l'investissement est virtuellement infini. Traduit en termes financiers, ce modèle implique que le retour sur investissement des mesures d'amélioration de la fiabilité, en particulier le renforcement des tests, est également décroissant – au point de devenir négatif au delà d'un certain stade.

Marc Giombetti, dans son ouvrage « Cost/Benefit-Aspects of Software Quality Assurance », illustre ce mécanisme avec l'exemple d'un système de traitement de commandes d'une enseigne de VTT, dont le modèle de revenu est relativement simple – le revenu est fondamentalement proportionnel au taux de disponibilité du système. Le niveau de maturité est mesuré par le temps moyen entre deux défaillances (MTBF : Mean Time Before Failure) :

Projet	Maturité	MTBF (h)	Disponibilité	Pertes (M$)	Gains (M$)	Coût (M$)	Changements (M$)	ROI
Système de traitement des commandes de VTT	Nominal	300	0,9901	5,31	0	3,45	0	-
	Elevé	10K	0,9997	0,16	5,15	3,79	0,344	14,0
	Très élevé	300K	0,99999	0,005	0,155	4,34	0,55	-0,72
	Extr. élevé	1M	1	0	0,005	5,38	1,04	-1,0

Tableau 1 : Valeurs exemples de calcul de ROI
Source : Cost/Benefit-Aspects of Software Quality Assurance - Marc Giombetti

L'investissement consenti pour passer du niveau nominal au niveau élevé est très rentable – 5,15 millions de dollars de gains pour 344 000 dollars investis. Le rendement pour passer aux niveaux supérieur est quant à lui négatif.

Ce mécanisme fournit une justification rationnelle à ceux qui refusent d'investir dans le renforcement des tests : il suffit de prétendre que le niveau de test actuel est déjà suffisant, et que tout effort additionnel risquerait d'avoir un rendement négatif. Bref, ils craignent la sur-qualité.

La sur-qualité, et les coûts qu'elle comporte, a longtemps été réservée à ceux dont l'analyse de risques montraient une sensibilité extrême aux défauts. Une start-up qui souhaite mettre en ligne un service payant doit être irréprochable dès la première version – à défaut, elle risque de créer un bouche-à-oreille défavorable qui la tuera dans l'œuf. Un système de compensation bancaire, qui brasse des dizaines de millions d'euros chaque jour, est extrêmement sensible aux erreurs. Le système de pilotage automatique d'une navette spatiale, quant à lui, ne supporte pas l'échec.

Ces quelques exemples, considérés il n'y a pas si longtemps comme l'exception, sont devenus la règle. Support direct aux opérationnels, middleware transactionnel ou canal de vente stratégique, un grand nombre d'applications informatiques sont devenues critiques. Avec une conséquence importante : les leviers financiers de leur fiabilité sont tels que la sur-qualité est devenue rentable. Pour reprendre l'exemple de Giombetti, le calcul du retour sur investissement global pour passer du niveau nominal au niveau très élevé (qui correspond à de la sur-qualité pour le système considéré) est encore très favorable. L'investissement total est de 1,93 millions pour un gain de 5,31 millions. Soit un ROI de l'ordre de 180%, encore très honorable.

La recherche d'un optima improbable, difficile à déterminer à l'avance, devrait en conséquent être remplacé par un mot d'ordre très simple : tester le plus possible, le plus souvent possible.

Dans ces conditions, quand un chef de projet ou un décisionnaire quelconque accepte de sacrifier certains tests pour respecter certaines contraintes de coûts ou de délais à court terme, il réalise probablement un arbitrage peu judicieux. Il est souvent préférable de réduire le périmètre du logiciel mais de s'assurer que ce qui est livré est fiable.

2.2 Tolérance au changement : le coût caché de la non qualité

Si le manque de fiabilité, qui se traduit par des défauts de fonctionnement aux conséquences mesurables, a pu faire l'objet d'analyses de coûts très poussées, il n'en est pas de même pour la maintenabilité. Cela se comprend aisément : un logiciel peu maintenable coûte cher à faire évoluer ; mais il est impossible de comparer ce coût à ce qu'il aurait été si le logiciel avait été plus maintenable.

Avant de justifier pourquoi nous estimons que le manque de maintenabilité est l'un des défauts logiciels les plus onéreux, puis d'identifier les principaux leviers permettant de l'améliorer, nous vous proposons d'analyser un peu plus ce que le terme recouvre.

La maintenabilité, outre qu'il s'agit d'un anglicisme à l'élégance discutable, est également un terme trompeur. Il renvoie à la notion de maintenance qui, traditionnellement, désigne les opérations d'entretien nécessaires pour maintenir un artefact en condition opérationnelle. Elle consiste en somme à lutter contre les assauts du temps, afin de prolonger la période d'usage d'un produit : vérifier et ajuster le réglage de certaines pièces, réparer ou remplacer celles qui sont usées, remplacer certains composants par une version plus récente, repeindre un revêtement, mettre en conformité avec de nouvelles réglementations… La maintenance a donc pour vocation traditionnelle de maintenir le statu quo. Au sens strict, un certain nombre d'opérations logicielles relève effectivement du domaine de la maintenance : correction de dysfonctionnements, réorganisation de la base de données, archivage des fichiers de journalisation, historisation des données, recompilation et adaptation à une nouvelle génération de processeur ou de

système d'exploitation, voire refonte de la charte graphique pour adhérer à l'air du temps. Mais en vérité, la plupart de ces opérations ne sont même pas considérées comme relevant du domaine de la maintenance logicielle. Elles sont plutôt vues comme appartenant au domaine de l'exploitation. Ce que l'on désigne généralement sous le terme maintenance logicielle, c'est un changement dans le code source du logiciel. Certains de ces changements peuvent être assimilés à la notion traditionnelle de maintenance (correction de bugs, recompilation, adaptation à un nouvel environnement technique, etc.), et d'autres non (extension d'une fonctionnalité existante, ajout de fonctionnalités nouvelles, support de nouvelles contraintes). Pour refléter cela, il est fréquent de distinguer la maintenance dite corrective – qui relève effectivement de l'entretien – et la maintenance évolutive – qui est un oxymore, puisque par définition, elle vise précisément à rompre le statu quo. C'est la raison pour laquelle nous estimons que le terme maintenabilité est trompeur, et lui préférons ceux d'évolutivité, d'adaptabilité ou encore de tolérance au changement.

Maintenance corrective	Maintenance évolutive
Correction d'un défaut détecté après la mise en production Refonte d'un algorithme de calcul erroné ou peu performant aboutissant au non respect du contrat de service Restauration d'un état antérieur cohérent	Ajout d'une fonctionnalité Adaptation du logiciel aux nouvelles exigences des utilisateurs Adaptation du logiciel à un nouvelle législation ou aux nouvelles exigences du marché

Tableau 2 : Maintenance corrective et évolutive

Ce long préambule terminologique nous amène au cœur du sujet.

L'approche traditionnelle de la maintenance logicielle s'appuie sur l'idée que les principaux changements apportés à un logiciel après sa première mise en service relèvent de la correction d'anomalies. Cette correction d'anomalies nécessite le plus souvent une intervention chirurgicale sur le code source, qu'un minimum de structuration et de documentation suffit à rendre possible pour un coût raisonnable – d'autant plus que l'on peut s'attendre à ce que le flux d'anomalies se tarisse avec le temps. De ce point de vue, seules quelques situations singulières justifient une attention poussée à la tolérance au changement : les éditeurs de logiciels commerciaux, en particulier, doivent régulièrement proposer des versions enrichies de leurs produits pour maintenir leurs revenus.

Nous pensons quant à nous que ce qui était une exception est désormais la règle – à supposer qu'elle ne l'ait pas toujours été. La plupart des logiciels que nous mettons aujourd'hui en service sont des évolutions de logiciels existants. Et il est de plus en plus communément admis que le nombre de lignes de code produites après la première mise en service d'une solution logicielle est supérieur à la celui produit avant. Dans des proportions toujours plus spectaculaires. Le graphique suivants illustre cette évolution du nombre de lignes de code en fonction du temps sur un projet, la première version n'étant souvent qu'un embryon de ce que sera l'application dans le futur.

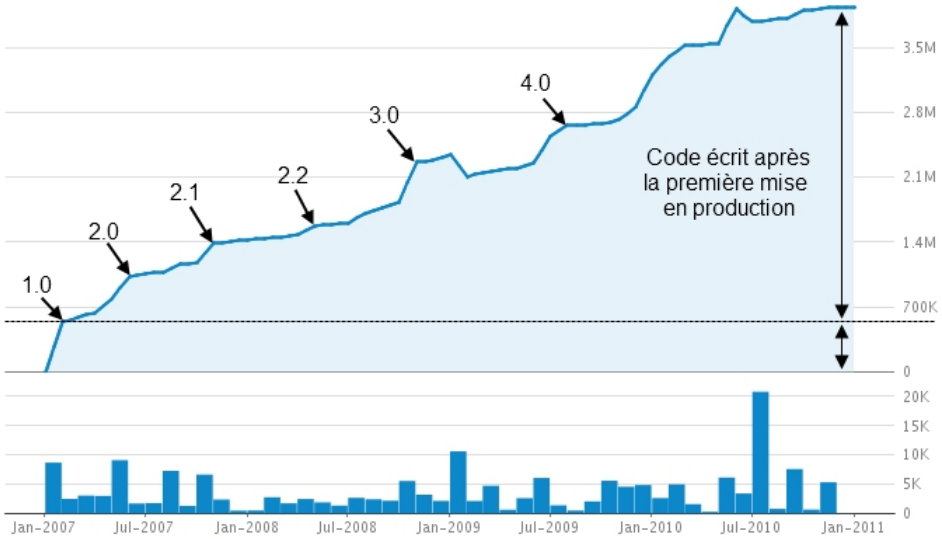


Illustration 1 : Evolution du nombre de lignes de code en fonction du temps

A cela, deux raisons : en premier, les applications sont de plus en plus proches des fonctions stratégiques des organisations, et doivent donc être en permanence ajustées à son environnement opérationnel ; nouvelles offres, nouveaux marchés, nouveaux concurrents, nouvelles façons de faire… le monde change vite, et les logiciels doivent s'adapter au même rythme. Deuxièmement, les architectures modernes, orientées réseau, autorisent la diffusion instantanée des changements à tous les utilisateurs pour des coûts très faibles. Pourquoi, dès lors, se priver de mises à jours fréquentes ?

Dans ces conditions, il va de soit que l'impact financier d'un manque d'évolutivité est démultiplié. Directement, et indirectement.

Directement, tout d'abord, puisque le coût d'une évolution est supérieur à ce qu'il devrait être, et que ce surcoût est acquitté à chaque nouvelle évolution : si les évolutions sont nombreuses, ce surcoût est mécaniquement démultiplié. De surcroît, le défaut d'évolutivité tend à s'amplifier avec le temps – un logiciel peu adaptable, que l'on fait évoluer au forceps, se dégrade encore davantage. Le coût marginal des évolutions est alors croissant, jusqu'au stade où la seule décision rationnelle devient la réécriture complète.

Le schéma suivant montre comment, la charge de maintenance évolue au fur et à mesure que l'application grossit en raison du coût marginal introduit par la non qualité des itérations précédentes. Le logiciel se retrouve petit à petit dans un état où les évolutions prennent de plus en plus de temps, nécessitant des efforts de plus en plus conséquent pour une production réelle de plus en plus mince. La charge de travail augmente alors que le nombre d'évolutions diminue.

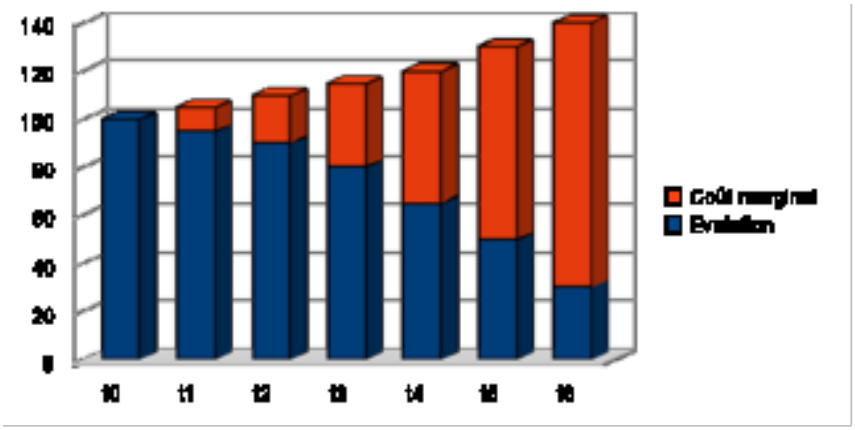


Illustration 2 : Evolution du coût marginal en fonction du temps

Indirectement, ensuite, puisque le manque de réactivité se répercute sur les activités auxquelles le logiciel apporte un support de plus en plus critique. Et cette répercussion peut avoir des conséquences opérationnelles dramatiques. Autre aspect du coût indirect : celui des autres défauts de qualité. En effet, tous les autres défauts évoqués plus haut – capacité fonctionnelle, ergonomie, fiabilité, etc. – peuvent être corrigés. Mais le coût et les délais de ces corrections dépendent directement de la tolérance au changement de l'existant. L'impact opérationnel de ces autres défauts est donc prolongé, ou amplifié, par le manque d'évolutivité.

Difficile, bien sûr, de mesurer avec précision le coût réel du manque d'adaptabilité d'un logiciel – qui dépend, bien sûr, des conditions d'usage de celui-ci. Nous estimons cependant que ce coût est considérable, et augmente avec la généralisation d'une informatique stratégique.

> ***Dès lors, comment améliorer la tolérance aux changements de nos logiciels ?***

Nous avons retenu trois mesures qui, selon nous, offrent le meilleur retour sur investissement dans ce domaine :

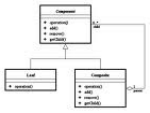
- les tests, une nouvelle fois, et plus particulièrement leur automatisation ;
- la conception, ensuite ;
- la lisibilité du code source, enfin.

Il en existe probablement un quatrième que nous ne traiterons pas ici : le développement agile. Ce dernier, en effet, considère chaque développement, dès la première ligne de code, comme une évolution de ce qui existe déjà. En conséquence, il fournit un moyen simple de contrôler en permanence que le logiciel produit est tolérant aux changements – et de corriger rapidement le tir dans le cas contraire.

Avant de détailler chacune de ces mesures, une précision s'impose. Nous n'avons en effet pas retenu la documentation dans cette liste. Nous estimons en effet qu'aucun effort de documentation ne peut corriger le défaut d'évolutivité d'un logiciel difficile à tester, mal conçu ou dont le code est illisible. Cela ne signifie en aucun cas que la documentation est inutile ; mais son utilité est directement tributaire de ces autres aspects.

L'automatisation des tests est la première mesure indispensable pour augmenter la tolérance au changement. La raison en est simple. Quelle que soit la raison pour laquelle on souhaite modifier un logiciel, la plupart des modifications se font à la marge. Autrement dit, le volume de code qui n'est pas modifié est très largement supérieur au volume de code modifié. Tout le risque réside alors dans les éventuelles régressions introduites dans le reste du code par cette modification marginale. Pour s'assurer contre ce risque de régression, il est nécessaire non seulement de tester les nouvelles fonctionnalités, mais également de vérifier que toutes les autres fonctionnent toujours correctement. En somme, le coût total d'une évolution comprend le développement de la nouvelle fonctionnalité, plus le test de l'intégralité du logiciel.

Le nombre de fonctionnalités allant croissant, cet effort de test ne fait qu'augmenter. Rapidement, le coût des tests de non régression excède très largement celui du développement des évolutions. Seule une automatisation poussée permet alors de maintenir ce coût dans des proportions acceptables. Ceci est d'autant plus vrai que les technologies modernes offrent de nombreuses possibilités d'automatisation des tests à faible coût : tests unitaires, tests d'intégration, tests fonctionnels, tests de performances… la plupart des tests peuvent aujourd'hui être automatisés avec des solutions gratuites. Nous y reviendrons dans le prochain chapitre.



Seconde mesure visant à augmenter la tolérance au changement de nos logiciels, la conception. Il existe plusieurs paradigmes de programmation : programmation impérative, fonctionnelle, procédurale, événementielle, orientée objet… Concentrons nous sur le dernier, le plus répandu à ce jour dans les applications métier, et probablement celui dont les fondements théoriques sont les plus directement liés à la problématique de l'évolutivité.

La programmation orientée objet a en effet spécifiquement été conçue dans les années 60 pour simplifier les évolutions de logiciels à la complexité croissante. Elle est sous-tendue par un modèle mathématique sophistiqué, dont la principale vertu est de limiter les effets de bord d'un changement. Pour cela, la conception objet repose sur un ensemble d'unités discrètes et indépendantes, les classes, chacune dotée d'un rôle ou d'une responsabilité distincts. Ces classes encapsulent données et comportements, et collaborent avec d'autres classes pour réaliser des traitements complexes. Correctement utilisées, les techniques de programmation orientée objet offrent des leviers spectaculaires pour limiter les efforts nécessaires à l'implémentation d'une nouvelle fonctionnalité, et les risques de régression associés. Malheureusement, et malgré le succès spectaculaire des langages orientés objet (Java, C++, C#, VB.NET, Ruby, PHP, etc.), le niveau de la conception objet reste bien souvent insuffisant ; pour tout dire, la plupart des applications que nous auditons, bien qu'écrites avec un langage objet, sont plutôt conçues sur un mode procédural. Améliorer la tolérance au changement des logiciels passe en conséquence par une élévation de la conscience objet des développeurs. Au-delà de la connaissance de certains design patterns, il est primordial de s'assurer qu'ils maitrisent certains des principes fondamentaux de la conception objet : loi de Demeter, principe de substitution de Liskov, principe Open Close, principe de cohésion, principe d'abstraction des éléments stables, etc. Bien qu'il soit possible de détecter certains défauts de conception objet au travers d'une analyse statique du code, cette analyse ne suffit pas. Pour améliorer véritablement le niveau de conception objet, certaines mesures sont indispensables : s'assurer que chaque équipe dispose d'au moins un concepteur objet expérimenté, réaliser la conception collectivement, procéder à des revues de code systématiques, pratiquer le pair-programming, etc. Nous reviendrons plus en détail sur ces aspects dans le second chapitre.

Enfin, dernière mesure visant à favoriser la tolérance au changement des logiciels, la lisibilité du code source. Une nouvelle fois, il s'agit d'un aspect très souvent négligé. Pourtant, ce défaut de lisibilité est souvent évoqué : une règle quasiment universelle du développement est qu'il est beaucoup plus difficile de lire du code existant que d'écrire du nouveau code. D'où la tentation récurrente de la part des développeurs de tout réécrire, plutôt que de chercher à comprendre ce qui existe. C'est vrai lorsque l'on consulte le code écrit par un autre développeur, mais c'est aussi vrai lorsque l'on consulte du code que l'on a écrit soi-même, quelques semaines ou quelques mois plus tôt. Si le code est difficile à lire, il est ardu de comprendre ce qu'il fait. Il est

alors malaisé de déterminer ce qui doit être changé et, pire encore, il est très difficile de réaliser un changement sans corrompre l'intégrité conceptuelle du code existant.

La raison pour laquelle il est plus difficile de lire du code que d'en écrire est très simple : lorsque l'on programme, la principale difficulté consiste à décrire un comportement en utilisant un langage formel très contraignant. A ce moment-là, l'enjeu réside principalement dans des considérations syntaxiques et algorithmiques : ce qu'attend le compilateur pour convertir le texte en programme exécutable. Le compilateur n'a que faire de la sémantique du code, ou de l'intention du programmeur. Pour lui, une méthode nommée acheterDesBananes a autant de sens qu'une méthode nommée calculerEncours, pour peu que les règles syntaxiques soient respectées. Les noms des classes, des méthodes et des variables lui importent peu : ils seront convertis en symboles adressables. Pour celui qui lit le code, par contre, le nommage et la simplicité sont primordiaux. Une méthode trop longue, enchaînant les structures de contrôle, est difficile à comprendre, même si elle est fonctionnellement correcte. Une méthode dont le nom ne reflète pas la fonction sera source de confusion, voire de malentendu, pour celui qui cherchera à en comprendre l'usage. Et nous pourrions multiplier les exemples.

Comment améliorer la lisibilité du code ? Certains outils d'analyse statique permettent de détecter des algorithmes trop complexe – c'est notamment le cas de l'indice de complexité de McCabe, ou indice de complexité cyclomatique. Mais cette analyse statique est insuffisante, puisqu'elle néglige la sémantique du code. La solution réside dans certaines pratiques de conception – notamment la conception pilotée par le domaine, ou Domain Driven Design, DDD -, mais surtout dans des pratiques collaboratives : standards de codage, revues de code systématiques, pair-programming…

Ces mesures sont indispensables à l'obtention d'un logiciel de qualité. Nous allons maintenant les étudier plus en détail, en cherchant à isoler, de manière pragmatique, en quoi elles contribuent individuellement à l'amélioration de la qualité et comment en tirer le meilleur parti, en les associant et en les outillant. Nous verrons également comment les connaissances et l'expérience des individus jouent un rôle primordial et comment accroitre la productivité et la qualité de leurs réalisations par la collaboration et le partage. Nous aborderons, enfin, les problématiques de concurrence et de délai de mise sur le marché – le fameux « Time to Market » – qui souvent orientent les décisions de le sens de la non qualité, préférant la réactivité immédiate à la productivité sur le long terme.

3 Tests : la garantie satisfaction utilisateur

3.1 Pourquoi s'en passer ?

S'il y a un élément indispensable à l'obtention d'un logiciel de qualité ce sont les tests. Eux seuls garantissent que l'application répond bien aux besoins et ne comportent pas d'anomalies. Ils sont néanmoins un coût important sur un projet. Il est communément admis qu'au moins 30% de la charge d'un projet doit être attribué aux tests. C'est une part importante mais il ne faut surtout pas la négliger.

Néanmoins, il existe des moyens d'augmenter la productivité des tests et du développement et de faire des tests l'élément central du développement de logiciel.

3.2 Les tests

3.2.1 Tests fonctionnels

Les tests fonctionnels permettent de valider que les fonctionnalités du logiciel sont bien implémentées et correspondent aux attentes des utilisateurs. Ils rendent compte de l'état d'adéquation au besoin en prenant en compte uniquement l'interface entre l'utilisateur et l'application. Ce pourra être par exemple une interface graphique, une ligne de commande ou un service (dans ce cas l'utilisateur est une autre application). Les données de test sont sélectionnées sans examiner le programme, qui est considéré comme une « boîte noire » pour laquelle nous voulons fournir des entrées et récupérer les sorties. Les tests sont réalisés dans un environnement le plus proche possible de l'environnement utilisateur cible. Les données et jeux d'essai doivent être des cas réels ou, dans le cas où cela est impossible (confidentialité), des cas simulant la réalité. Il sera possible de passer par une phase d'anonymisation des données de production afin d'avoir une base de jeux d'essai suffisante et réaliste.

> ***Cette phase de test est primordiale et représente le minimum à mettre en oeuvre pour un logiciel de qualité.***

En effet, la base et première caractéristique d'un logiciel de qualité est la satisfaction utilisateur et seuls les tests fonctionnels peuvent garantir que les fonctionnalités attendues sont implémentées et correspondent au besoin.

Dans le cas d'une reprise d'un logiciel existant, la première étape à mettre en oeuvre sera de définir les scenarii de tests fonctionnels appropriés et les jeux de données adéquats. Nous serons alors en mesure de le valider fonctionnellement et d'en assurer la non régression.

Il existe plusieurs façons de faire des tests fonctionnels. Le cas d'école préconise de structurer, organiser, référencer les tests afin d'être en mesure de connaître clairement les tests permettant de valider une fonctionnalité et de les rejouer facilement, de manière déterministe, par un testeur quel qu'il soit. Cette technique offre de nombreux avantages : reproductibilité des tests, complétude, réflexion en amont, mutualisation des cas de tests, association fonctionnalité/cas de test, formation des testeurs aux fonctionnalités du logiciel faible. D'un autre côté, cette méthode peut être lourde à mettre en oeuvre. En effet, maintenir un référentiel, organisé, demande des ressources importantes, surtout si l'application évolue fortement. Une équipe de testeurs connaissant bien le logiciel et ses fonctionnalités, peut obtenir de très bon

résultats sans ce référentiel ou avec un référentiel réduit. Leur capacité à se comporter comme des utilisateurs réels, leur rapidité d'exécution et leur capacité à détecter des anomalies non prévues du fait de leurs connaissances fonctionnelles élevées leur confère une efficacité bien supérieure. Néanmoins, il faut investir en formation, et le processus de validation repose alors plus sur les personnes ce qui peut engendrer un risque.

3.2.2 Tests unitaires

Les tests unitaires, quant à eux, se concentrent sur de petites portions de l'application (une méthode, une classe, un paquetage) et permettent de détecter et corriger les erreurs à faible coût. Plus près du code, ils sont faciles à écrire car les cas à tester sont réduits. L'échec d'un test cible directement la portion de code à revoir ou à corriger. Cependant, il est souvent impossible ou trop lourd de réaliser ces tests dans un environnement complet. Certains modules utilisés ne sont pas accessibles, non encore implémentés ou ralentissent l'exécution du test. Celui-ci doit s'exécuter rapidement, en isolation, et de manière systématique lors du développement. S'il prend plus de 100 ms sur un poste de développement moderne, c'est trop long. Pour ces raisons, les tests unitaires utilisent couramment des « bouchons ». Ceux-ci permettent de simuler le comportement d'un composant en l'absence de celui-ci et ainsi permettre aux modules qui en dépendent de s'exécuter et d'être tester.

> ***Leur efficacité pour détecter et isoler les anomalies les rendent nécessaires à toute stratégie qualité. Ils permettent de détecter au plus tôt les anomalies et ainsi de limiter leurs impacts.***

Associés à d'autres techniques comme nous le verrons par la suite (cf. chapitres 3.3 , 3.4.4), ils apportent une productivité importante à faible coût.

3.2.3 Tests d'intégration

Différents modules d'une application peuvent fonctionner unitairement, leur intégration entre eux ou avec des services tiers peut engendrer des dysfonctionnements. Comme nous l'avons vu précédemment, il est souvent impossible de réaliser les tests unitaires dans l'environnement cible avec la totalité des modules à disposition. Les tests d'intégration ont pour objectif de créer une version complète et cohérente du logiciel (avec l'intégralité des modules testés unitairement) et de garantir sa bonne exécution dans l'environnement cible.

3.2.4 Tests de charge

Les tests définis précédemment, valident le logiciel en terme de fonctionnement simple et unitaire. En effet, dans la plupart des cas, chaque test simule une opération d'un unique utilisateur. Cela permet de s'assurer de la fiabilité du logiciel mais ne garantie en rien sa robustesse. Quel sera son comportement lorsque plusieurs utilisateurs accèderont aux fonctionnalités du logiciel simultanément ?

Avant d'y répondre il est important de définir la charge que celui-ci doit pouvoir supporter et ses capacités à fournir les mêmes services à plusieurs utilisateurs simultanément. Les programmes de type « batch » par exemple ne permettent souvent pas de traitement parallèle alors que les sites Web doivent pouvoir accueillir un nombre maximum de visiteurs. Dans ce dernier cas, il est important de bien estimer le besoin dès le début et son évolution.

Les tests de charge simulent, via des injecteurs, le comportement d'un nombre définit d'utilisateurs. Ils se basent sur des scripts simulant des actions utilisateurs réelles ou réalistes et avec des jeux de données proches de celles de production. En effet, l'action de 100 utilisateurs demandant le détail d'un client sera

différente si le nombre de clients est de 10 ou de 100 000. De même, si les 100 utilisateurs demandent le détail du même client, il est probable que celui-ci soit mis en cache quelque part par le logiciel et que le résultat soit extrêmement rapide.

Il est par conséquent extrêmement important de bien penser le test de charge pour le rendre le plus réaliste possible et donc plus efficace. Comme tous les tests, le plus tôt ceux-ci seront mis en oeuvre, plus vite une éventuelle erreur dans la conception, l'implémentation ou le dimensionnement de l'infrastructure sera détectée et plus rapidement les correctifs pourront être mis en oeuvre en minimisant leur impact.

3.3 La non régression

Le développement initial n'est que la partie émergé de la vie d'une application. 60 à 80% du code est produit lors de la maintenance évolutive ou corrective. Il en découle qu'un ratio tout aussi important d'anomalies seront produites après la première mise en production. Comment s'assurer que les fonctionnalités existantes n'ont pas été corrompues par les modifications apportées ?

Le moyen le plus sûr réside dans le principe de tests de non régression. Ces tests ont pour objectif de valider les fonctionnalités existant préalablement sur la nouvelle version du logiciel. Le plus souvent cela consiste à rejouer la totalité des tests et cela induit un coût important sur le projet ; chaque livraison nécessitant de tester la totalité de l'application.

Néanmoins, il est possible de cibler les tests à réaliser en fonction des modifications apportées et de l'impact de celles-ci sur l'existant, mais cela nécessite une grande maîtrise de l'environnement aussi bien technique que fonctionnel et induit une part de risque. Une autre technique pour optimiser ces tests consiste à les automatiser et ainsi permettre de les rejouer indéfiniment sans surcoût humain. Les tests de non régression font alors partie intégrante du développement au même titre que les tests unitaires. Associés à un processus d'intégration continue, les régressions engendrées par une modification sont détectées au plus tôt et corrigées aussitôt. Cela nécessite néanmoins un investissement préalable pour l'écriture et la maintenance des tests et le retour sur investissement peut varier en fonction de la fréquence des évolutions du logiciel.

3.4 Efficacité des tests

3.4.1 Trouver des bogues au plus tôt

Tester complètement une application peut prendre un temps infini. Tester tous les cas possibles, à tous les niveaux est une perte de temps considérable, n'a pas d'intérêt réel et offre un retour sur investissement négatif. Un bon test est un test qui essaye d'identifier une anomalie ou un défaut (cf. *G. J. Myers (The Art of Software Testing, 1979)*).

> ***Un test fait pour trouver des bogues a une valeur, qu'il en trouve ou non.***
>
> ***Un test fait pour "réussir / passer" n'a pas de valeur.***

Leur efficacité dépend, de plus, de la précocité de leur mise en oeuvre. Plus une anomalie est détectée tôt, plus elle sera facile à corriger. Le schéma suivant illustre bien ce constat. Il représente le coût relatif d'une anomalie en fonction de la phase de développement durant laquelle elle a été découverte.

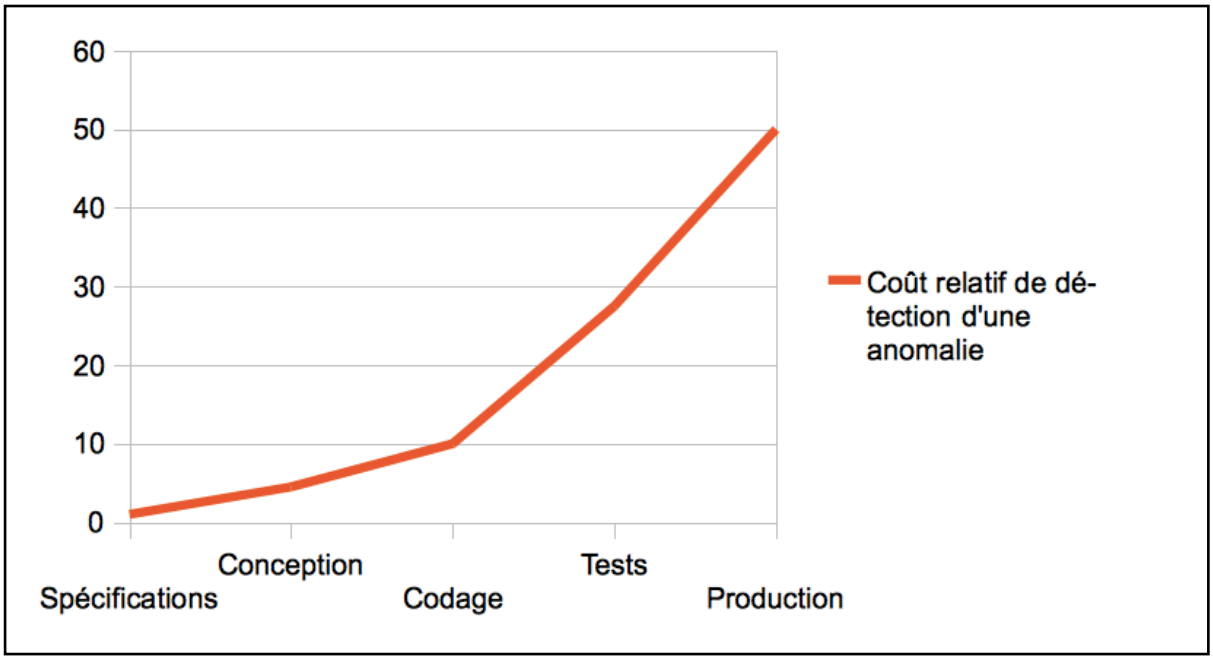


Illustration 4 : Coût relatif de détection d'une anomalie

On voit ici que le coût est multiplié par 5 entre un défaut détecté pendant la réalisation ou en production. Cette valeur est considérée comme plancher est peut être bien plus élevée (x30, x70) en fonction du contexte et de la qualité des rapports d'anomalie. Il est par conséquent important de tester le plus tôt possible afin d'en retirer le maximum de bénéfices.

3.4.2 Couverture des tests

> ***« Que valent mes tests, quantitativement et qualitativement ? »***
>
> ***« Quelle est leur couverture technique et fonctionnelle ? »***

Par ces questions, le testeur cherche à connaître la qualité de ses tests. Faire des tests est une chose, mais encore faut-il qu'ils soient représentatifs de ce que les utilisateurs vont réaliser par la suite.

Il y a divers moyens de mesurer la couverture des tests :

Couverture de code

La première consiste à mesurer, de manière automatique, la couverture de code réalisé par la campagne de tests. Cela signifie mesurer le pourcentage de code traversé par les tests par rapport à l'ensemble du code écrit. Le résultat donne une indication sur le fait que les tests sont passés ou non sur l'ensemble des composants, méthodes et lignes de l'application mais ne garantie en rien la pertinence des tests. Une couverture incomplète doit générer un action corrective :

- « Mes tests sont ils incomplets ? » - ***Compléter les tests.***
- « Mon application contient-elle du code mort ? » - ***Supprimer le code inutile.***

Associée aux tests automatisés, elle permet d'identifier au plus tôt les éventuels manques des tests unitaires par exemples.

Couverture fonctionnelle

La deuxième est plus délicate à mettre en oeuvre. Il s'agit de s'assurer que l'on a testé toutes les fonctionnalités, règles métier du logiciel. Cette fois-ci, il s'agit de s'assurer qu'à chaque fonctionnalité de l'application correspond bien un plan de test chargé de le valider. Il faut maintenir un référentiel comportant d'un côté les différents cas d'utilisation, règles métier, fonctionnalités du logiciel et de l'autre les scenarii de tests. Cela peut être fait via une feuille de calcul ou par des outils spécialisés du marché. L'objectif est de pouvoir avoir une matrice associant règles et tests pour détecter les oublis éventuels.

Cela nécessite de bien connaître, maitriser et maintenir ce référentiel qui peut vite devenir obsolète s'il n'est pas mis à jour continuellement.

3.4.3 Test Driven Development[3]

> ***Comment tester le plus tôt possible une application ? Avec une couverture maximale ?***

Une technique consiste à définir les tests avant la réalisation. Pour chaque fonctionnalité, il faut définir les tests appropriés permettant de valider son bon fonctionnement. La pratique la plus courante concerne les tests unitaires mais cela peut et doit être étendu aux tests fonctionnels. Le principe est de créer le test avant même l'écriture de la fonctionnalité et de l'exécuter afin de valider qu'il échoue bien puisque la fonctionnalité qu'il est sensé tester n'existe pas encore. Dans le cas contraire, le test ne sert à rien et doit être revu. Ensuite seulement, le code est ajouté pour répondre aux besoins et testé pour valider cette fois que le programme fonctionne correctement.

Les avantages sont multiples :

- Permet de préciser les spécifications en rapport avec un comportement réel attendu et de réfléchir aux différents cas pouvant intervenir.
- Nécessite d'avoir une vision précise de la manière dont le programme va être utilisé avant même d'envisager son implémentation. La conception émerge des tests et des cas concrets d'utilisation. Les erreurs de conception, dues à une implémentation précipitée, sont ainsi limitées.
- Assure que le logiciel est testable simplement. La couverture des tests est maximale.
- Permet d'avoir à disposition les tests de non régression associés à la version du logiciel. Outre la capacité à valider qu'il n'y a pas de régression par rapport à la version précédente, cela facilite le « refactoring[4] » de l'application et donc sa capacité à évoluer. La capacité à tester facilement permet de s'assurer rapidement qu'une modification n'a pas d'impact sur le résultat attendu. Le code est, par conséquent, modifié sans crainte et la santé de l'application s'en trouve améliorée.
- Une fonctionnalité est effective lorsqu'elle est testée. Les tests sont, dans ce cas, déjà définis au moment où la fonctionnalité a fini d'être codée. Les tests peuvent avoir lieu au plus près des développements et débuter dès qu'un sous ensemble applicatif est prêt sans attendre la fin du développement.

3.4.4 Automatisation

Afin de capitaliser et ainsi minimiser le coût des tests, il est important de recourir à leur automatisation dès que cela est possible. C'est particulièrement simple et efficace pour les tests unitaires. Nombre de framework permettent d'écrire simplement des scripts de tests unitaires dans un environnement donné

3 Test Driven Developpement (TDD) : développement piloté par les tests. Méthode de développement de logiciel qui préconise d'écrire les tests unitaires avant d'écrire le code source d'un logiciel.

4 Refactoring : opération de maintenance du code informatique visant à améliorer sa lisibilité, simplifier sa maintenance, ou changer sa généricité.

(xUnit). D'autres permettent de reproduire et valider fonctionnellement le logiciel d'un point de vue utilisateur (Selenium). Ils peuvent être exécuté de manière automatique et permettent ainsi :

- De valider une fonctionnalité ou l'ensemble de l'application de manière automatique.
- De s'assurer que les modifications apportées n'ont pas engendré de régression dans le logiciel.
- De détecter au plus tôt une anomalie et par conséquent d'en limiter l'impact et le coût.

Associée au Test Driven Development, l'automatisation génère une économie importante sur le coût des tests et de maintenance. En effet, malgré le surcoût induit par la création des scripts d'automatisation, le gain en terme de non régression et de précocité des détections d'anomalies rend en général la pratique rentable très rapidement. On s'assure ainsi à moindre frais qu'un correctif n'engendre pas de défaillance annexe sur le système. L'évolution du logiciel est encadrée, avec un risque maîtrisé.

> ***L'automatisation est un investissement, qui devient obligatoire si l'on fait des livraisons sur des cycles courts.***

L'expérience montre que l'automatisation est rentable, en charge de tests, à partir de la 4e campagne. La faculté de l'exécuter de manière répétée, tout au long du processus de développement et de maintenance, apporte un bénéfice supplémentaire bien que plus difficilement quantifiable.

3.4.5 Vers l'amélioration continue

Nous l'avons vu, les tests doivent être définis le plus tôt possible et limiter ainsi les anomalies en production. Le zéro défaut n'existe malheureusement pas et il est probable que malgré les tests, un certain nombre d'anomalies apparaissent en production.

Comment tirer parti de cet événement pour améliorer le logiciel et le processus de développement ?

La première règle à mettre en place est la création systématique des tests permettant de détecter l'anomalie et de l'ajouter au référentiel. Nous sommes en mesure de la reproduire, par conséquent de la corriger et de la tester par la suite. On s'assure ainsi que les prochaines versions n'auront pas le problème.

On s'interroge ensuite sur la raison de cette anomalie. Pourquoi cette anomalie s'est-elle produite ? Spécifications erronées, incomplètes ? Couverture des tests trop faible ? Code trop complexe, conception perfectible ? Processus non maîtrisé ? Paramétrage incomplet en production ? La réponse donnera une indication sur ce qu'il faut faire pour améliorer la qualité du développement du logiciel.

> ***Chaque bogue doit être perçu comme le point de départ de l'amélioration du processus de développement dans son ensemble.***

Cette pratique est particulièrement bénéfique lors de la reprise d'un existant ne comportant pas de tests. Nous ne savons souvent pas par où commencer pour la création des tests. Cela donne une direction, un point de départ pour sécuriser l'application.

3.5 Le coût réel des tests

Les tests ont un coût, c'est certain, mais eux seuls peuvent garantir un résultat conforme aux attentes des utilisateurs et un bon niveau de qualité. S'en passer, c'est courir le risque de délivrer un logiciel de mauvaise qualité avec les impacts financiers associés : perte d'image, arrêt de la production, abandon du projet, régressions et difficultés d'évolution, perte de compétitivité.

On peut considérer les tests comme une assurance, à la différence près que les logiciels non testés ont toujours des anomalies, beaucoup.

C'est un investissement dont le niveau de rentabilité peut varier en fonction des techniques et de la durée de vie du logiciel. Tester au plus tôt réduit les coûts de développement et de maintenance. L'automatisation des tests est, quant à elle, d'autant plus rentable que la durée de vie est longue et que l'application évolue.

L'expérience, enfin, dans l'organisation et l'écriture de tests efficaces permet d'améliorer le retour sur investissement.

4 Conception : les fondations du logiciel

Nous l'avons vu précédemment, les tests permettent d'obtenir un logiciel qui répond aux besoins des utilisateurs. Leur création au plus tôt et leur automatisation permettent d'en accroître l'efficacité et ainsi de réduire leur coût de mise en oeuvre.

Ils permettent, de plus, de s'assurer que les modifications apportées au logiciel n'engendre pas de dysfonctionnement. Cependant, ces modifications (correctifs ou évolutions fonctionnelles) ont pu être plus ou moins longues et complexes à mettre en oeuvre. Cette capacité à évoluer rapidement et facilement est un aspect important de la qualité d'une application et seule une conception intégrant cette problématique peut y répondre favorablement.

4.1 Caractéristiques d'une bonne conception

La conception est au coeur du développement du logiciel. Elle lui confère son organisation, sa capacité à répondre aux besoins fonctionnels. Cette phase doit prendre en compte de multiples paramètres afin d'orienter les possibilités de l'application en fonction des besoins techniques et organisationnels, présent et à venir.

Le premier objectif de la conception est de trouver les réponses aux problèmes de développement du logiciel et de permettre sa réalisation. Elle devra prendre en compte les contraintes liées à l'utilisation du logiciel par les utilisateurs finaux comme la fiabilité et la facilité d'utilisation.

De même, la plupart des logiciels doivent répondre à des problématiques de performance, de robustesse et de tolérance aux fautes. La conception doit prendre en compte ces notions dès le début si elle veut pouvoir y répondre car elles sont fondamentales. Un logiciel mono-poste et un site web commercial supportant 2000 connexions simultanées ne seront pas conçus de la même façon.

Enfin, une des caractéristiques principales d'une bonne conception réside dans la capacité du système à évoluer, à s'adapter aux changements. Il est souvent simple de réaliser un composant pouvant répondre à une spécification. Mais comment le faire évoluer pour répondre à des cas particuliers ou une évolution du marché. La réponse repose en général par une multiplication de tests ou de programmes et de cas particuliers noyant le cas général. La lecture du code, sa compréhension et sa maintenabilité sont dégradées et les risques d'anomalies dues aux incompréhensions et à la complexité sont élevés.

La modularité permet d'avoir des composants bien définis et indépendants améliorant ainsi la maintenabilité générale. Les composants peuvent être implémentés et testés en isolation avant d'être intégrés dans le logiciel final. La division du travail s'en trouve facilité.

Les composants ainsi créés doivent capturer l'essence de la fonctionnalité dont ils ont la charge, ni plus, ni moins. Ils peuvent ainsi être réutilisés partout où le même besoin est exprimé.

On le voit, les répercutions liées à la qualité de conception sont multiples. Un code bien conçu sera facilement compréhensible, réutilisable, maintenable et évolutif. En conséquence les anomalies dues aux incompréhensions, à la complexité seront réduites.

Le facteur humain est ici primordial. En effet, l'expérience joue un rôle important dans la conception pour appréhender les problèmes et les résoudre au plus tôt et le mieux possible sans perte de temps.

4.2 Le logiciel évolue, toujours, souvent

> ***Un logiciel est fait pour durer et évoluer.***
> ***On estime que 60 à 80% des développements ont lieux après la première mise en production.***

Les applications évoluent pour s'adapter aux besoins sans cesse mouvants des utilisateurs et du marché. La compétitivité de la société en dépend.

On comprend alors la nécessité d'intégrer dès la conception la capacité d'évolution et d'adaptation. Néanmoins, une conception, aussi bonne soit elle, devra elle aussi s'adapter pour répondre aux nouveaux besoins. Pour ce faire, le « refactoring », opération de maintenance visant à revoir une partie de la conception ou du code pour lui permettre de supporter les modifications à venir, améliorer sa lisibilité, simplifier sa maintenance, doit être pleinement intégré au processus de développement.

Contrairement aux idées reçues, « refactoring » ne veut pas dire refaire. C'est une opération nécessaire à toute application qui évolue. Sans « refactoring », l'application perd son aspect modulaire, évolutif, en greffant divers cas particuliers, modifiant les responsabilités d'un composant pour lui faire réaliser de nouveaux besoins. (cf. chapitre 8.1).

Quid des régressions liées au « refactoring » ? Revoir continuellement le code et la conception peut s'avérer dangereux. Nous hésitons souvent à modifier ce qui fonctionne. Les tests et leurs automatisations prennent alors une ampleur toute particulière. Ils sécurisent, en effet, les phases de modification et de « refactoring ». Ils permettent de faire vivre le logiciel sans craindre les régressions. Sans ces tests, les applications sont amenées à se figer rapidement. Dans les processus de développement itératifs, le projet est découpé en phases très courtes à l'issue desquelles une nouvelle version incrémentée est livrée. La maintenance commence ici très tôt, dès la première itération. Le « refactoring » et les tests font ainsi parti intégrante du développement.

4.3 La conception orientée objet

La programmation orientée objet, offre des possibilités uniques sur la tolérance aux changement des applications. L'abstraction et le polymorphisme, par exemple, permettent de dé-corréler les traitements de leur définition et par là même de définir plusieurs implémentations en fonction du contexte sans impacter les appelants. Ce type de conception permet de limiter les efforts nécessaires à l'implémentation d'une nouvelle fonctionnalité, et les risques de régression associés. Mais utiliser un langage de programmation objet ne signifie pas programmer en objet et, malgré le succès spectaculaire des langages objet, le niveau de conception réellement objet sur les projets reste bien souvent insuffisant. Cela nécessite d'avoir un recul suffisant pour appréhender le problème et l'implémenter de façon moins systématique, plus abstraite. Il est plus facile d'écrire du code linéaire, comme nous le ferions avec un langage procédural, et d'ajouter des conditions, pour faire évoluer son comportement, que de penser objet. Le revers viendra après avec l'ajout de fonctionnalités, les adaptations de l'existant. Le code se traduira le plus souvent par des méthodes de plus en plus longues, des enchainements de conditions interminables dans lesquels le développeur, même expérimenté se perdra facilement, engendrant une explosion des temps de développement et des anomalies plus fréquentes. Il existe bien des « Design Patterns[5] », permettant de répondre à un certains nombre de besoins bien connus, mais leur connaissance n'est pas suffisante. Il est en effet nécessaire, pour réaliser une conception adaptée, de maîtriser certains des principes fondamentaux de la conception objet :

5 Design Pattern : modèle de conception destiné à résoudre les problèmes récurrents par une solution standard.

- Principe Open Close : une nouvelle fonctionnalité doit pouvoir être ajoutée en conservant le plus possible le code existant.
- Loi de Demeter ou principe de connaissance minimale : un objet doit faire aussi peu d'hypothèses que possible sur la structure interne de quoi que ce soit d'autre. En d'autres termes, « Ne parlez pas aux inconnus ! ». L'objectif est de limiter le couplage entre les objets et permettre ainsi de faire évoluer une partie du système avec le minimum d'impact sur le reste.
- Et aussi : les principes de substitution de Liskov, de cohésion, d'abstraction des éléments stables, etc. L'objectif, ici, n'est pas de détailler l'ensemble des principes et pratiques liées à la conception objet. Nous vous invitons, par conséquent, à consulter la littérature très complète sur le sujet.

La puissance de la conception objet, sa capacité à supporter efficacement le changement, émerge de la bonne compréhension de ces principes et de leur application et non de l'utilisation du langage de programmation. Il convient, pour tirer pleinement parti des avantages des langages objet de bien comprendre leur fonctionnement et les principes qui les sous tendent. Plus complexe au premier abord, la conception orienté objet peut paraître lourde et longue à mettre en oeuvre. Pourtant l'expérience des concepteurs, et le gain engendré par la capacité à supporter le changement, la rendent cependant très efficace et adaptée aux applications d'entreprise d'aujourd'hui.

4.4 La conception au coeur du processus de développement ?

Initiée par Eric Evans en 2004, un processus de développement fait le pari de mettre la conception du métier au centre de la réalisation avec la modélisation comme outil de communication et d'échange. Il s'agit du « Domain Driven Design », littéralement conception pilotée par le domaine[6]. Faut-il tout modéliser ? Pour qui ?

> ***La philosophie derrière l'expression « domain-driven design » (DDD) est de placer son attention sur le coeur de l'application, et se concentrer sur la complexité du domaine.***

Le DDD consiste en un ensemble de bonnes pratiques pour la création d'applications d'entreprise à partir d'un modèle de domaine. Avec lui, nous créons des modèles sur des problèmes métier et l'environnement technique comme la persistance, l'interface utilisateur ou la gestion des messages viennent ensuite. Le domaine doit être compris car c'est l'élément central du système qui crée la différence de l'entreprise face à la compétition.

Mais, pour créer un modèle du domaine qui capture les problèmes réels et les implémenter, il faut être en mesure de comprendre parfaitement le besoin et le traduire. Pour cela,

> ***les experts du domaine et les développeurs communiquent continuellement en utilisant les concepts introduits dans le modèle.***

Si une idée ne peut être facilement exprimée, cela indique qu'un concept manque, ou est mal formulé, et l'équipe travaille ensemble pour l'identifier. Une fois identifié, il en découlera une nouvelle notion, un nouveau champ de formulaire ou une nouvelle colonne en base de données par exemple.

Un autre élément important du DDD est de bien séparer les responsabilités. Comme indiqué plus haut, le coeur du domaine ne s'intéresse pas aux aspects de persistance ou de présentation. Des composants

6 Domaine : sujet d'application auquel l'utilisateur applique un programme.

dédiés vont prendre en charge ces aspects, qui bien qu'importants sont annexes et ne doivent pas être imbriqués dans le domaine. Ces enrichissement techniques environnementaux peuvent être ajoutés, supprimés, remplacés sans remettre en cause la logique métier du domaine. Le logiciel peut évoluer, s'adapter aux besoins du marché (accès Web, SOA, etc.) plus facilement et sans risque de dégrader le coeur de l'application. De plus, il est possible de tester simplement le domaine puisqu'il est indépendant de l'environnement technique.

En associant une conception soignée sur le coeur du métier et une communication continue entre experts du domaine et développeurs, le DDD propose un cadre à la conception et une remise en ordre des priorités.

Il en découle des applications pour lesquelles les problématiques métiers comportent peu d'erreur car mieux comprises par l'ensemble des intervenants et mieux pensées. Le code est plus simples à comprendre, sans mélange entre le métier et la technique ; il évoluera plus facilement.

5 Qualité du code

Pour un développeur, la qualité d'une application réside dans la capacité de son code à être lu, compris et repris par d'autres développeurs. Un code de mauvaise qualité sera plus difficile à maintenir et source d'anomalies. L'incompréhension est, en effet, une cause de bogues non négligeable mais peut aussi devenir une surcharge importante pour le projet.

5.1 Les bienfaits des normes et standards

Les normes et standards de développement permettent de s'accorder sur un ensemble de pratiques permettant d'assurer la bonne compréhension du code par l'ensemble des développeurs qu'ils soient présents sur le projet dès le démarrage ou dans le futur.

Ces normes sont souvent simples et facilement appropriées comme les conventions de nommage. CamelCase[7], Sun[8], par exemple, préconisent un certain nombre de pratiques pour le langage Java allant des conventions de nommages à l'indentation.

D'autres sont des règles de bon sens :

- Simplicité du code : un code simple et court sera facile à lire, à comprendre, à tester unitairement. Les sources de bogues seront réduites et facilement identifiables. C'est le paradigme KISS (« Keep It Simple Stupid »).
- Documentation : un code suffisamment documenté permettra une meilleure compréhension sur ce que réalise ou doit réaliser le code concrètement.
- Code mort : le code inutile pollue et nuit à la lisibilité et la compréhension. Il faut l'éliminer systématiquement. Les gestionnaires de sources permettent de mémoriser un état antérieur en cas de besoin.
- Certaines, enfin, proposent des indicateurs sur l'application de bonnes pratiques ou permettent de détecter d'éventuelles sources d'anomalies. On notera, par exemple, la détection :
- Du manque d'abstraction d'un paquetage ou des paramètres des méthodes publiques. La capacité d'évolution s'en trouve amoindrie.
- Ressources non libérées : les ressources sont bloquées et ne peuvent être réutilisées engendrant des anomalies.
- Traitement des exceptions et gestion des traces : lorsqu'une erreur se produit dans le logiciel, il est important de la traiter correctement, de fournir à l'utilisateur un message approprié et de fournir dans des traces une indication la plus claire possible sur l'évènement qui s'est produit et le contexte associé. La maintenance en sera simplifié et le délai avant correctif réduit pour l'utilisateur.

Ces pratiques doivent, pour être efficaces, être connues, comprises et acceptées de tous. Elles permettront la réalisation, à moindre frais, d'un code cohérent et compréhensible par tous.

7 Conventions de nommage CamelCase : http://en.wikipedia.org/wiki/CamelCase

8 Conventions de nommage de Sun pour le langage Java : http://www.oracle.com/technetwork/java/codeconvtoc-136057.html

5.2 Cibler, automatiser, intégrer au développement

Comment tirer le meilleur parti de ces normes de développement ? Un grand nombre de règles de développement existent et peuvent être vérifiées automatiquement. La présence de commentaire et les règles de nommage, mais également des règles a priori plus subtiles comme la simplicité du code. Des métriques simples comme le nombre de lignes de code d'une méthode ou d'une classe par exemple donne une bonne indication. D'autres, plus profondes, donnent une vision de la structure même du code. La complexité cyclomatique, développée par Thomas McCabe, mesure la complexité du code en évaluant le nombre de chemins possibles du programme. Au delà de 7 chemins, il devient difficile de comprendre, maintenir et tester le code. Il faut alors repenser l'implémentation ou la conception de cette portion de l'application.

L'exemple suivant visualise les chemins de deux méthodes, l'une simple, l'autre plus complexe :

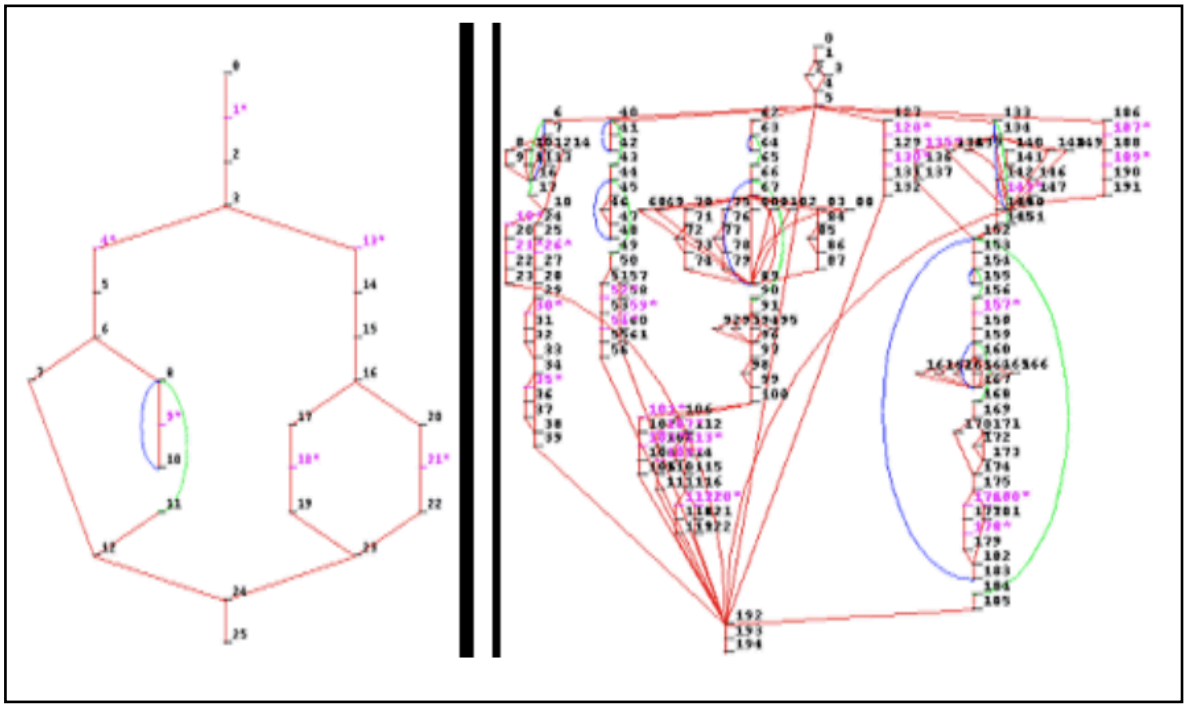


Illustration 5 : Comparaison de complexité cyclomatique

Source : http://www.mccabe.com/iq_developers.html

On visualise clairement la différence en terme de difficulté de compréhension et de test.

Cette métrique, permet en respectant un seuil prédéfini, de conserver un code simple et par conséquent engendre une multitude d'actions de mise en qualité. Un dépassement du seuil, nécessitera de repenser le code, de revoir la conception et donc de l'améliorer continuellement. Il sera également plus facile à tester et donc moins propice aux anomalies.

Il sera également très intéressant d'intégrer les règles permettant de détecter directement des causes de dysfonctionnement comme le risque de référence à un objet non initialisé par exemple. Néanmoins, il faut éviter de tomber dans l'excès de normes qui nuisent à la productivité des développeurs.

> ***Il est important de s'accorder sur un ensemble restreint de pratiques, comprises par tous, et comportant les règles les plus fondamentales et utiles en fonction de l'architecture.***

Pour gagner en efficacité, ces pratiques doivent être vérifiées à l'intérieur même de l'environnement de développement. Le développeur doit être alerté au plus tôt du non respect d'une norme dans son code. Il pourra alors très rapidement le corriger. Le respect des normes doit générer une surcharge la plus minime possible pour un développeur en phase d'apprentissage pour rapidement devenir un réflexe.

Un bon nombre d'outils permettent de vérifier le bon respect de normes et standards de développement et détecter des risques d'anomalies. Il conviendra de choisir le ou les outils offrant le meilleur compromis richesse des détections/efficacité en optant par exemple pour :

- Un ou plusieurs outils de détections immédiates intégrés dans l'IDE (Checkstyle, PMD, Findbugs, etc.).
- Si besoin, un ou plusieurs outils de détections plus poussées exécutable séparément pour les détections complexes d'architecture par exemple.
- Un outil de consolidation permettant d'avoir une vision d'ensemble cohérente de la qualité des applications (Sonar).

Ce dernier va, en outre, assurer l'uniformité des règles utilisées sur un ensemble de projets. Il sera alors possible de connaître la qualité du parc applicatif. La consolidation des indicateurs permet de cibler rapidement les points à améliorer en priorité. La répartition par catégorie (fiabilité, maintenabilité, etc.) ou criticité (critique, majeur, etc.) permet d'axer ses effort en fonction de l'importance du moment.

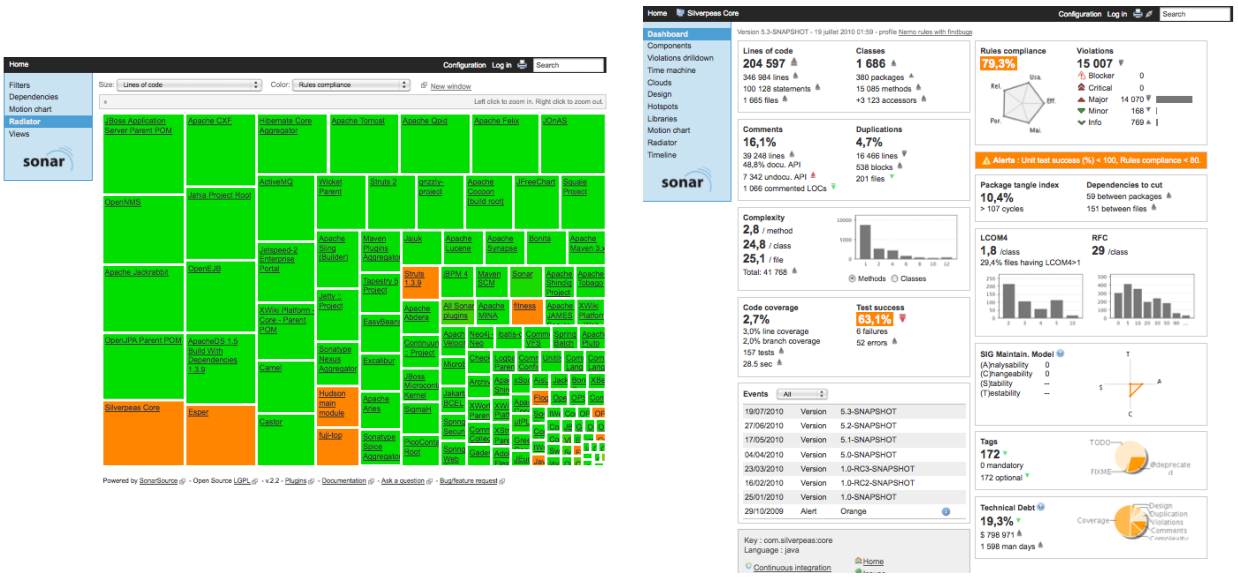


Illustration 3 : Tableaux de bord de suivi de la qualimétrie

Source : http://www.sonarsource.org/

Un autre aspect intéressant de ce type d'outils est d'apporter une dimension temporelle à l'analyse. Nous ne sommes plus dans l'analyse instantanée mais dans le suivi. On est en mesure de vérifier l'impact d'un plan d'action qualité sur la durée.

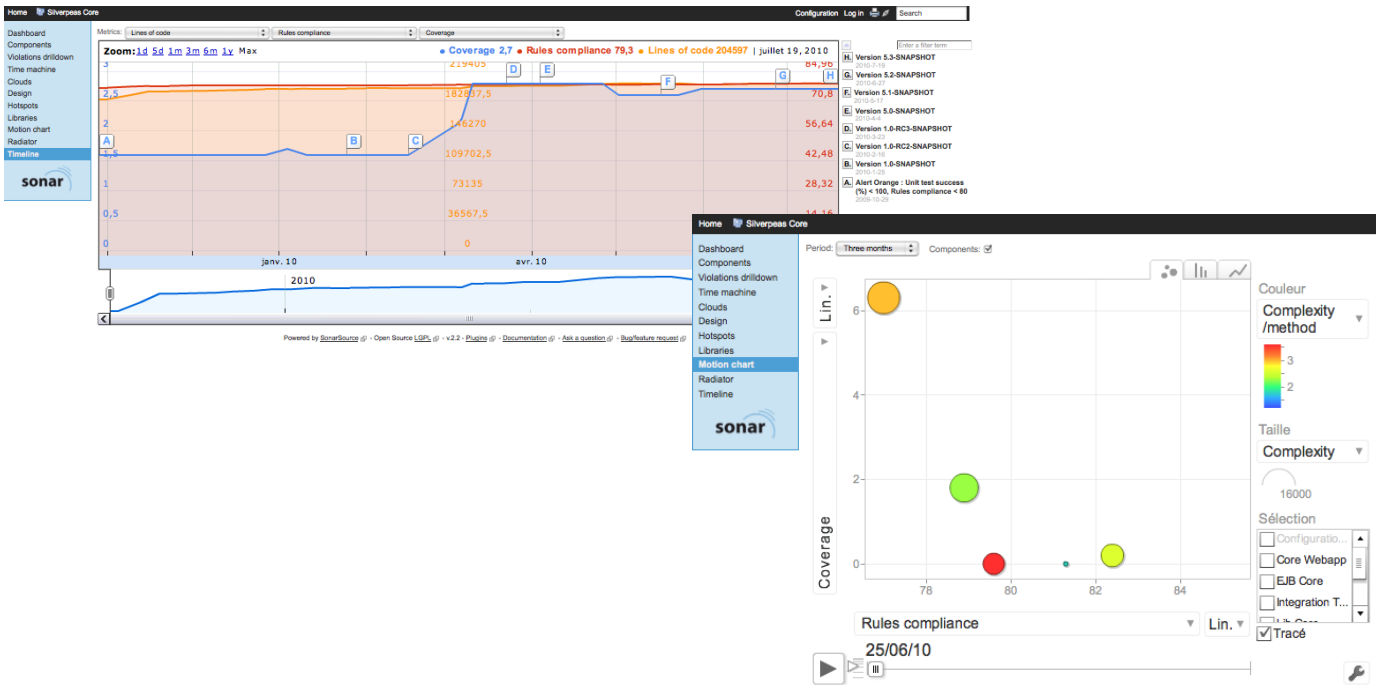


Illustration 4 : Graphes de suivi de la qualimétrie en fonction du temps

Source : http://www.sonarsource.org/

5.3 Relecture par les pairs

Outre les normes de développement et l'outillage permettant de les vérifier à moindre coût, un moyen efficace de s'assurer de la bonne écriture du code et de sa bonne compréhension par un autre développeur est la relecture par un pair.

Un regard neuf, permet de déceler rapidement des erreurs simples, de s'assurer que le code est lisible, les commentaires compréhensibles et suffisants, et de lever des interrogations. Les questions posées par une personne n'ayant pas réalisé le développement sont souvent perspicaces et permettent de résoudre des problèmes de conception, de codage, de compréhension en amont.

5.4 Programmation en binôme

La programmation en binôme ou « pair programming » est une méthode de programmation dans laquelle deux développeurs travaillent, en binôme, sur la même partie de code, sur un même poste de travail. Le premier, aussi appelé le « pilote », écrit le code tandis que l'autre, le « copilote », l'assiste, suggérant de nouvelles possibilités ou décelant d'éventuels problèmes. Les rôles sont inversés régulièrement pour conserver la dynamique et la concentration de chacun. Les deux développeurs collaborent dans toutes les phases du développement, de la conception aux tests.

Cette collaboration engendre une qualité supérieure avec des programmes plus courts, mieux conçus, avec moins d'anomalies. Plusieurs points de vues, alternatives, sont confrontés continuellement avec, au final, une conception plus simple, plus maintenable. Le « copilote » aide également à déceler le manque de lisibilité, les anomalies, la non application de normes et standards au plus tôt, offrant une réactivité et par conséquent un gain important sur le coût de la non qualité.

La programmation en binôme offre, en outre, l'avantage d'accroître la qualité du logiciel sans impacter les délais de livraison. Bien que contre-intuitif, deux développeurs travaillant sur un même poste vont produire autant de fonctionnalités que s'ils travaillaient séparément, mais avec une qualité bien meilleure.

Un dernier point en faveur de cette pratique est le partage de connaissances et la montée en compétence. En effet, le travail en binôme permet de partager la connaissance aussi bien technique que fonctionnelle sur un sujet. Cette connaissance est, au fur et à mesure, partagée par l'ensemble de l'équipe, réduisant les risques liés au départ d'un membre. De plus, l'expérience de l'un bénéficie à l'autre et vice versa. Chacun, apprend des techniques et de l'expérience de l'autre et, au fil du temps, de l'ensemble des développeurs, contribuant ainsi à l'accroissement du niveau de maitrise général de l'équipe.

Cependant, cette technique, pour être efficace, nécessite une adhésion forte des participants. Certaines personnes acceptent mal le regard de l'autre, se sentent inutiles lorsqu'ils n'écrivent pas de code, préfèrent travailler seul. Il faut bien prendre en considération les sensibilités de chacun et trouver le mode de fonctionnement qui convient le mieux.

5.5 Conclusion

Assurer un code de qualité est un moyen simple et efficace de limiter la dette technique d'un logiciel. Intégré à l'environnement de développement, les outils permettent aux développeurs de respecter les normes et standards avec un maximum d'efficacité. Néanmoins, un logiciel bien codé, dans le sens où il respecte les règles, ne veut pas dire qu'il est de qualité. Les normes offrent simplement un cadre et des outils permettant de faciliter la mise en qualité et la cohérence globale du code. La programmation en binôme, enfin, concentre les savoirs, les pratiques et l'attention de deux personnes sur un même développement et permet ainsi d'améliorer la qualité du code produit.

6 Intégration continue : vers l'industrialisation

Littéralement, faire de l'intégration continue, signifie faire l'intégration de l'application dans son ensemble, code source et ressources, continuellement.

Il s'agit, à l'origine, d'une méthode issue des commandements de l'Extreme Programing. Partant du constat que la phase d'intégration des projets est longue et coûteuse, l'idée est de la faire en continue pour détecter les problèmes, et par conséquent les corriger, au plus tôt. Pour permettre l'exécution d'une phase d'intégration quotidienne, il faut avant tout penser à l'automatisation. Si l'intégration était testée manuellement, elle finirait par occuper l'équipe de développement à plein temps ce qui est totalement contre-productif.

La réponse à ce problème est la mise en place d'une chaîne d'intégration. C'est une chaîne de processus automatiser visant à valider l'application depuis les sources jusqu'à l'installation finale. Elle contient dans l'ordre :

- La récupération des sources à jour
- La compilation des sources
- L'exécution des tests unitaires
- L'exécution des tests d'intégrations

Un des but principaux étant de découvrir les bugs et régressions au plus tôt, il est important de la lancer au moins une fois par jour. Idéalement, elle doit faire partie intégrante du système utilisé par les développeurs pour construire l'application. Pour guidez vos pas vers l'intégration continue, il vous suffit d'observer six régles simple.

6.1 Gérer les sources

Le premier chose à mettre en place, est un système de gestion de versions du code source. L'intégralité des sources et ressources nécessaires à la construction de l'application doivent-être stockés dans le dépôt. Utilisé comme référence, il permettra aux développeurs de toujours travaillé sur la même version de la base de codes. L'utilisation d'un système de gestion de version diminue le temps passé à fusionner le travail des développeurs et réduit les risques inhérent à la-dite fusion. La gestion des sources offre, en plus, une garantie de flexibilité avec la possibilité de revenir aux versions précédentes d'un code ou de l'application entière.

Bien qu'il existe de nombreuse façon de gérer le dépôt de sources, quelques règles élémentaires à l'usage des développeurs sont à suivre :

- Mettre régulièrement ses sources à jour.
- Mettre à jour ses sources avant d'envoyer des modifications sur le dépôt.
- Envoyer les modifications des sources au moins une fois par jour.
- Marquer dans le dépôt les versions des sources livrés.
- Ajouter des commentaires lisibles aux commits.

D'une façon générale, plus les commits sont fréquents, plus la période d'intégration des sources est courte, et plus un retour en arrière, suite à la détection d'une régression par exemple, sera ciblé, simple, et rapide.

6.2 Automatiser la compilation

L'ensemble des processus permettant de construire l'application sous une forme livrable doit-être automatisé. Le développeur doit pouvoir faire une seule action (lancer un script, cliquer sur le bouton approprié de son IDE), pour construire et lancer l'application. L'objectif est là encore d'économiser le temps du développeur qui, en principe, construit l'application plusieurs fois par jour. D'autre part, la compilation de l'intégralité des sources modifiées permet de vérifier et garantir, la validité et la cohésion du code produit. L'automate doit-être suffisamment intelligent pour ne re-compiler que ce qui est nécessaire à savoir les fichiers modifiées ou ajoutées et les sources qui en dépendent. Il s'agit surtout de garantir un temps de construction court. Avec l'intégration continue, l'automate est utiliser quasi-tout le temps, il est donc utile de le maintenir pour que son temps d'exécution soit et reste le plus court possible.

6.3 Intégrer les tests

La chaine de construction de l'application doit exécuter les tests unitaires. Cette étape est obligatoire pour construire et tester l'application. Ainsi les tests sont exécutés à chaque nouvelle compilation, ce qui permet de trouver les régressions au plus tôt. Attention, pour que cette étape prenne tout son sens, il est important que le moindre échec dans les tests unitaires suffise à bloquer la construction de l'application. La correction des régressions est une priorité absolue de l'intégration continue. De manière plus générale, tout ce qui bloque la chaîne d'intégration doit-être traité en priorité.

Les tests d'intégration font aussi parti de la chaîne. Chaque développeur doit pouvoir les exécuter en local sur son poste de travail. Ils sont exécutés après la construction complète de l'application (compilation, tests unitaires et packaging). L'exécution des tests d'intégration est un mal nécessaire, mais cela peut s'avérer long. Pour éviter aux développeurs d'y passer trop de temps, leur exécution n'est obligatoire que lors de l'envoi des sources modifiés.

En général, les tests unitaires prennent peu de temps et doivent être exécutés le plus souvent possible, au plus proche des modifications. En fonction des tests, ceux d'intégrations par exemple qui peuvent s'avérer long, les exécuter systématiquement peut être coûteux en ressources machine. Pour remédier à ce problème, nous acceptons dans certains cas de réduire leur fréquence pour n'en faire, par exemple, qu'un par nuit, incluant également de la qualimétrie et d'autre rapports sur le projet. Nous perdons un peu en réactivité sur certains types d'anomalies, mais on réduit le coût de l'infrastructure qui peut alors être plus facilement mutualisée.

6.4 Maintenir les versions stables

Tous le monde doit pouvoir récupérer la dernière version intégrée de l'application. Par version intégrée de l'application, entendez, la version de l'application la plus récente ayant passé avec succès les étapes de la chaîne d'intégration continue. Cela permet d'obtenir un aperçu rapide des développements en cours. Ces versions pourront servir pour, des démonstrations, des tests, ou toute autre raison.

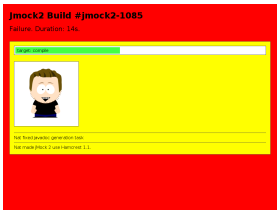
Mettez en place un dépôt pour stocker les paquets de l'application construite par la chaîne d'intégration. Dans ce dépôt, il faudra archiver les versions temporaires de l'application construite à chaque commit. Il est inutile de les maintenir toutes, conservez seulement les dix dernières, ou celles des huit derniers jours. Archivez aussi les versions livrées de l'application. Si vous utilisez un mode de développement itératif, sauvegardez toujours la version de fin d'itération. Ce dépôt peut coûteux à mettre en place, économise

grandement le temps des développeurs en leur évitant la re-construction d'une version antérieure depuis les sources.

6.5 Faire connaître l'état

Tous le monde doit connaître l'état de l'application en développement et les derniers changements qui y ont été apportés. Il faut avant tout informer de l'état général de la chaîne d'intégration continue. Si elle est cassée, les développeurs doivent le savoir rapidement pour pouvoir prendre leurs responsabilités. Que l'équipe fasse tout pour résoudre le problème ou non, le tout est que la réaction soit rapide.

Plusieurs solutions sont envisageables des lampes de couleurs au système d'e-mailing en passant par le Nabaztag et la cloche.



Maintenez un historique plus complet qui suivra l'état de la chaîne dans le temps. Faîtes par exemple des points rouge ou vert sur un tableau ou un calendrier. Une majorité de rouge, révèle un projet en mauvaise santé. Certaines solution permettent de recouper ces informations avec les commits du dépôt de sources pour tracer la ou les modifications à l'origine du problème et son auteur. La mise en place d'un système de notification responsabilise les développeurs au regard du statut globale de l'application et plus seulement des fonctionnalité dont ils ont la charge. Avec le temps les échecs se feront de plus en plus rare.

6.6 Déployer l'application

Si vous respectez les règles des tests d'intégration mentionnées précédemment, vous devez avoir plusieurs environnements de test à disposition. Les développeurs jonglent continuellement entre les différents environnements. Le déploiement de l'application n'échappe pas à la règle,de l'automatisation. En fin de chaîne, l'application peut donc être déployée sur un environnement de test pour le développeur, ou de qualification pour la recette. Si vous automatisez le déploiement en production, pensez à automatiser aussi le roll-back en cas d'erreur. Assurez vous que le script est capable de (re)-déployer une version antérieure réputée stable de l'application.

Ce dernier maillon de la chaîne réduit le temps de validation et favorise la mise à jour régulière de l'application en production.

6.7 Utiliser un serveur d'intégration continue

Pour faciliter la mise en place de l'intégration continue, utilisez un serveur d'intégration continue. Il en existe plusieurs sur le marché libre ou propriétaire dont nous ne débattrons pas le choix. La plupart du temps, il s'agit d'une application web installée sur un serveur qui sera dédié à la compilation et à l'intégration du ou des projets. Ce serveur surveillera les activités sur le dépôt de sources et lancera la chaîne d'intégration à chaque modification par exemple. Le serveur ne se contente pas d'exécuter la chaîne, il doit aussi fournir un retour par affichage et notifications de l'état de la dernière exécution. En répondant à la grande majorité du cahier des charges, le serveur économise un temps précieux dans la mise en place de l'intégration continue. C'est aussi une assurance de la qualité des versions, qui seront toutes construites dans le même environnement et uniquement à partir des sources du dépôt de référence. Il n'est pas rare d'une part que les développeurs utilisent des postes de travail hétérogènes et d'autre part que la version des sources dont ils disposent en local ne soient pas la copie conforme de celle du dépôt. C'est enfin, une économie de temps pour les développeurs qui n'ont plus la charge de construire les versions livrables.

6.8 Synergie des pratiques qualité

L'Extreme Programming, méthode de développement agile, repose sur l'ensemble des pratiques énoncées précédemment (tests, conception, refactoring, programmation en binôme, normes de développement, intégration continue). Comme l'illustre le schéma qui suit, celles-ci sont intimement liées et la mise en oeuvre de l'une sans les autres fait perdre la cohérence et l'efficacité de l'ensemble. L'intégration continue, sans tests unitaires par exemple, perd une part importante de son intérêt. Nous ne serons pas en mesure de détecter au plus tôt les régressions introduites par le code qui vient d'être soumis.

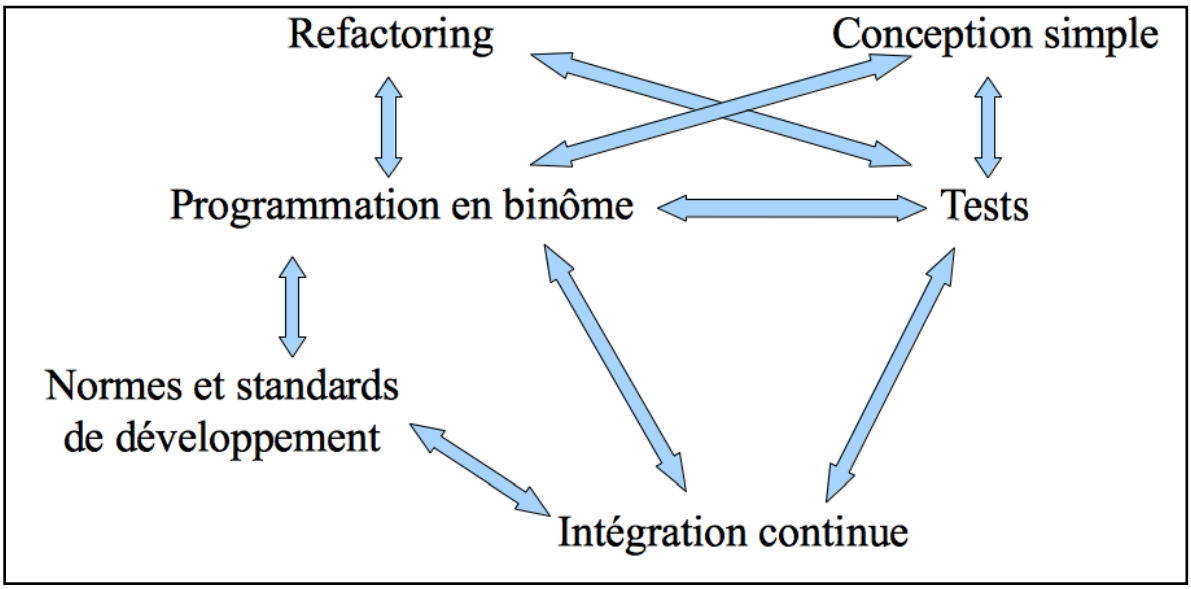


Illustration 8: Interdépendances des pratiques qualité

De la même manière, une conception simple se traduira par des tests simples à écrire, facilitant l'atteinte d'une couverture maximale. La programmation en binôme rendra le refactoring plus efficace et sûr, car reposant sur l'intelligence et l'expérience de deux personnes. De même, les normes et standards seront mieux appliquées, chacun veillant à sa bonne application. Le refactoring sans test est risqué et ne sera souvent pas réalisé par peur des régressions.

Le point important à considérer ici est, qu'en dehors du processus de développement utilisé, la qualité repose non pas sur une, mais sur un ensemble de pratiques interdépendantes qui assureront l'efficacité du processus d'amélioration continue. Les tests seuls ne garantirons que l'adéquation des résultats fournis avec ceux attendus mais n'assurera ni l'évolutivité, ni la maintenabilité du logiciel. A l'inverse, un code peut respecter parfaitement les normes et standards de développement, cela ne signifie pas qu'il est fiable, robuste, et qu'il répond au besoin fonctionnel exprimé.

7 L'importance de l'humain

Toutes ces pratiques pour être correctement appliquées, doivent, d'une part, être comprises et assimilées mais aussi et surtout avoir l'adhésion des membres des équipes de développement ainsi que de l'encadrement.

7.1 Sensibiliser, convaincre

Si les personnes ne sont pas convaincues par les méthodes à mettre en place, ils ne les utiliseront pas, ou de manière inefficace. Il est facile d'écrire des tests qui ne vérifient rien, mais cela ne sert à rien non plus. De même, la relecture de code, la programmation en binôme, n'a d'intérêt que si les personnes qui les exécutent sont réellement impliqués. Subir un processus amène souvent à le contourner, à le transgresser ou en retirer toute substance par une application erronée.

De la même manière, si la direction n'a pas conscience de l'investissement nécessaire pour atteindre les objectifs de qualité et du retour sur investissement qu'il génère, les équipes auront du mal a réaliser des applications pérennes et de qualité. Les pressions, l'incompréhension entre équipes de développement et direction s'accroissant avec la dégradation de l'application induite par le manque d'investissement.

Il est donc primordial que l'ensemble des personnes impactant les projets soient sensibilisées aux problématiques de qualité, convaincues par leur intérêt afin de trouver les pratiques qui seront comprise et acceptées par tous.

7.2 Formation, expérience et partage

7.2.1 Formation et tutorat

La plupart des erreurs de conception ou d'implémentation sont faites, dans un premier temps, parce que les personnes qui les mettent en oeuvre manquent d'expérience sur un sujet donné tout en étant contraintes à une réalisation rapide ; et dans un deuxième temps, par la dégradation de modules existants déjà mal conçus et pour lesquels le « refactoring » fait peur.

La formation, le tutorat ou l'accompagnement par des personnes expérimentées permettent de démarrer dans de bonnes conditions et de la bonne façon sur des technologies nouvelles ou mal maitrisées par l'ensemble de l'équipe. De plus, une équipe de développement est souvent hétérogène en terme de connaissance ou d'expérience. Le partage au sein de l'équipe est alors primordial. Une erreur fréquente est la conservation de l'information et le manque de partage. Une personne est la seule à maitriser un sujet, elle se rend indispensable. Le problème est que lorsqu'une autre personne intervient sur le sujet, souvent en urgence, le résultat peut être désastreux. Il faut briser l'individualisme qui forge des équipes de micro-spécialistes pour aller vers du partage continu de connaissances, source d'enrichissements mutuels. Il n'est d'ailleurs par rare que le spécialiste en question ait une vision erronée sur un sujet qu'il est sensé maitriser et qu'un regard neuf, un échange productif permette d'améliorer l'application. La programmation en binôme offre un espace privilégié à l'échange des connaissances et des points de vue. Sa généralisation ou son encouragement est un vrai plus pour la maturité technique d'une équipe et le maintient des compétences sur un projet (cf. chapitre 5.4).

7.2.2 Les risques de la spécialisation

Comme nous l'avons vu dans le paragraphe précédent, un module complexe, peu documenté, voir mal conçu, pourra être repris par son concepteur et évoluer, avec plus ou moins de facilité. En revanche, un autre intervenant aura beaucoup de difficultés à faire la même chose et génèrera certainement des anomalies par incompréhension ou maitrise partielle du composant. L'application, sa maintenance, son évolution reposera de plus en plus sur les personnes qui l'ont créée faisant courir un risque important au projet. De plus, par souci de rentabilité, il arrive souvent qu'une seule personne ne maîtrise réellement qu'une partie de l'application, accentuant le problème.

Qu'advient-il alors lorsque des personnes quittent le projet, sont malades, en vacances ? Un transfert de compétence tardif est souvent peu efficace, le projet se retrouve vulnérable, les parties de l'application impactées vont évoluer difficilement, avec un risque accrue en terme de régressions et d'anomalies.

Comment s'en prémunir ?

- Il est important de partager les connaissances tant fonctionnelles que techniques avec les autres membres de l'équipe. Le « pair programming » permet à deux personnes d'acquérir la même vision technique d'une fonctionnalité (en plus des aspects qualitatifs que cette pratique apporte). Elle est particulièrement importante et efficace pour permettre à un nouveau développeur d'intégrer un projet.
- Documenter « suffisamment » le code et les besoins fonctionnels auxquels il répond. Ici, tout réside dans le juste degré de documentation nécessaire pour comprendre et pouvoir intervenir sans trop de difficulté dans l'application. La documentation doit être utile et utilisable. Une documentation obsolète n'a pas d'intérêt. Elle doit vivre et par conséquent être facilement accessible, simple et compréhensible. Un document de 500 pages au format papier aura peu d'intérêt (sauf obligations contractuelles) car ne sera pas utilisé.
- Une conception de qualité rendra l'application plus facile à comprendre, maintenir et faire évoluer par les nouveaux intervenants même sans transfert de compétence.

Ces pratiques permettent de limiter l'impact du départ d'un membre de l'équipe et ainsi réduire les risques associés pour le projet. La connaissance n'est pas perdue car partagée, le code peut être compris par d'autres. Néanmoins, le départ d'une personne aura toujours un impact sur la productivité. Le remplacement d'une personne s'accompagne d'un délai d'adaptation. Le transfert de connaissances, la formation du nouvel arrivant sur l'environnement tant technique que fonctionnel aura des répercutions sur la productivité de l'ensemble de l'équipe. L'ambiance, la dynamique générale repose, pour beaucoup, sur l'esprit d'équipe qui se crée avec le temps. Les changements fréquents des membres du projet sont par conséquent à éviter et le management joue ici un rôle important. Bien être, qualité de l'environnement de travail, chalenges et diversité technique, perspectives d'avenir, évolution et formation, salaire, sont autant d'axes contribuant à maintenir la motivation des individus permettant de conserver la stabilité nécessaire aux projets et maintenir la productivité et la qualité générale.

7.3 Impatience et enthousiasme face à la qualité

Un aspect important pour la productivité d'une équipe est sa motivation, son enthousiasme, son envie de voir avancer le projet. Cependant, cette dynamique, si elle n'est pas maîtrisée, peut vite devenir de la précipitation. Très naturellement, les tâches qui ne se voient pas ou ne font pas avancer le projet fonctionnellement ne sont pas satisfaisantes, voire stressantes. Chacun veut se rassurer sur sa capacité à réaliser la tâche à laquelle il est affectée et pour cela, il faut voir le résultat au plus vite. Cette volonté d'avancer vite s'accompagne généralement d'un manque de réflexion, de conception, de tests. Car une fois le résultat obtenu, il reste le plus « rébarbatif » à réaliser : « refactoring » du code (puisqu'il a été écrit en

mode « quick and dirty »), écriture des tests unitaires (le TDD[9] n'est pas la règle dans ce cas), traitement des cas limites (souvent oubliés), documentation (remise à plus tard). Cette étape qui permettrait finalement d'obtenir un développement de qualité est malheureusement faite avec moins de passion et négligée, voire omise.

Comme nous venons de la voir, une qualité importante comme l'enthousiasme peut devenir néfaste pour la qualité d'une application. Pour répondre à cette problématique, nous pouvons nous inspirer des pratiques issues des méthodes agiles comme Scrum[10] en définissant clairement ce que signifie « terminé » pour une tâche, de partager cette définition et de veiller à ce que chaque tâche remplisse ses critères de validation. Nous pourrons, par exemple, définir une fonctionnalité comme terminée lorsqu'elle est développée bien sûr, qu'elle s'exécute et a pu être validée fonctionnellement, mais également que les tests unitaires associés ont été écrits et passent avec succès, qu'une relecture a été effectuée par un pair, que les normes et standards de développement spécifiées sont respectées, etc... Cela permet de réduire les risques en forçant la réalisation, même après coup, des pratiques d'assurance qualité. La pratique quasi-systématique du « pair programming » permettra également de limiter ce genre de situation, les deux intervenants se neutralisant et cherchant généralement plus à montrer leurs qualités plutôt que leurs défauts.

Une fois encore, la prise de conscience de l'importance des pratiques qualité par l'ensemble de l'équipe permettra d'accroître l'intérêt des membres sur les pratiques qualité qui ne seront plus reléguées aux tâches secondaires mais feront partie intégrante du développement jusqu'à devenir naturelles.

9 TDD : Test Driven Development (cf. chapitre 0)

10 Scrum : méthode de développement agile fédéré autour du Manifeste Agile.

8 Time to market et qualité

Dans les chapitres précédents, nous avons vu que la qualité est accessible et rentable à condition d'investir en amont : tests, conception, « refactoring ». Néanmoins, cet investissement peut être un frein à la réactivité face au marché. C'est une des principales raisons de la non qualité ou, en tout cas, de la non mise en place des processus à même de garantir cette qualité sur le long terme.

8.1 Quand qualité rime avec productivité

On oppose régulièrement qualité et productivité. Les efforts pour atteindre la qualité et les processus associés sont parfois lourds et tendent à réduire la productivité des équipes de développement. Les initiatives pour fournir rapidement une fonctionnalité sont ralenties afin de répondre aux exigences de qualité et au processus de validation.

Pourtant, même si ces arguments sont recevables, les développements faits sans pratiques qualité ont, au final, une productivité inférieure. En effet, nous confondons rapidité et production, avec réactivité et productivité. Il est possible d'être plus réactif avec un mode de développement moins contraignant et peut être plus productif sur le moment. Mais que signifie être productif ? Que prend-on en considération lorsque l'on évalue la productivité d'une équipe de développement ?

Nous l'avons vu précédemment, une défaillance en production peut avoir des répercutions importantes sur l'entreprise ; financières mais aussi en charge de correction associée. De ce fait, un développement rapide mais avec beaucoup de bogues nécessitera au final plus de temps qu'un développement plus long mais sans anomalie.

De la même façon, un développement rapide mais sans penser au « refactoring » aura tendance à alourdir le code utilisé, le rendre plus difficilement maintenable et évolutif. Un développement intégrant une phase de « refactoring » sera plus long à développer mais prendra moins de temps à faire évoluer ou à maintenir.

> ***La productivité est très relative.***
>
> ***Elle dépend essentiellement de l'instant auquel ont regarde le projet. Une forte productivité à l'instant t0 peut engendrer une productivité beaucoup faible par la suite par manque de qualité.***

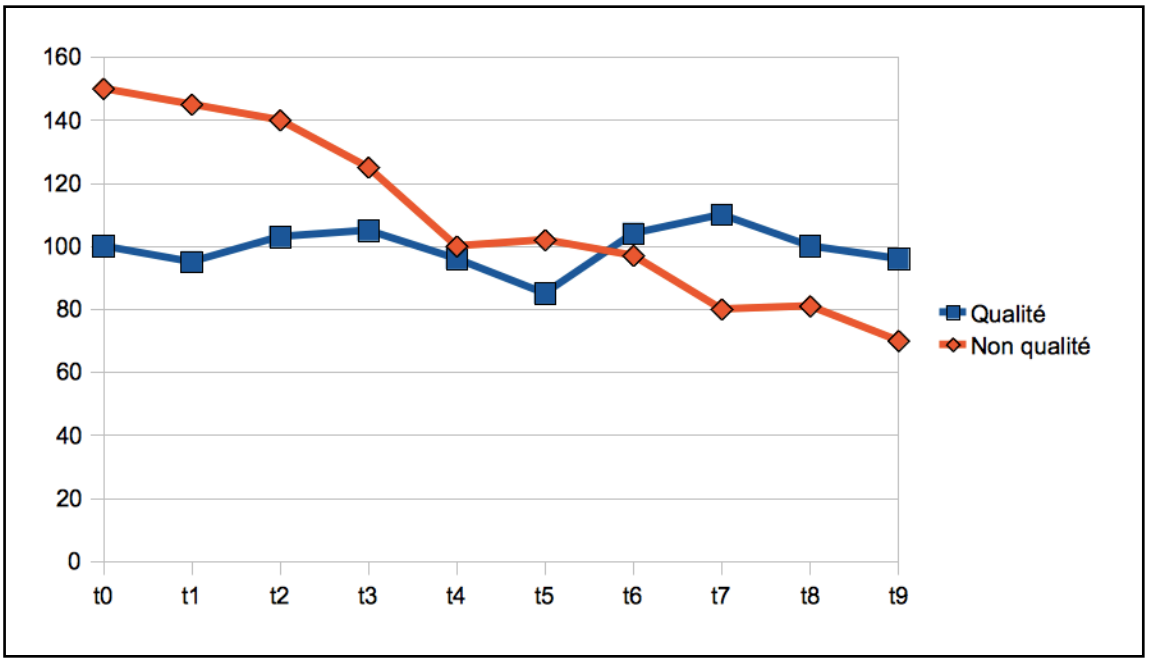


Illustration 9: Evolution de la productivité en fonction du temps

Le graphique ci-dessus, donne une vision de l'évolution de la productivité en fonction du temps suivant que l'on prenne en en compte les problématique de qualité où non. La charge de développement d'une fonctionnalité à t0 et à t9 est très différentes. Avec le temps, les équipes de développement ayant produit des applications sans se soucier de la qualité seront de plus en plus occupées à adapter l'existant à quelque chose pour lequel il n'a pas été conçu, à corriger des anomalies de plus en plus nombreuses. Dans le cas d'un processus qualité et d'amélioration continue, des phases de « refactoring » vont, certes, réduire la productivité pendant une période mais vont permettre de l'augmenter sur d'autres assurant ainsi, avec la limitation des anomalies, une relative constance dans la productivité.

Combien de fois avons nous été confronté au même dilemme : « il faudrait refaire entièrement cette partie de l'application ou pire refaire le projet de zéro ! » Pourquoi ? A trop attendre, trop accumuler de dette technique, trop pervertir le code existant pour obtenir les fonctionnalités attendues en un minimum de temps, il devient quasi impossible de le repenser en un temps raisonnable. Nous sommes alors confrontés à des projets dans les projets, avec une charge associée extrêmement importante et un risque élevé. Nous n'avons pas su faire évoluer l'application au fil du temps et maintenir la productivité. Une part non négligeable des budgets informatiques sont ainsi alloués à la refonte de systèmes existants, ou pour re faire des applications qui sont devenues impossibles à faire évoluer ou coûtent trop cher en maintenance.

> ***Comment alors concilier qualité et réactivité ?***
>
> ***Comment réduire l'impact de la qualité sur la productivité de mes développements ?***

Nous l'avons vu précédemment, un certain nombre de techniques, l'expérience et la formation permettent d'optimiser le retour sur investissement des pratiques qualité. Mais elles nécessitent tout de même un investissement humain et en temps qui peut aller à l'encontre d'un besoin urgent. Comment gérer au mieux ces urgences ?

8.2 Bon sens et expérience face au dogmatisme

La qualité à tout prix ou la qualité pour la qualité n'a que peu d'intérêt et est même contre productive. Lorsque l'on crée un logiciel, il est souhaitable qu'il soit de bonne qualité. Néanmoins, comme nous l'avons vu précédemment, le passage d'un niveau de qualité à un autre peu s'avérer extrêmement coûteux pour un gain minime. Dans un contexte concurrentiel fort, il est important de bien jauger le niveau de qualité souhaité et de mettre les moyens appropriés pour y parvenir. L'expérience des équipes et le bon sens sont les principales armes face à ce défit permanent. Elles permettent, en effet, d'isoler les composants les plus critiques, de cibler les efforts et de faire l'impasse ou de remettre à plus tard des développements ou des pratiques qualité sur une partie pour répondre aux urgences du marché au plus vite, avec une qualité optimale.

Le Lean repose essentiellement sur le bon sens et le pragmatisme. « ne pas faire quelque chose qui ne sert à rien » ; « si des éléments internes ou externes nuisent à la productivité, trouver une solution pour les supprimer ou au moins les réduire » ; « je dois tester mon code » ; « je dois concevoir avant de coder », etc. sont autant d'exemples qui ne font pas débat et qui pourtant ne sont pas appliqués.

Les méthodes agiles prennent ainsi le contre pied des méthodes de développement plus traditionnelles. Au lieu de se reposer sur un processus bien établit et rigoureux, comme CMMI par exemple, en prônant le fait que la qualité découlera naturellement de la bonne application du processus, elles prennent le parti du produit fini comme base au développement. L'objectif principal est de délivrer un logiciel, de qualité, qui réponde aux besoins des utilisateurs. Le processus, lui, émerge de cette production, de l'expérience acquise au fur et à mesure des livraisons. Pour cette raison, les méthodes agiles se basent sur des itérations courtes de développement avec des livraisons fréquentes, permettant ainsi d'acquérir rapidement, en continu, les retours permettant d'adapter, d'améliorer le processus. Bon sens et réactivité sont au coeur de ces méthodes.

Mais faire preuve de bon sens, n'est pas toujours simple. Il est plus aisé de suivre un processus bien huilé ou le chemin le plus direct plutôt que de se raisonner, de prendre de la hauteur, de réfléchir à ce qui fonctionne ou pas et à ce qui pourrai être amélioré. Une des grandes difficultés rencontrées par les équipes Scrum, par exemple, est l'application de ce bon sens et l'amélioration continue. La discipline nécessaire est bien plus importante qu'avec un processus bien établi. Le plus grand piège est de sombrer dans l'anarchie la plus complète où l'on a simplement supprimé ce que l'on n'aimait pas faire.

Les méthodes permettant la réalisation d'applications de qualité sont largement connues, leur mise en oeuvre est, elle, parfois lourde, avec un investissement disproportionné, inadaptée à l'environnement ou la maturité de l'entreprise, trop contraignante. Le bon sens aide alors à choisir ce qui va pouvoir être appliqué, à adapter ses pratiques pour gagner en productivité. Les processus, méthodes, outils qualité doivent être réévalués et adaptées constamment. Qu'apportent-ils réellement ? Sont-ils appliqués ? Correctement ? Quels sont les gains ? Peuvent-ils être améliorés ? Comment ? D'une équipe à l'autre, les pratiques qualités peuvent être appréhendées de manière très différentes, en fonction de vécu et de la sensibilité de chacun. Une méthode fonctionnant pour l'une pourra échouer pour l'autre et inversement. L'application d'un processus incompris sera contreproductif. Il ne faut donc pas s'obstiner sur un processus qui ne fonctionne pas, surtout s'il est lourd. Mieux vaut le revoir, l'adapter si cela est possible, ou en choisir un autre qui sera mieux compris, accepté par l'ensemble.

Pour citer le grand physicien Albert Einstein :

> ***« La folie est de toujours se comporter de la même manière et de s'attendre à un résultat différent. »***

8.3 Gestion des risques

Il est extrêmement important de bien estimer les risques encourus et d'apporter les efforts appropriés pour obtenir la « juste » qualité permettant de répondre aux besoins du marché à temps. Tout le dilemme réside dans la capacité à livrer une application à temps avec une qualité acceptable. Mais que signifie acceptable ? C'est tout l'enjeux de la gestion des risques.

Quel sera le préjudice d'une interruption de service ? Quelles sont les fonctionnalités critiques d'un point de vue utilisateur, métier, financier ? Un mode dégradé est-il acceptable ? Pour quels composants ? Pour quels utilisateurs ? Mes coûts de maintenance vont-ils augmenter indéfiniment ?

Doit-on, par exemple, apporter la même attention aux instruments de pilotage d'un avion qu'au logiciel de visualisation de films ? Bien que sur un long courrier, les passagers seront très sensibles à la possibilité de visionner des films ou écouter de la musique, nous pouvons estimer qu'une interruption de service de quelques minutes sera acceptable. Pour les instruments de pilotage par contre...

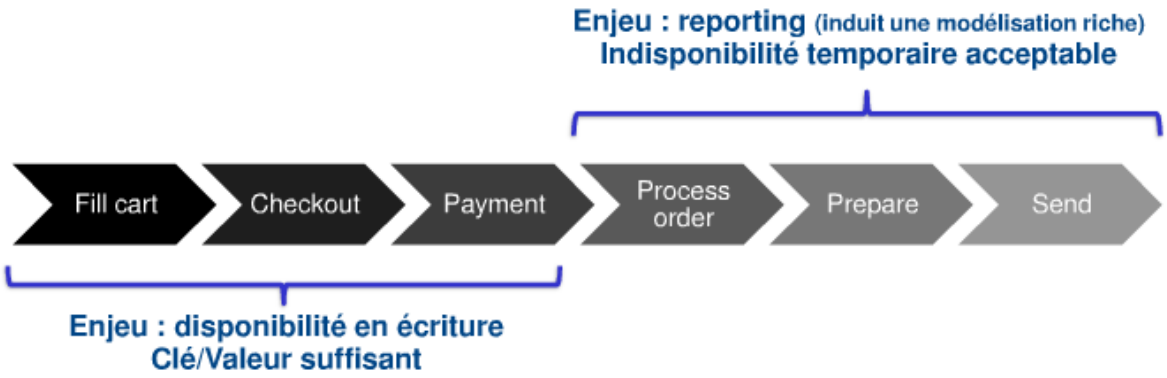
> ***On ne regarde plus directement la qualité d'une application mais plutôt les risques associés à la non qualité.***

Il est donc nécessaire de bien définir ce qui est acceptable ou non pour chaque service, fonctionnalité. Nous nous attacherons à réduire les risques les plus critiques en premier alors que les moins bloquants seront fait si le temps le permet ou remis à plus tard.

Si les risques sont bien estimés, les impacts de la non qualité seront contrôlés, minimes par rapports aux exigences des services fixées initialement.

Cela ne veut pas dire qu'il n'y a pas d'impacts, mais que ceux-ci sont maîtrisés, anticipés. Si une urgence ou un imprévu a abouti à la dégradation du processus qualité du logiciel, il est important de le rétablir après la livraison pour ne pas accumuler de plus en plus de dette technique au fur et à mesure des versions. La gestion des risques serait de plus en plus délicate. En effet, un risque de dysfonctionnement jugé faible sur une version n pourra être forte sur une version ultérieure si les modifications apportées n'ont pas été associées à des phases de « refactoring », si les tests de non régression n'existent pas, etc.

Un exemple de cette gestion des risques est l'approche prise par Amazon lorsqu'elle a revu l'architecture de son site marchant. Pour cela, elle a regardé le cycle de vie d'une commande de la navigation sur le site par un utilisateur, jusqu'à l'achat et la livraison.



Elle a identifié 2 types de qualité de service :

- Le premier ne tolère pas de défaillance, de blocage, de ralentissement, d'arrêt du service : cela correspond à la phase de navigation et de commande par l'utilisateur jusqu'au paiement. Cette phase est vitale pour l'entreprise, l'utilisateur ne doit pas avoir envie de changer de site, d'annuler sont achat, c'est le coeur du métier d'Amazon. Cette phase doit répondre parfaitement, rapidement, de manière fiable, en permanence. Nous sommes dans le domaine de le haute disponibilité.
- Le deuxième est plus souple. Il s'agit d'extraire la commande, de la préparer et de l'envoyer. Dans cette phase, la qualité est importante mais un arrêt de quelques heures n'est pas critique et sera souvent imperceptible pour l'utilisateur final.

Ce découpage a abouti à une architecture mixte, intégrant d'un côté une architecture simple, efficace et robuste pour laquelle Amazon a re-développé sa propre base de données lui permettant de répondre spécifiquement à son besoin d'accès rapide et sûr en lecture et écriture du panier utilisateur. De l'autre, une architecture plus conventionnelle offre une plus grande richesse et une capacité de reporting importante pour la facturation, le suivi des commandes, les statistiques d'achat, etc.

Amazon, a ici adapté ses efforts de qualité à la criticité des besoins pour l'entreprise. Sans pour autant laisser de côté les autres, elle a su optimiser ses investissements en se concentrant sur les phases sensibles de son métier.

Il faudra, néanmoins, faire attention à ne pas confondre gestion des risques et prise de risques. La gestion des risques est un processus à même d'estimer le risque associé à un événement donné et ses conséquences pour l'entreprise, les utilisateurs, les parties prenantes. Son objectif est de prévoir et limiter les impacts d'évènements imprévus pour répondre à un besoin de réactivité.

8.4 Gestion des priorités

Une autre façon d'adresser le problème du « Time to Market » est de gérer finement et efficacement les priorités sur les fonctionnalités à implémenter. La méthodologie agile Scrum permet de concilier réactivité et qualité composante centrale de l'agilité.

En effet, Scrum intègre, entre autres, les principes suivants :

- Des itérations courtes (sprints) de 2 à 4 semaines permettant de délivrer une nouvelle version du logiciel.
- Une liste de fonctionnalités triées par priorité par rapport aux besoins fonctionnels (backlog).
- La qualité et l'amélioration continue comme élément central.

Les fonctionnalités sont implémentées dans l'ordre des priorités. Celles apportant le plus de valeur sont implémentées en premier. A chaque itération, l'équipe de développement prend dans l'ordre les fonctionnalités qu'ils implémenteront. Par conséquent, si entre deux sprints, les besoins métier ont évolué, l'ordre des fonctionnalités suivantes peut être revu pour les prochains sprints sans impact sur les développements.

Il est ainsi possible d'adapter le logiciel au fur et à mesure des besoins du marché et de livrer des versions apportant le plus de valeur à l'entreprise. Sans changer les règles de développement et donc sans pénaliser la qualité, l'équipe est en mesure de livrer les fonctionnalités attendus au plus tôt. Contrairement à un processus de développement « classique » où le besoin est exprimé bien en amont avant d'être réalisé d'un trait, le besoin peut évoluer tout au long du développement permettant ainsi une meilleure adéquation avec

le besoin réel au moment de la livraison. En effet, il n'est pas rare qu'entre le besoin initial et le besoin réel au moment où le logiciel est opérationnel, plusieurs mois voire plusieurs années après, celui-ci ait évolué, et parfois de manière importante. C'est la réalité du marché. Le logiciel ne répond plus réellement aux attentes même s'il répond aux besoins fonctionnels exprimés. Le développement itératif et la gestion des priorités de Scrum, permet d'obtenir les bonnes fonctionnalités au bon moment et de ne pas gaspiller du temps, de l'énergie et donc de l'argent sur des fonctionnalités qui ne seront, au final, pas utilisées. La productivité, en terme de fonctionnalités développées utiles, est ainsi maximisée.

Que ce passe-t-il si le besoin est plus urgent encore et doit être intégré au sprint en court (dans la mesure où le temps de développement le permet) ? Sur un sprint, les fonctionnalités à implémenter sont également triées et implémentées dans l'ordre des priorités. Cela permet de fournir à la fin d'un sprint les fonctionnalités les plus prioritaires même dans le cas où la totalité n'a pu être réalisé. Il est par conséquent possible, exceptionnellement d'ajouter la nouvelle fonctionnalité au dépend d'une autre moins prioritaire qui ne pourra être réalisée. L'objectif est ici toujours de conserver le niveau de qualité normal.

De cette façon, il sera possible de délivrer fréquemment de nouvelles fonctionnalités aux utilisateurs, parmi celles qui leur apportent le plus de valeur, avec un même niveau de qualité.

9 Conclusion

La qualité a un coût, c'est une évidence. Faire des tests, du « refactoring », concevoir plutôt que développer « tête baissée », pratiquer le pair programming, la relecture de code, etc. consomment du temps, des ressources. Cependant, nous l'avons vu précédemment, le retour sur investissement est positif. Nous parlons donc ici d'investissement, qui doit être anticipé, mutualisé et mis en adéquation avec le besoin. Les projets ne peuvent supporter cet investissement que s'il est compris par la direction et pris en compte dans les budgets initiaux.

Une erreur courante est de ne pas investir pour limiter les délais et le budget du projet et devoir rajouter dix fois le prix par la suite pour répondre aux problèmes de fiabilité, de maintenance, d'évolutivité et de robustesse. Le coût initial d'un projet peut être trompeur et vouloir réduire les coûts à tout prix s'avérer au contraire très cher à moyen terme. La note peut même s'avérer extrêmement salée si l'on prend en compte les coûts induits comme le manque à gagner causé par l'arrêt d'un service de réservation, des dégâts matériels ou pire humains liés à un dysfonctionnement, la perte de compétitivité et d'image en raison d'un logiciel peu fiable ou incapable de s'adapter aux besoins du marché.

Il est pourtant possible de limiter cet investissement, nécessaire, en estimant les besoins en terme de qualité et les risques relatifs à la non qualité. Il sera alors possible d'adapter ses processus, ses pratiques pour cibler principalement les éléments sensibles tout en acceptant une qualité « moindre » mais encadrée sur le reste. Tout le monde n'est pas la NASA et toutes les fonctionnalités ne nécessitent pas la même attention. Cela permet d'adapter l'effort au besoin, éviter ainsi la sur-qualité et maximiser le retour sur investissement.

Les méthodes agiles, dont Scrum, enfin, offrent un cadre à la mise en place pragmatique de la qualité. En effet, la qualité et la satisfaction utilisateur sont au centre des préoccupations du projet mais également la gestion des priorités, l'échange et la communication ainsi que l'amélioration continue. Qualité, productivité et réactivité y sont intimement liés, le qualité étant la clé de voute indispensable aux deux autres sur le long terme. Pouvoir réagir vite sans sacrifier sur la qualité est la principale force de ces méthodes et qui fait leur succès aujourd'hui.

La qualité ne peut être sacrifiée encore longtemps. Gageons que l'avenir de l'informatique sera plus axé sur la qualité qu'il ne l'a été par le passé. Le temps du bricolage à grande échelle qui donne cette image d'improvisation, voir « d'amateurisme » à l'informatique doit laisser la place, à une image plus professionnelle, alliant qualité, réactivité et créativité dans le contexte sans cesse en mouvement qui est le nôtre aujourd'hui.

Xebia
SOFTWARE DEVELOPMENT DONE RIGHT

10 Annexes

1.1 L'auteur

Frédéric DUBOIS

Frédéric a 9 ans d'expérience en architecture, conception et développement d'applications JEE avec une sensibilité particulière pour la qualité logicielle et l'agilité. Plus qu'un attachement aux processus, ou aux connaissances, c'est par une attitude pragmatique, de perfectionnemant continu, d'échange et de partage qu'il contribue, chaque jour, a l'élaboration de solutions pérennes pour ses clients.

Avec la participation de Séven Le Mesle

Architecte Java de 8 ans d'expérience, Séven est un passionné de technologies web tant du côté navigateur que du côté serveur. Convaincu par le libre et l'agilité, il croit fermement dans le partage des connaissances, la synergie du groupe et l'amélioration continue. Avec sa sensibilité toute particulière pour la production et le système, Séven s'intéresse de prêt à l'industrialisation, aux déploiements et à l'exploitation des solutions mises en oeuvre chez ses clients.

1.2 Références

CHAOS Summary 2009, Standish Group, 2009
The Art of Software Testing, G. J. Myers, 1979
Cost/Benefit-Aspects of Software Quality Assurance, Marc Giombetti, 2008
Domain-Driven Design - Tackling Complexity in the Heart of Software, Evans, E., 2004
Lean Software Development : An Agile Toolkit, Mary Poppendieck and Tom Poppendieck, 2003
Manifesto for Agile Software Development, http://agilemanifesto.org
Scrum : Le guide pratique de la méthode agile la plus populaire, Claude Aubry, 2010
CISQ - Consortium for IT Software Quality, www.it-cisq.org

Xebia IT Architects SAS

156 Boulevard Hausmann
75008 PARIS

Tél : 01 53 89 99 99
Mail : info@xebia.fr

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

CC BY NC ND