UNIVERSITÉ DU QUÉBEC
À MONTRÉAL

PROJET DE SYNTHÈSE EN GÉNIE LOGICIEL
PRÉSENTÉ À
UNIVERSITÉ DU QUÉBEC À MONTRÉAL

COMME EXIGENCE PARTIELLE
À L'OBTENTION DE LA
MAÎTRISE EN GÉNIE LOGICIEL

PAR
Martin BÉRUBÉ

MISE EN ŒUVRE DES REVUES PAR LES PAIRS DANS UNE ORGANISATION DE DÉVELOPPEMENT LOGICIEL

MONTRÉAL, 13 OCTOBRE 2017

PRÉSENTATION DU JURY

CE RAPPORT DE PROJET A ÉTÉ ÉVALUÉ

PAR UN JURY COMPOSÉ DE :

Mme Sylvie Trudel, directrice de projet
Département d'informatique à l'Université du Québec à Montréal

Monsieur Louis Martin, directeur du programme de maîtrise en génie logiciel
Département d'informatique à l'Université du Québec à Montréal

Monsieur Normand Séguin, vice-doyen aux études
Département d'informatique à l'Université du Québec à Montréal

IL A FAIT L'OBJET D'UNE SOUTENANCE DEVANT JURY ET PUBLIC

LE 12 OCTOBRE 2017

À L'UNIVERSITÉ DU QUÉBEC À MONTRÉAL

REMERCIEMENTS

Un travail de longue haleine comme celui-ci est impossible sans le soutien indéfectible d'une multitude de personnes.

Tout d'abord, un immense merci à ma directrice de projet Sylvie Trudel, autant pour tes connaissances, tes compétences, ton éthique de travail que tes aptitudes pédagogiques et ton énergie. Nos rencontres m'ont enrichi de bien des façons et ce que j'ai appris durant ces cinq ans à te côtoyer dépasse largement le cadre du génie logiciel. Continue de motiver tes élèves comme tu sais si bien le faire. Ce fut un privilège de travailler avec toi. Merci encore Sylvie.

Je tiens à remercier également la direction du Centre d'information Rx ltée, en particulier Alain Thiffault, Stéphane Morin et Mathieu Courchesne pour leur appui dans le cadre de ce projet. Merci aussi aux membres de l'équipe Web-Micro pour votre collaboration, votre travail, vos commentaires et vos suggestions durant toutes les étapes de réalisation. Un merci tout spécial à Alain Boudreault pour ta confiance et tes encouragements à me lancer dans l'aventure qu'a été cette Maîtrise en génie logiciel. Tu y croyais plus que moi et tu as eu raison d'insister. Merci Alain.

Finalement, un merci tout spécial à ma famille : Évelyne, Samuel et Alice, vous avez dû composer avec un conjoint et un père trop absent durant les dernières années, mais votre aide, votre patience et votre compréhension m'ont aidé à persévérer et à passer au travers des moments de fatigue et de découragement. Vous étiez aussi là pour partager les joies, les succès et la fierté du travail bien fait. Je n'y serais pas arrivé sans vous. Merci pour tout.

MISE EN ŒUVRE DE REVUES PAR LES PAIRS DANS UNE ORGANISATION DE DÉVELOPPEMENT LOGICIEL

Martin BÉRUBÉ

RÉSUMÉ

Les entreprises dans le commerce de détail sont dans une compétition féroce où l'innovation, les délais de plus en plus courts de mise en marché et l'expérience client sont des éléments clés pour se démarquer. En tant que fournisseur de solutions technologiques à une entreprise de ce secteur, le Centre d'information Rx ltée (CIRX) est toujours en quête d'améliorations lui permettant d'offrir à ses clients des logiciels de la plus haute qualité à coût compétitif.

Ce rapport examine une initiative du CIRX pour mettre en place des revues par les pairs et ainsi atteindre plusieurs objectifs comme l'amélioration de la qualité des logiciels qu'elle produit, l'augmentation de sa productivité, le transfert de connaissance et de compétence entre les membres d'une équipe et une meilleure uniformité du processus de développement. Durant ce projet, les aspects autant techniques qu'humains de la revue par les pairs ont été abordés, documentés et mesurés. Compte tenu de la courte durée du projet (six mois), le processus de revue s'est concentré sur le code source des logiciels produits développés par l'équipe Web-Micro du CIRX.

Les résultats présentent les mesures de 11 projets réalisés et révisés entre janvier et juillet 2017. Ils permettent de faire les constats suivants :

- La quantification de certains aspects du processus de développement (comme le temps de *rework*) représente un défi important et demande une rigueur de tous les participants.
- Une documentation rigoureuse et constante des défauts découverts en tests et en production est essentielle pour mesurer de façon précise l'impact des revues sur la qualité des logiciels.
- Les prises de mesures ont permis de déceler des écarts dans des processus existants comme la documentation des défauts et la mesure de la taille fonctionnelle des projets.
- Le coût relatif par point de fonction d'un projet semble fortement corrélé au nombre de défauts détectés, confirmant l'hypothèse qu'une réduction des défauts à la source améliore la productivité. Toutefois, les résultats obtenus ne permettent pas de conclure que la revue a une influence significative sur le nombre de défauts détectés.
- Malgré tout, un sondage réalisé en fin de projet a confirmé que, malgré le manque de résultats quantitatif, les membres de l'équipe ont appris beaucoup de l'expérience, la considère comme positive et désirent poursuivre l'amélioration continue du processus de revue.

INTRODUCING PEER REVIEWS IN A SOFTWARE DEVELOPMENT ORGANIZATION

Martin BÉRUBÉ

ABSTRACT

Companies in the retail business are in a fierce competition where innovation, shorter time to market and user experience are keys to stand out. As a technology solutions supplier for a retail company, Centre d'information Rx ltée (CIRX) always tries to innovate and improve its processes to offer high quality software at competitive cost to its clients.

This report presents an initiative of CIRX to implement peer reviews and reach various targets such as better software quality, increased productivity, knowledge transfer between team members, and better consistency in applying the development process. During this project, technical and human aspects of peer reviews have been covered, documented, and measured. Considering the shortness of the project (six months), the peer review process has focused on the source code developed by the Web-Micro team of CIRX.

Results present metrics from 11 projects conducted and reviewed from January to July 2017. The data leads to the following observations :

- Quantifying some aspects of the development process (like rework effort) is a significant challenge and requires rigor from all participants.
- A proper and constant documentation of defects found in tests and production is essential to be able to measure precisely the impact of peer review on software quality.
- The measurement process allowed us to find discrepancies in existing processes like defect documentation and functional size measurement of projects.
- The relative cost per function point of projects seems strongly correlated to the number of defects found, confirming the hypothesis that the reduction of defects in the development process increases productivity. However, the results cannot allow us to conclude that reviews have a significant influence on the number of defects found.
- Nevertheless, a survey at the end of the project confirmed that, regardless of few quantitative results, team members have learned a lot from the experience, they described it as positive and they want to continue improving the peer review process.

TABLE DES MATIÈRES

Page

LISTE DES TABLEAUX

LISTE DES FIGURES

Page

LISTE DES ABRÉVIATIONS, SIGLES ET ACRONYMES

CIRX	Centre d'information Rx ltée
CMMI	Capability Maturity Model Integration/Modèle intégré d'évolution des capacités
Commit	Terme anglais désignant l'enregistrement d'un ensemble de changements dans les fichiers composants une application.
CRIM	Centre de Recherche Informatique de Montréal
IDE	Integrated Development Environment/Environnement de développement intégré
ISO/IEC	International Organization for Standardization / International Electro-technical Commission
GJC	Groupe Jean Coutu
KSLOC	Un millier de lignes de code source (Kilo Source Lines of Code)
Merge	Terme anglais désignant l'action de fusionner les commits de deux branches de code dans le gestionnaire de code source
Métho	Nom commun donné à la méthodologie de développement du CIRX
PCI DSS	The Payment Card Industry Data Security Standard
Poker planning / Planning poker	Terme anglais désignant une méthode d'estimation d'effort collective utilisant un jeu de cartes pour alimenter les discussions sur l'effort requis pour effectuer certaines tâches. Voir la page suivante pour plus de détails sur la méthode : https://en.wikipedia.org/wiki/Planning_poker
PP	Problèmes Potentiels
Pull request	Terme anglais désignant l'action de demander à un autre développeur la permission de fusionner une branche source du code source dans une branche de destination. L'autorisation sera généralement donnée ou refusée après la revue des changements effectués dans la branche source.
QA	Assurance qualité (Quality assurance)
Refactoring	Terme anglais désignant l'action de modifier la structure interne du code dans le but de l'améliorer ou de la simplifier tout en conservant le comportement existant.
Rework	Terme anglais désignant l'effort nécessaire pour identifier, documenter, corriger et tester un défaut non détecté par une revue.

SOX	Sarbanes-Oxley
SOW	Statement Of Work/Analyse préliminaire
UAT	Tests d'acceptation utilisateur (User Acceptance Tests)
VER	Vérification
XP	eXtreme Programming

INTRODUCTION

Ce rapport est le résultat d'un stage de maîtrise en génie logiciel réalisé en entreprise. La direction de l'entreprise en question, Centre d'Information Rx Ltée, a mandaté l'auteur de ce rapport pour la mise en place de revues par les pairs dans son processus de développement. Elle utilise déjà une méthodologie formelle pour la réalisation de ses projets de développement logiciel et cette méthodologie inclut une étape de révision par les pairs. Toutefois, cette étape du processus n'a jamais été étudiée et documentée. Certaines équipes mettent en place des revues systématiques, d'autres non. De plus, la façon de réaliser ces revues est laissée à la discrétion de chacun ce qui cause des disparités importantes dans la qualité des artéfacts logiciels produits. Le but de cette initiative est d'expérimenter la mise en place d'un processus de revue formel et systématique dans une des équipes de développement pour ensuite profiter des données recueillies pour étendre la revue par les pairs aux autres équipes de développement.

Ce rapport est structuré comme suit :

- Le chapitre 1 traite du contexte de l'entreprise et de son processus actuel de développement.
- Le chapitre 2 présente le cadre théorique c'est-à-dire une revue de la littérature sur l'état des pratiques de revues.
- Le chapitre 3 présente la méthodologie utilisée pour la mise en place du processus de revue, la réalisation de l'initiative et le suivi de celle-ci.
- Le chapitre 4 décrit la méthodologie de développement adopté pour l'initiative.
- Le chapitre 5 présente les résultats obtenus.
- Le chapitre 6 revient sur certains évènements rencontrés en cours de projet qui mérite discussion.
- Le chapitre 7 contient un ensemble de recommandations pour l'entreprise afin de poursuivre l'initiative et de l'étendre aux autres équipes de développement.

CHAPITRE 1

CONTEXTE ET PROBLÉMATIQUE

1.1 Centre d'information Rx ltée

Le Centre d'information Rx ltée (CIRX) est une entreprise de technologie située à Varennes sur la Rive-Sud de Montréal. Elle a été fondée le 11 août 1972 et emploie environ 160 personnes. Le CIRX est une filiale du Groupe Jean Coutu (GJC) et offre ses services technologiques aux différentes filiales du GJC et aux succursales du réseau. Le CIRX offre plusieurs services qui sont fournis par l'une de ses deux divisions soit les services techniques – qui incluent le centre d'assistance, les services d'infrastructure et le support technique – et le développement logiciel. Les divisions sont composées de plusieurs départements, chacun étant géré par un ou deux directeurs. La Figure 1.1 donne un aperçu de l'organigramme de l'entreprise. Dans le cadre de la mise en œuvre de la revue par les pairs, l'accent a été mis sur les équipes de développement, principalement l'équipe Web-Micro.

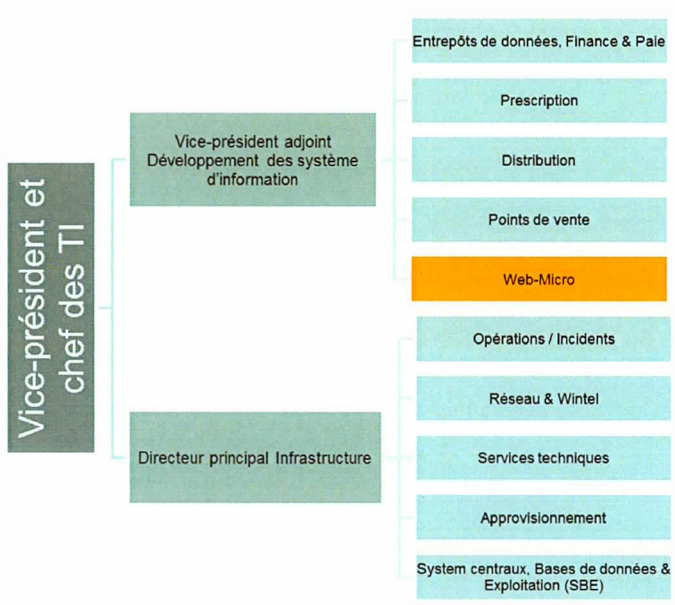


Figure 1.1 Organigramme de CIRX

1.2 L'équipe Web-Micro

L'équipe Web-Micro est composée de deux directeurs, un architecte technique, cinq analystes, onze développeurs et deux intégrateurs web. L'équipe Web-Micro s'occupe de développer, maintenir et faire évoluer un portfolio d'applications web et client riche déployées autant au siège social du GJC que dans les succursales. Elle s'occupe également des applications destinées aux consommateurs comme le site web grand public de l'entreprise et les applications sur les plateformes mobiles comme iOS et Android. Elle se spécialise dans les technologies .Net de Microsoft, mais maintient également des applications dans d'autres technologies telles Delphi et Lotus Notes. L'équipe est expérimentée et le taux de roulement du personnel est très bas. La moyenne d'ancienneté des membres de l'équipe est de près de 10 ans.

Depuis quelques années, l'équipe Web-Micro est impliquée dans un nombre croissant de projets. Ayant été pendant longtemps la seule équipe de l'entreprise à utiliser et maitriser les technologies .Net de Microsoft, l'équipe est responsable du maintien et du support d'un port-

folio d'environ 90 applications et de nouvelles applications sont développées chaque année. La plupart des applications du portfolio sont de taille petite à moyenne ce qui fait que la plupart des projets de développement d'application n'impliquent qu'un seul analyste et un seul développeur. Comme la demande ne cesse d'augmenter, l'équipe doit régulièrement faire appel à des contractuels ou à des membres d'autres équipes de l'entreprise qui ont une connaissance du .Net (ou une facilité à l'apprendre) pour répondre à des besoins ponctuels. Comme de plus en plus de personnes au CIRX utilisent les technologies Microsoft, ce genre de transfert de personnel entre les équipes est appelé à se multiplier dans le futur.

1.3 La méthodologie de développement

Au milieu des années 2000, le CIRX a mis en place une méthodologie de développement dans le but de standardiser le processus de développement logiciel qui était très différent d'une équipe à l'autre. Cette méthodologie (ci-après nommée la Métho) définit un certain nombre d'étapes à réaliser, de livrables à produire et d'autorisations à obtenir, adaptés à la taille du projet. Les étapes ont été approuvées par la direction et les services juridiques pour s'assurer que l'entreprise a en place toutes les procédures nécessaires pour respecter des réglementations telles que Sarbanes-Oxley (SOX) et PCI DSS. L'approche est en cascade donc les premiers livrables comprennent les documents d'analyse préliminaire (appelés *Statement Of Work* ou SOW), de spécification des exigences et les analyses fonctionnelles. Quelques initiatives agiles ont été tentées dans les dernières années, mais il n'y a pas d'utilisation d'un processus itératif à grande échelle dans l'entreprise jusqu'à maintenant. L'équipe responsable des processus s'assure que les équipes produisent les livrables requis selon le type, la taille et l'avancement du projet, mais est assez flexible sur le format et la séquence des livrables à produire. Les équipes sont encouragées à adapter les livrables à leur contexte technologique si cela leur permet d'améliorer leur productivité. Par exemple, la Métho exige de documenter les résultats des tests unitaires. Ce document est basé sur un modèle de document Word, mais a été remplacé par certaines équipes par un rapport d'exécution des tests unitaires automatisés. Les informations incluses dans le rapport sont similaires à celles du rapport papier ma-

nuel donc respectent les exigences de la Métho. La Métho intègre également des standards dans les méthodes de travail des équipes comme le calcul de la taille fonctionnelle des applications avec la méthode COSMIC pour les estimations des projets et l'utilisation de Microsoft Sharepoint pour le suivi et l'archivage des documents.

1.4 Adaptation de la méthodologie par l'équipe Web-Micro

Lors de la mise en place de la Métho, la plupart des projets de développement se faisaient encore sur les systèmes centraux (*mainframe*) pour lesquels les outils de développement sont assez limités (en nombre et en fonctionnalités). Les documents de la Métho permettaient de remédier au manque d'informations et de fonctionnalités disponibles dans ces outils.

Dans un souci constant d'améliorer sa productivité et la qualité de ses logiciels, l'équipe Web-Micro a mis en place depuis environ une dizaine d'années un ensemble de pratiques de développement modernes : utilisation systématique d'un gestionnaire de code source, tests unitaires automatisés, systèmes de traçabilité des bogues et des items de travail, intégration continue, déploiement automatisé et suivi des déploiements dans les différents environnements. Toutes ces initiatives ont permis à l'équipe de maintenir une bonne productivité si on la compare aux moyennes compilées dans l'industrie malgré une charge de travail qui croît sans cesse depuis quelques années. Lorsqu'est venu le temps de mettre en place la Métho dans l'équipe, il était évident que les outils de développement modernes et l'automatisation permettaient de remplacer plusieurs documents de la Métho par des rapports produits automatiquement ou sur demande (ex : rapports de tests unitaires automatisés) ou par des artéfacts générés par les outils comme l'historique des changements dans le code source et les outils de déploiement automatisé. L'équipe est toujours à la recherche d'optimisation possible qui lui permettrait de respecter toutes les exigences de la Métho en réduisant la charge de travail sur les développeurs et les analystes.

1.5 Problématique

La Métho inclut des étapes pour la revue des analyses fonctionnelles, des spécifications détaillées et du code par un pair. Étant donné que les technologies diffèrent grandement dans l'entreprise, peu de lignes directrices ont été données sur la façon de réaliser la revue et sur les éléments à inclure dans les revues. Les formulaires de revue prévus par la Métho sont très sommaires et doivent donc être bonifiés par les équipes. Le manque de précision sur la forme et le contenu des revues provoque de grandes variabilités dans les revues, dépendant de la personne assignée à la réaliser. Il n'y a pas non plus de recommandations sur le bon moment et la fréquence des revues. Étant donné le modèle en cascade, la Métho place la revue de code à la fin du développement. Or, il est arrivé fréquemment dans le passé que des problèmes soient soulevés lors d'une revue, mais que ceux-ci ne soient pas adressés parce que l'impact des changements était trop important à cette étape [avancée] du projet alors qu'une revue plus tôt dans le projet aurait permis une amélioration du produit (ou de l'architecture du code) avant que le coût du changement n'outrepasse les bénéfices de celui-ci.

La direction a donc décidé de mandater l'architecte de l'équipe Web-Micro pour revoir la façon de faire des revues par les pairs et proposer des façons de faire qui pourront être applicables à l'équipe, mais aussi potentiellement aux autres équipes de développement. En mettant en place cette initiative, la direction cherche à résoudre plusieurs problèmes.

Elle souhaite d'abord améliorer la productivité des développeurs. Comme mentionné précédemment, l'équipe est plutôt petite par rapport aux nombres d'applications dans son portfolio. Les développeurs peuvent donc travailler sur un grand nombre d'applications dans une même année et il peut aussi se passer plusieurs années avant qu'un développeur ne retravaille dans une application qu'il connait déjà. Après une durée de temps si longue, il est peu probable que le développeur se souvienne en détail du fonctionnement et de l'architecture de l'application et doit pratiquement reprendre son apprentissage de zéro à chaque fois. Aussi, comme l'équipe a de plus en plus tendance à faire appel à des contractuels ou des employés temporaires provenant d'autres équipes, la standardisation des façons de faire est de plus en

plus importante, autant pour que le nouvel employé puisse être productif rapidement que pour l'équipe Web-Micro qui devra par la suite déployer, maintenir et supporter le code produit par cet employé qui n'est pas encore habitué aux façons de faire de l'équipe.

Un des bénéfices reconnus de la revue est l'amélioration générale de la qualité du logiciel (CMMI® Product Team 2010). Bien que la qualité des applications soit plutôt bonne du point de vue des utilisateurs finaux, beaucoup de tests sont effectués par l'équipe pour s'assurer d'offrir un produit de qualité. Toutes ces vérifications demandent cependant beaucoup de temps et si la mise en place systématique des revues permettait de réduire la durée des tests et l'effort de correction, cela pourrait être un apport important à la productivité de l'équipe.

1.6 Approche proposée

Nous proposons de déployer un cadre de revue par les pairs dans l'équipe Web-Micro et d'étudier comment ce cadre pourrait s'adapter aux autres équipes de CIRX. La mise en place de ce cadre devra tenir compte des réalités de l'équipe c'est-à-dire la quantité et la nature des projets réalisés par celle-ci. Pour ce faire, différentes techniques de revue populaires seront étudiées et évaluées pour déterminer celles qui s'adapteraient le mieux au fonctionnement de l'équipe et aux objectifs à rencontrer dans le cadre de l'initiative. En effet, le cadre choisi devra tenir compte des méthodes de travail actuelles et s'y adapter avec un minimum de perturbations. Les besoins en acquisition matérielle, logicielle ainsi que les besoins en formation devront également être pris en compte.

Le responsable de l'initiative, l'auteur de ce document, est aussi architecte dans l'équipe Web-Micro et devra prendre un ensemble de décisions durant toute l'initiative. Cependant, l'approche pour toutes les étapes de la réalisation sera basée sur la consultation des différents participants au processus de revue afin de s'assurer que les suppositions faites et les décisions prises par l'architecte et la direction correspondent bien à la réalité du travail réalisé par les développeurs et les analystes. Il est important de profiter de leur expérience et de prendre en

considération leurs commentaires. L'approche se veut également itérative c'est-à-dire qu'à différents moments de l'initiative, les participants et les données recueillies seront consultés pour ajuster le processus et ainsi le rendre le plus efficace possible et favoriser son adoption.

Finalement, les participants et la direction seront tenus informés à tout moment de l'avancement des travaux et ils seront encouragés à participer activement au succès de l'initiative en proposant des améliorations.

CHAPITRE 2

CADRE THÉORIQUE

L'inspection des produits logiciels est un domaine de processus qui a été largement étudié et documenté. Elle est décrite comme la pratique de vérification (VER) dans le CMMI (*Capability Maturity Model Integration* ou Modèle intégré d'évolution des capacités) (CMMI® Product Team 2010) et fait partie des pratiques de maturité de niveau trois. La vérification englobe toutes les activités qui permettent de s'assurer que le logiciel et les produits intermédiaires nécessaires à sa création sont conformes aux normes de qualité établies par l'organisation et aux spécifications. Ces activités incluent, entre autres choses, les tests, les simulations, les démonstrations au client et bien sûr, les revues par les pairs. Les revues par les pairs peuvent être effectuées à différents moments dans le cycle de vie du logiciel et peuvent également être réalisées avec un degré plus ou moins grand de formalisme. Les différents types de revues et leurs caractéristiques seront décrits en détail plus tard dans le chapitre. Pour être efficaces, ces revues doivent être planifiées dès le départ, car le temps et les ressources nécessaires doivent faire partie du plan de projet pour éviter les surprises et réduire les chances de voir les processus de revue escamotés en fin de projet lorsque l'échéancier ou le budget deviennent plus serrés.

2.1 Historique

Micheal Fagan fut l'un des premiers à expérimenter et documenter des procédures formelles sur la façon de réaliser des revues chez IBM dans les années 1970 (Wiegers 2002) et il a ainsi jeté les bases de l'inspection des produits logiciels. Bien qu'au départ cette technique servait principalement à détecter les erreurs dans le code une fois celui-ci complété, il réalisa rapidement qu'une révision de plusieurs artéfacts du processus de développement logiciel comme les documents de spécifications et les analyses fonctionnelles tôt dans le projet permettait de réduire en amont les erreurs et les omissions et ainsi réduire le nombre de défauts

injectés dans le logiciel durant le développement. Il a été clairement démontré qu'un défaut détecté tôt dans le processus (par exemple, lors de la révision des spécifications) coûte jusqu'à 40 fois moins cher à corriger que durant le développement et jusqu'à 1000 fois moins cher que s'il est découvert une fois le logiciel en production. Bien que les valeurs relatives puissent varier d'une étude à l'autre, il a été prouvé à de nombreuses reprises par des études et des mesures prises dans les entreprises que plus un défaut est trouvé tôt dans le processus et corrigé rapidement, plus l'effort et le coût de correction sont bas. Ou, pour paraphraser Jason Cohen (Cohen, Teleki et al. 2013) : « find bugs early and often ».

Par la suite, les travaux de Fagan ont été repris par plusieurs autres auteurs qui ont voulu vérifier et confirmer ces affirmations. Tom Gilb et Dorothy Graham ont notamment réuni dans leur livre *Software Inspection* (Gilb and Graham 1993) les résultats de leurs recherches et de leurs expériences personnelles. Dans cet ouvrage, les auteurs décrivent de façon détaillée les étapes nécessaires à l'inspection des produits logiciels et les rôles de chaque intervenant dans le processus. Par exemple, ils insistent sur le fait que les inspections sont coordonnées par un expert (« Inspection Leader ») dument formé pour mener efficacement les revues. Les auteurs donnent également un ensemble de conseils pratiques (par exemple, des modèles de documents et de formulaires) pour rendre les revues plus efficaces. Ils ont également consacré une bonne partie du livre aux étapes nécessaires à la mise en place d'un processus d'inspection dans l'entreprise, incluant les difficultés les plus souvent rencontrées et des stratégies pour atténuer celles-ci.

Bien que les travaux de Fagan, Gilb et Graham démontrent les bénéfices des inspections des produits logiciels, c'est un processus qui demeure très lourd et qui demande des ressources importantes. Ce modèle de revues, principalement mis en place dans les entreprises de grande et très grande taille, n'est pas facilement applicable aux petites et moyennes entreprises. De plus, l'arrivée de l'approche Agile (Beck, Beedle et al. 2001) et des méthodes de développement à itérations courtes comme Scrum (Schwaber and Sutherland 2016) et eXtreme Programming (XP) (Beck and Andres 2004) au tournant des années 2000 nécessitaient une révision des méthodes d'inspection afin de vérifier si elles étaient toujours pertinentes et

efficaces dans ce contexte. Karl Wiegers (Wiegers 2002) s'est attardé à inventorier les différentes techniques de revues par les pairs et à les comparer entre elles afin de déterminer lesquelles sont les plus appropriées et dans quel contexte. Il a réitéré que l'inspection formelle permet généralement de trouver le plus grand nombre de défauts dans les produits logiciels et que les équipes devraient au moins expérimenter l'inspection formelle pour déterminer si cette façon de faire apporte les bénéfices escomptés, mais que dans un contexte où les produits sont moins complexes ou critiques, des techniques d'inspection plus « légères » peuvent apporter des bénéfices similaires à un coût moindre. Une de ces techniques moins formelles est la programmation en binôme (« pair programming »). Elle a été popularisée par Kent Beck dans sa méthodologie de développement nommé eXtreme Programming (XP). En ayant deux développeurs qui travaillent ensemble sur le même terminal, une revue informelle et continue est faite. Toutefois, les opinions divergent à savoir si cette forme de revue remplace ou non les inspections plus formelles. Selon Wiegers, bien que des expérimentations aient démontré que la programmation en binôme améliore la qualité du code (Williams and Kessler 2000), elle ne peut remplacer l'inspection.

Étant donné que peu d'études ont comparé l'efficacité de la revue par les pairs et la programmation en binôme, Matthias M. Müller (Müller 2005) de l'université Karlsruhe en Allemagne a voulu vérifier ce qu'il en est et a tenté l'expérience avec des étudiants en 2002 et 2003. Il a en effet constaté que la plupart des études sur l'efficacité de la programmation en binôme vérifiaient deux variables en même temps soit le temps nécessaire pour réaliser la tâche et un aspect de la qualité de celle-ci, soit le nombre de défauts trouvés. Généralement, les conclusions démontraient que la programmation en binôme produisait des logiciels de meilleure qualité, mais qu'il fallait plus de temps pour y arriver (puisque le temps de programmation est multiplié par deux). Müller a voulu éliminer la variabilité de la qualité et déterminer le temps nécessaire pour une équipe de deux développeurs (en binôme) à réaliser une tâche avec un niveau de qualité prédéterminé (95 % des tests d'acceptation devaient réussir). Il a par la suite donné la même tâche à un participant seul, mais qui pouvait tirer profit d'une revue fait par un autre développeur. Celui-ci devait identifier les défauts dans le code et l'auteur avait l'obligation de les corriger. L'étude est très détaillée sur la méthodologie

utilisée pour s'assurer d'obtenir des résultats qui pourront être comparés adéquatement entre les deux groupes. Elle arrive à la conclusion que les deux groupes réussissaient à obtenir un niveau de qualité similaire, avec un léger avantage pour les équipes en binôme. Le coût est cependant plus élevé en moyenne pour les équipes en binôme, mais l'écart de coût est plutôt faible (environ 13 %) et cet écart tend à diminuer lorsque le niveau de qualité demandé augmente, car les revues doivent être plus fréquentes et plus approfondies.

Plus récemment en 2013, l'entreprise SmartBear software, par les auteurs Jason Cohen, Steven Teleki, Eric Brown et leurs collaborateurs (Cohen, Teleki et al. 2013), ont voulu remettre à jour les connaissances sur la revue et voir comment les dernières études et les expériences récentes confirmaient ou infirmaient les méthodes préconisées par Fagan, Gilb et Wiegers. Comme eux, les auteurs insistent sur la valeur ajoutée des revues, non seulement pour améliorer la qualité des logiciels pour les utilisateurs, mais aussi comme avantage compétitif, particulièrement dans le logiciel grand public. Un logiciel sans défaut majeur permet de prendre ou de conserver des parts de marché alors qu'une version mal ficelée d'un logiciel très populaire (l'exemple donné est Acrobat Reader d'Adobe) peut avoir des effets catastrophiques sur la fidélité des utilisateurs et, par conséquent, sur la capacité de survivre de l'entreprise. Les auteurs n'hésitent pas à adresser les difficultés à mettre en place les revues par exemple l'égo des développeurs et la perception de lourdeur qui accompagne les rencontres d'inspection (préparer et lire les documents, planifier la rencontre, le temps passé à la rencontre elle-même et la documentation des défauts). Ils expliquent également comment les outils de travail modernes qui permettent d'identifier rapidement les changements dans le code et de les vérifier ont permis la mise en place de processus de revue plus léger, ne nécessitant pas de réunir tout le monde dans la même pièce, avec une efficacité pratiquement similaire selon l'étude réalisée par Votta (Votta 1993), corroboré plus tard par une autre étude de Diane Kelly et Terry Shepard citée dans l'analyse de SmartBear. De plus, les équipes de développement sont de plus en plus décentralisées avec tous les problèmes que cette situation peut apporter comme la difficulté à organiser des rencontres, les différences de langues et de fuseau horaire. Un autre élément à prendre en compte est le changement dans les méthodes de développement. Les travaux de Fagan et Gilb ont été réalisés à une époque où le dévelop-

pement se faisait principalement de façon procédurale et un réviseur pouvait relativement facilement analyser le fonctionnement du logiciel à l'aide d'une impression du code source. Avec la programmation orientée objet (OOP), il est difficile pour un réviseur externe de comprendre le fonctionnement du code qui est morcelé dans un grand nombre de classes interagissant entre elles sans obtenir d'informations préalables sur ces différentes classes et l'importance de chacune. Il devient essentiel de trouver des solutions de rechange viables aux inspections « classiques ».

Sans recommander une méthode de revue précise, les différentes études que les auteurs ont analysées donnent quelques pistes intéressantes :

- Une lecture individuelle du code est pratiquement aussi efficace qu'une rencontre de revue et permet aux participants de la réaliser au moment qui leur convient le mieux.
- Il faut limiter la durée à environ 60 minutes, maximum 90 minutes. La limite de temps est importante, car les expériences réalisées ont démontré que le taux de détection diminue rapidement après une heure de concentration intense et que très peu de défauts sont trouvés au-delà de 90 minutes (voir figure 2.1).
- Le plus grand indicateur du nombre de défauts trouvés est la vitesse de lecture. Aussi évident que ça puisse paraitre, c'est un élément important. Si vous voulez trouver plus de défauts, révisez plus lentement.
- Il est à peu près impossible de définir des ratios normalisés de défauts par unité de taille (ex. : 12 défauts par millier de lignes de code [KSLOC]). Il y a trop de différences entre les plateformes, les projets, les langages de programmation, les méthodes de travail des équipes pour établir un ratio qui s'appliquerait à l'industrie en entier. Il peut être possible d'en établir pour le contexte spécifique d'une équipe pour un type de projet précis, mais même cela s'avère tout de même difficile.
- Les listes de vérification sont très importantes. Elles permettent non seulement de détecter les erreurs potentielles dans le code, mais également d'identifier certaines omissions, plus difficiles à révéler.

— La plupart des études sur le sujet ont un petit échantillon. Il faut donc s'assurer de chercher les tendances qui reviennent dans les différentes études, car individuellement, aucune d'entre elles ne permet de tirer des conclusions.

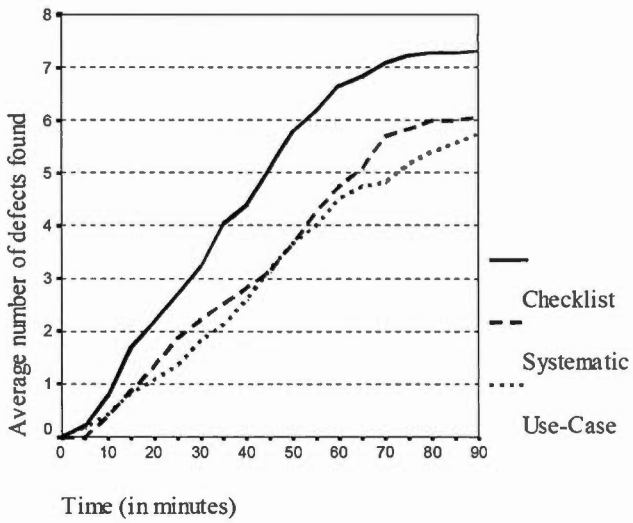


Figure 2.1 Taux de détection des défauts dans le temps (Cohen, Teleki et al. 2013)

Les techniques de revue ont également dû être adaptées à la réalité de projets en code source libre (Open Source) auxquels contribuent des milliers de personnes. Par exemple, le projet Chromium de Google utilise un processus de revue systématique des changements. (Google 2017) L'utilisateur qui soumet un correctif ou une nouvelle fonctionnalité doit obligatoirement identifier des responsables pour la revue. Normalement, elle sera effectuée par plus d'une personne et elles doivent avoir une connaissance poussée du module modifié. Tant que ces personnes n'auront pas approuvé le changement, le code ne sera pas introduit dans le code source public et donc, dans une version future du logiciel. Comme l'équipe de Chromium veut garder un haut niveau de qualité, il est excessivement rare qu'une correction soit approuvée telle quelle. Au dire de développeurs impliqués dans le projet, cela demande une bonne dose d'humilité, mais les gens comprennent vite que le but n'est pas d'humilier qui que ce soit, mais bien d'assurer un niveau de qualité élevé, nécessaire à un projet auquel travaillent des milliers de personnes. Comme les contributeurs sont répartis partout dans le monde, tout le processus se fait à distance et de façon désynchronisée. Le réviseur va vérifier

les changements dans le code, écrire ses commentaires pour l'auteur qui devra alors en tenir compte et apporter les correctifs nécessaires. Jamais ils n'auront été en contact direct durant toutes les étapes de la revue. On est très loin du modèle de Fagan où tous les intervenants se trouvent en même temps dans la même pièce!

2.2 Les types de revues

En fonction du niveau de complexité et la criticité du produit à réviser, différentes techniques de revue avec différents niveaux de formalisme sont possibles. Dans son ouvrage Peer Reviews in software (Wiegers 2002), Wiegers décrit en détail sept types de revues, allant du très structuré au très informel. Les auteurs de SmartBear ont également détaillé quelques types de revues qui reprennent pour l'essentiel celles de Wiegers, mais en détaillant certaines variations lorsque l'équipe utilise des outils modernes comme les courriels ou des outils de revue spécialisés. Pour cette raison, ils ne seront pas repris ici. Toutefois, une autre méthode de revue sera décrite soit celle créée par le Centre de Recherche Informatique de Montréal (CRIM) pour répondre aux besoins de revues des entreprises du Québec.

2.2.1 L'inspection

C'est le type de revue le plus formel et le plus détaillé. C'est la technique décrite par Fagan et Gilb et toutes les activités relatives à l'inspection sont très prescriptives. Il y a des étapes précises à respecter, des rôles définis avec des tâches claires pour chacun et des recommandations sur la vitesse des inspections.

Les rôles définis pour l'inspection sont :

- l'auteur : celui qui a fait les changements à réviser;
- le modérateur ou *leader* d'inspection : celui qui s'assure du bon déroulement de toutes les étapes de la revue;
- le lecteur : qui décrit, à haute voix et dans ses mots, la partie de document à réviser;
- l'enregistreur ou le scribe : qui documente les changements à faire par l'auteur;
- les inspecteurs : qui vérifient le travail de l'auteur;

— le vérificateur : qui s'assure que les corrections demandées sur le document soient bien réalisées par l'auteur par la suite.

Il est important de noter que les rôles de lecteur, d'enregistreur et de vérificateur sont assignés à des inspecteurs déjà présents lors de la revue. Il ne s'agit pas de personnes supplémentaires à inclure dans le processus de revue.

Il est recommandé d'avoir au minimum deux inspecteurs en plus de l'auteur pour la révision. Le nombre de six inspecteurs est le maximum suggéré par Wiegers qui fait état d'études démontrant que le fait d'augmenter le nombre de participants n'améliore pas nécessairement le nombre de défauts trouvés. En fait, certaines études ont même démontré que deux équipes inspectant le même document trouvent rarement les mêmes défauts et que plusieurs inspections avec un petit nombre de personnes seraient préférables à une seule revue incluant plusieurs personnes. Par contre, d'autres études ont contredit cette affirmation donc il est important de valider les deux hypothèses dans le contexte de l'entreprise pour voir laquelle s'applique.

Les étapes de l'inspection sont :

— La planification : à cette étape, on assigne un modérateur à la revue (qui n'est pas l'auteur du document à réviser). Le modérateur et l'auteur vont étudier le document à réviser et déterminer les personnes qui devraient inspecter le document, compte tenu de leurs champs d'expertise. Ils vont également estimer, basé sur les historiques d'inspections précédentes, le temps nécessaire pour réviser le document, dépendant de la taille de la section visée par la revue.

— Rencontre de démarrage: il s'agit d'une courte rencontre informelle où seront invités les inspecteurs. Les documents à réviser leur seront remis et ils seront informés des éléments pertinents comme les sections à réviser, le contexte du document et les éléments connexes à connaitre pour bien comprendre le document. La rencontre peut être remplacée par une note, remise avec le document, expliquant les mêmes éléments.

— La préparation : durant cette étape, les inspecteurs vont parcourir attentivement et individuellement le document pour y détecter des défauts. Un outil intéressant pour

orienter le travail des inspecteurs est une liste de vérification des défauts les plus communs ou les plus critiques. Chaque inspecteur prend note des défauts trouvés en préparation pour la rencontre de revue.

- La rencontre de revue : c'est le cœur du processus d'inspection. Le lecteur passe au travers des sections à inspecter et pour chacune d'entre elles, les inspecteurs partagent avec les autres les défauts qu'ils ont trouvés. Les défauts peuvent être discutés entre les inspecteurs et avec l'auteur afin de déterminer s'il s'agit vraiment de défaut et si c'est le cas, l'enregistreur prend en note le défaut identifié qui devra alors être corrigé par l'auteur.
- La correction : ayant reçu la liste des corrections à faire, l'auteur apporte les changements nécessaires à son artéfact. Si la correction ne peut pas être faite immédiatement, le défaut devra être consigné dans une base de données pour ne pas perdre la trace du travail à faire et y revenir dans le futur.
- Le suivi : une fois les corrections complétées, le modérateur (et parfois un vérificateur) vont s'assurer que tous les défauts identifiés sont corrigés ou du moins, adressés s'il s'agit d'un problème qui devra être corrigé plus tard.
- L'analyse : dernière et importante étape de la revue, l'analyse des défauts trouvés. Le modérateur (et peut-être les experts du processus de l'entreprise) va vérifier pourquoi ces défauts ont été introduits et déterminer s'il y a une façon de les éliminer à la source en changeant le processus, avec des outils ou d'autres techniques d'ingénierie logicielle. Si ce n'est pas le cas, ils détermineront si ce défaut devrait se retrouver sur la liste de vérification pour être détecté plus facilement lors de revues futures.

La vitesse des inspections est également un élément à considérer. Comme décrit plus haut, la vitesse d'inspection est un des éléments qui permet le mieux de prédire le nombre de défauts trouvés dans les artéfacts. La littérature sur le sujet suggère une vitesse d'environ 150 à 200 lignes de codes (sans les commentaires) par heure et de trois à quatre pages par heure pour les documents textuels comme les spécifications. Il s'agit de moyennes qui peuvent varier grandement d'une entreprise à l'autre tout dépendant des technologies et des modèles de documents utilisés. Un document de spécifications très aéré sera probablement inspecté plus

rapidement sans pour autant compromettre la qualité du travail. Il s'agit toutefois d'un rythme relativement lent, prouvé comme étant efficace, mais qui peut rebuter certaines équipes, surtout si elles ont l'habitude de faire des documents de spécifications de plusieurs centaines de pages.

L'inspection est de loin la méthode la plus formelle et tend à détecter le plus grand nombre de défauts, mais c'est également la plus coûteuse des méthodes de revue, tant pour le nombre d'heures nécessaires que pour le nombre de personnes impliquées.

2.2.2 La revue d'équipe (*team review*)

C'est une version légère de l'inspection. Elle est un peu moins formelle dans les rôles. Par exemple, l'auteur fait souvent office de modérateur et il n'y a généralement pas d'étape de vérification une fois les corrections apportées. Elle comporte toutefois les éléments importants de l'inspection soit la présence d'au moins deux inspecteurs, une préparation individuelle suivie d'une rencontre pour échanger sur les différents défauts trouvés.

La revue d'équipe tendrait à trouver un peu moins de défauts que l'inspection et n'est pas aussi stricte sur les étapes à suivre, principalement après la rencontre, mais conserve les éléments clés de l'inspection comme la préparation individuelle et la rencontre.

2.2.3 Parcours du code ou autre artéfact (*walkthrough*)

Avec cette méthode, l'étape de préparation individuelle est escamotée et l'auteur convoque plusieurs personnes à une rencontre où il va parcourir l'artéfact à réviser en sollicitant les commentaires et suggestions des gens présents. Il n'y a aucune mention du nombre de personnes qui doivent être présentes, pas plus que les rôles de chacun.

Il y a peu de données sur l'efficacité des parcours de code, car, étant plus informelles, peu de données sont recueillies et il est donc difficile d'établir des ratios. Cependant, certaines rares

études sur le sujet notent une diminution de détection des défauts de 50 % par rapport à l'inspection (Bankes and Sauer 1995).

Cette technique a toutefois le pouvoir de se faire rapidement et sans préparation, ce qui peut être utile pour certains types de revues comme des revues d'interface utilisateur, des propositions d'architectures ou des remue-méninges pour déterminer une implémentation ou un modèle de données par exemple. Toutefois, comme l'auteur est en contrôle de tous les éléments de la rencontre et qu'il n'y a pas de préparation des autres participants, les parcours de code ne sont d'aucune utilité pour détecter des erreurs dans des parties de document que l'auteur juge déjà adéquates, ce qui n'est pas nécessairement le cas.

2.2.4 Programmation en binôme

Popularisée par les adeptes de l'XP, la programmation en binôme consiste à former des équipes de deux développeurs qui travailleront ensemble sur le même ordinateur. Un des coéquipiers est au clavier (le conducteur) pendant que l'autre (le réviseur) regarde le code qui est écrit, commente et fait des recommandations. À intervalle régulier, généralement toutes les heures, les rôles sont inversés et le conducteur devient le réviseur. Pour les tenants de cette méthode, il s'agit ni plus ni moins que d'une revue en continu du code. Cette interaction constante entre deux personnes permet d'attraper plus rapidement les erreurs, d'identifier les parties du code qui ne sont pas assez claires et aide à produire un meilleur design. Aussi, le fait d'avoir plus d'une personne qui connait très bien le code est un avantage pour les entreprises où le taux de roulement du personnel est élevé puisqu'on limite la perte d'expertise.

Ici, pas de documents à préparer, pas de rencontre, tout ce fait au fur et à mesure. La revue est cependant informelle, car aucune trace des défauts rencontrés n'est laissée, ce qui ne permet pas aisément de mettre en place des listes de vérification qui pourraient profiter à d'autres paires de développeurs. Aussi, cette technique ne profite pas d'un œil extérieur qui pourrait identifier des incohérences ou un manque de clarté dans le code, car les deux personnes qui révisent le code en ont une connaissance fine. Il va sans dire que cette technique

s'applique plus difficilement à d'autres types d'artéfacts comme les documents ou les diagrammes.

2.2.5 Revue individuelle par un pair (*Peer Deskcheck*)

Dans ce type de revue, on demande à un collègue de revoir un bout de code ou une section de document. Celui-ci l'étudiera individuellement et enverra ses commentaires à l'auteur. C'est un type de revue très économique, car elle n'implique qu'une personne et celle-ci n'est occupée que pour le temps nécessaire à réviser le document (pas de préparation, pas de rencontre). Comme ce type de revue n'implique qu'une seule personne, il peut cependant y avoir une très grande variabilité dans le taux de défauts trouvés. Tout repose sur les connaissances, l'attention et le bon vouloir du reviseur. Aussi, la disponibilité de celui-ci peut parfois être problématique si les délais de livraison sont serrés et que le réviseur est très pris par ses propres projets.

2.2.6 Revue individuelle par plus d'un pair (*Passaround*)

Ce type de revue est exactement le même qu'une revue par un pair sauf que l'artéfact à réviser est envoyé à plus d'une personne. Cette façon de faire réduit les chances de voir des éléments importants oubliés puisque tout ne repose plus sur une seule personne, mais sur plusieurs. On réduit également les chances de devoir attendre longtemps pour obtenir des commentaires, car il est probable qu'un ou plusieurs des réviseurs enverront leurs commentaires dans un délai raisonnable. De plus, comme les personnes n'ont pas à être disponibles en même temps pour une rencontre d'inspection, il est possible de demander à plus de réviseurs de travailler sur le document, permettant ainsi de récolter un maximum de commentaires en un minimum de temps.

Lorsque plus d'une personne révise un artéfact de façon individuelle, il peut y avoir certains défis à réunir tous les commentaires, surtout si certains d'entre eux sont contradictoires. Il revient alors à l'auteur de décider, ou alors il doit convoquer les réviseurs en réunion pour discuter de la situation et tenter de déterminer quelle voie doit être prise. Si ce genre de situa-

tion se produit fréquemment, on perd un peu la flexibilité de la revue individuelle et une inspection plus formelle aurait alors été aussi efficace puisque les points litigieux auraient alors été discutés directement.

2.2.7 Revue ad hoc

Le type de revue le plus informel et aussi le plus populaire, car tout le monde l'a probablement déjà utilisé un jour ou l'autre : un auteur demande à un collègue de passer à son bureau pour vérifier un bout de code ou de document et de passer ses commentaires. Comme ce type de revue n'implique qu'une seule personne, que cette personne n'est pas dédiée à cette tâche, mais doit plutôt arrêter ce qu'elle fait pour aider son collègue, il peut y avoir une très grande variabilité dans le taux de défauts trouvés. Tout repose sur les connaissances, l'attention et le bon vouloir du reviseur.

2.2.8 La méthode d'inspection du CRIM

La méthode développée par Sylvie Trudel pour le CRIM (Trudel 2005) est basée sur le processus d'inspection tel que défini par Gilb et Graham (Gilb and Graham 1993). En plus d'avoir été francisée, ce qui est un plus pour les entreprises québécoises, elle résume bien les rôles et les étapes à suivre et plusieurs aide-mémoires (ou « *cheat sheet* ») ont été créés pour qu'il soit facile de visualiser le processus. Elle porte également une attention particulière à l'aspect humain de la revue, qui sera discuté en détail dans la section suivante.

2.3 L'aspect humain dans la revue

Au-delà de toutes les techniques décrites précédemment, il a été démontré qu'aucune d'entre elles ne peut être mise en place avec succès si l'on ne tient pas également compte de l'aspect humain dans le processus de revue. En effet, il est important de faire preuve de beaucoup de doigté et de diplomatie lors des revues, sans quoi l'auteur pourrait sentir qu'on l'attaque personnellement en critiquant son travail. Ce n'est pas le but de la revue par les pairs, mais les perceptions des individus pourraient en être autrement.

Cet aspect de la revue est tellement important que Tom Gilb a jugé bon d'inclure une section sur la bonne attitude à avoir lors des revues dans son livre. Il donne quelques stratégies pour amoindrir les impacts émotifs sur les participants. Quelques exemples tirés de ses constatations (Gilb and Graham 1993 (p. 187-194)) :

- Il ne faut en aucun cas critiquer l'auteur. Il faut tenter d'identifier les éléments de l'artéfact qui ne correspondent pas aux normes établies par l'équipe et décrire le plus objectivement possible laquelle des normes n'est pas respectées et pourquoi.
- Il est important de ne pas chercher à trouver une meilleure façon de faire dans le travail de l'auteur. Chacun a une opinion sur la façon dont une fonctionnalité peut être développée. Si le travail de l'auteur respecte les normes établies par l'équipe, il n'est pas du devoir de l'inspecteur de donner son opinion sur la façon dont il aurait fait le travail, mais de respecter la façon de faire de l'auteur.
- Utiliser des termes neutres pour décrire les éléments à corriger. Certaines organisations utilisent le terme problème potentiel (*issues*) au lieu de défaut (*defects*).
- Lorsqu'un problème est identifié, tenter de trouver la raison de son introduction en cherchant une faille possible dans le processus de développement. La faute ne devrait en aucun cas être rejetée sur l'auteur, car même le meilleur des employés ne peut produire un travail de qualité si les processus mis en place ne le permettent pas. Une fois la faille identifiée, il faut permettre aux participants de suggérer des façons de corriger cette faille pour éviter de détecter à nouveau ce genre de problème dans le futur.
- Les participants doivent sentir qu'ils ont l'appui de la direction et que celle-ci a une volonté réelle d'améliorer la qualité des artéfacts. Si les revues sont mises en place à contrecœur pour respecter les politiques corporatives et que les délais de livraison demandés restent les mêmes malgré une demande d'amélioration de la qualité, les participants se sentiront piégés entre deux choix impossibles à concilier : faire une inspection de qualité ou livrer les fonctionnalités demandées le plus rapidement possible, quitte à devoir rogner sur la qualité.

- Dans le même ordre d'idée, si les mesures de performances des participants demeurent basées sur leur capacité de livrer les artéfacts demandés dans les temps prescrits sans aucune considération pour la qualité de ceux-ci, les participants auront non seulement le sentiment de perdre leur temps, mais que la revue leur est nuisible puisque l'amélioration de la qualité n'aura aucune incidence positive sur la reconnaissance que leur porte l'organisation alors que le temps passé en revue réduit le temps disponible pour livrer des artéfacts.

D'autres constatations pertinentes ont également été faites par le CRIM (Trudel 2005) lors de la mise en place de leur méthode de revue par les pairs :

- Avoir une attitude ouverte et constructive, tant pour les réviseurs que pour l'auteur;
- Permettre à tous de s'exprimer pour éviter qu'un individu domine les discussions et intimide les autres participants;
- Utiliser les inspections pour améliorer le produit et les processus et non comme méthode d'évaluation, de récompense ou de réprobation des individus;
- Pour la même raison, toujours fournir des résultats collectifs, jamais individuels.

Bref, une attention particulière doit être mise sur le respect de tous les individus qui participent aux revues. Tout le monde a quelque chose à apprendre, même les plus expérimentés. Les gens ont des réactions différentes face à la critique et il est important de faire le maximum d'efforts pour que tout le monde sente que le processus les aide à progresser dans l'objectif de livrer chaque jour un travail de meilleure qualité. Ils doivent également sentir que l'organisation les appuie dans cette démarche. Ce sont des conditions essentielles à la réussite de la mise en œuvre d'un processus de revue.

CHAPITRE 3

MÉTHODOLOGIE

3.1 Objectifs

Le but premier de la revue par les pairs est de détecter le plus rapidement possible les défauts pour en atténuer l'impact négatif sur la qualité et le coût du produit livré. Un défaut trouvé rapidement après son introduction est plus simple et moins coûteux à corriger que s'il est détecté plus tard dans le cycle de vie du logiciel :

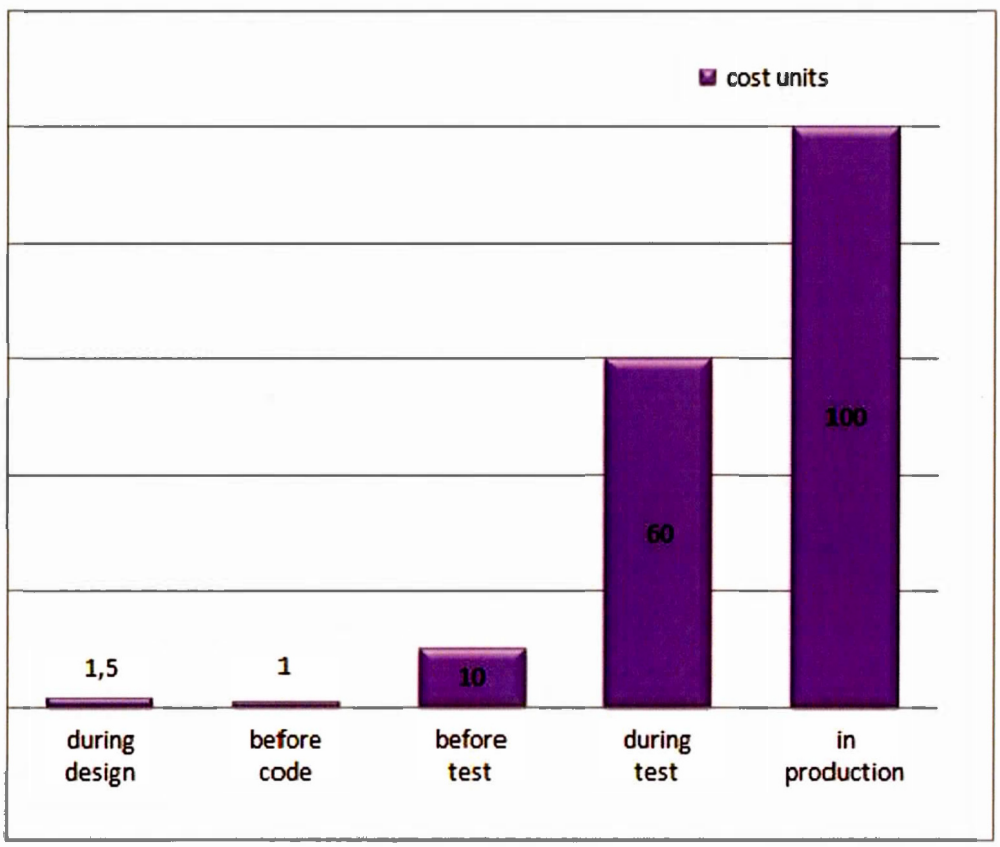


Figure 3.1 Coût relatif de correction selon l'étape de détection. (Sprunck 2012)

Les données de ce graphique sont tirées des études réalisées par Gilb (Gilb and Graham 1993) dans les années 70 et 80. Le domaine du logiciel a énormément évolué et le coût de

livraison des applications a grandement diminué depuis avec les déploiements automatisés sur le web ou les plateformes infonuagiques qui ont remplacé le transfert du logiciel sur des médias physiques (disquettes, CD ou autre) qui devaient être expédiés aux utilisateurs. Toutefois, même si le coût relatif d'une mise en production a été réduit, les inconvénients pour les clients et les utilisateurs demeurent et la perte de productivité potentielle engendrée par un défaut dans le logiciel fait qu'encore aujourd'hui, il est beaucoup plus avantageux de détecter les erreurs avant de déployer le logiciel en production même si les ratios sont probablement différents.

Les objectifs spécifiques du CIRX pour le processus de revue par les pairs dans l'équipe Web-Micro sont :

- Objectif 1 : Assurer que les bonnes pratiques de sécurité du code soient systématiquement appliquées et vérifiées.
- Objectif 2 : Réduire l'effort de correction de défauts (*rework*) qui peut représenter, selon les études, une moyenne de 44 % des coûts d'un projet (voir Figure 3.2).
- Objectif 3 : Augmenter la qualité interne du code et des documents pour en réduire l'effort de maintenance.
- Objectif 4 : Assurer un transfert de connaissances sur les applications.
- Objectif 5 : Partager les connaissances et les compétences techniques.
- Objectif 6 : Uniformiser les pratiques de développement entre les différents projets.

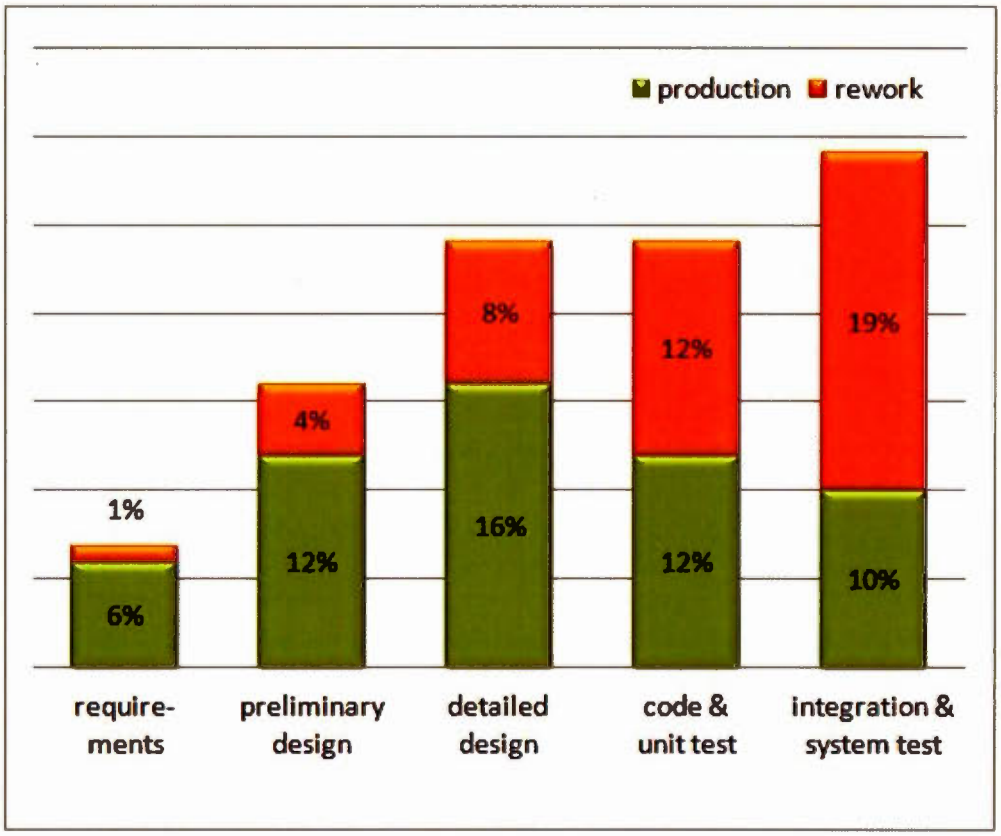


Figure 3.2 Proportion entre les efforts de développement « productifs » et le *rework* (Sprunck 2012)

Dans le cadre du projet, les objectifs sont :

1. Mesurer l'effort de *rework*.
2. Augmenter la qualité du code.
3. Inclure des pratiques de vérification de sécurité dans le processus.
4. Évaluer la satisfaction des développeurs (les participants aux revues sont-ils davantage satisfaits qu'avant l'introduction des revues?).

3.2 Portée

Pour des raisons pratiques, l'expérimentation a été limitée à Web-Micro. Lors du démarrage de nouveaux projets ou de demandes de changement, les directeurs, en collaboration avec

l'architecte, détermineront si ces projets devront faire partie de l'initiative. Pour le déterminer, certains critères seront pris en compte :

- Taille du projet : Pour les premiers projets, l'idéal serait de tester le processus de revue sur des projets de taille moyenne (20 à 150 PFC). Dans un projet trop petit, il est fort probable, pour le moment, que la revue n'apporte que peu de valeur. Dans un projet trop grand, l'équipe pourrait avoir trop de défis à bien cadrer la nouvelle pratique de revue par les pairs dans son processus.
- Complexité du projet : le processus de revue sera testé sur des projets de complexité moyenne.
- Singularité du projet : le processus de revue doit être testé avec des projets typiques généralement réalisés par l'équipe. Nous excluons les projets atypiques puisque, en raison de leur nature, ils sont peu nombreux et présentent peu d'intérêt pour le moment.

Il peut s'agir de nouveaux projets ou de requêtes de maintenance. Toutefois, dépendant des projets démarrés durant la phase de réalisation, il est possible que des projets plus importants soient également inclus, mais qu'ils soient découpés en incréments (bien que le cycle de développement soit davantage de type *Waterfall*) pour obtenir des résultats plus rapidement. L'idéal serait de pouvoir réaliser plusieurs revues par projet afin de pouvoir ajuster le processus au fur et à mesure que des ajustements sont nécessaires et déterminer si ces ajustements apportés ont un impact positif sur les objectifs que l'équipe veut atteindre.
Pour les besoins de l'initiative, la direction a un objectif d'au moins quatre projets pour lesquels quelques revues auront été réalisées. Une dizaine de revues au total serait un échantillon minimum pour l'analyse.

3.3 Hypothèses

Dans le cadre du projet, les hypothèses suivantes seront vérifiées:

- H1. La revue par les pairs diminuera le nombre de défauts trouvés lors des tests intégrés (par l'équipe de QA et lors des tests utilisateurs) et en production.

- H2. L'effort de revue sera compensé au moins entièrement par une réduction du coût de *rework*. La productivité de l'effort de développement et de maintenance (en heure par PFC) devra demeurer stable et idéalement s'améliorer une fois le processus de revue mis en place et après une période d'apprentissage par les membres de l'équipe.
- H3. Le partage de connaissances sur les applications et les compétences techniques améliorera la satisfaction des développeurs en leur permettant d'acquérir de nouvelles connaissances qui les rendront plus productifs. De plus, l'apprentissage de nouvelles façons de faire pourra leur donner des idées nouvelles sur la façon d'attaquer et de régler les problèmes qui se présenteront à eux à l'avenir.

3.4 Aperçu de la méthodologie

La méthodologie utilisée pour cette initiative comprend quatre phases et un ensemble d'activités à réaliser. La figure suivante résume ces différentes activités et les sections suivantes les décriront en détail.

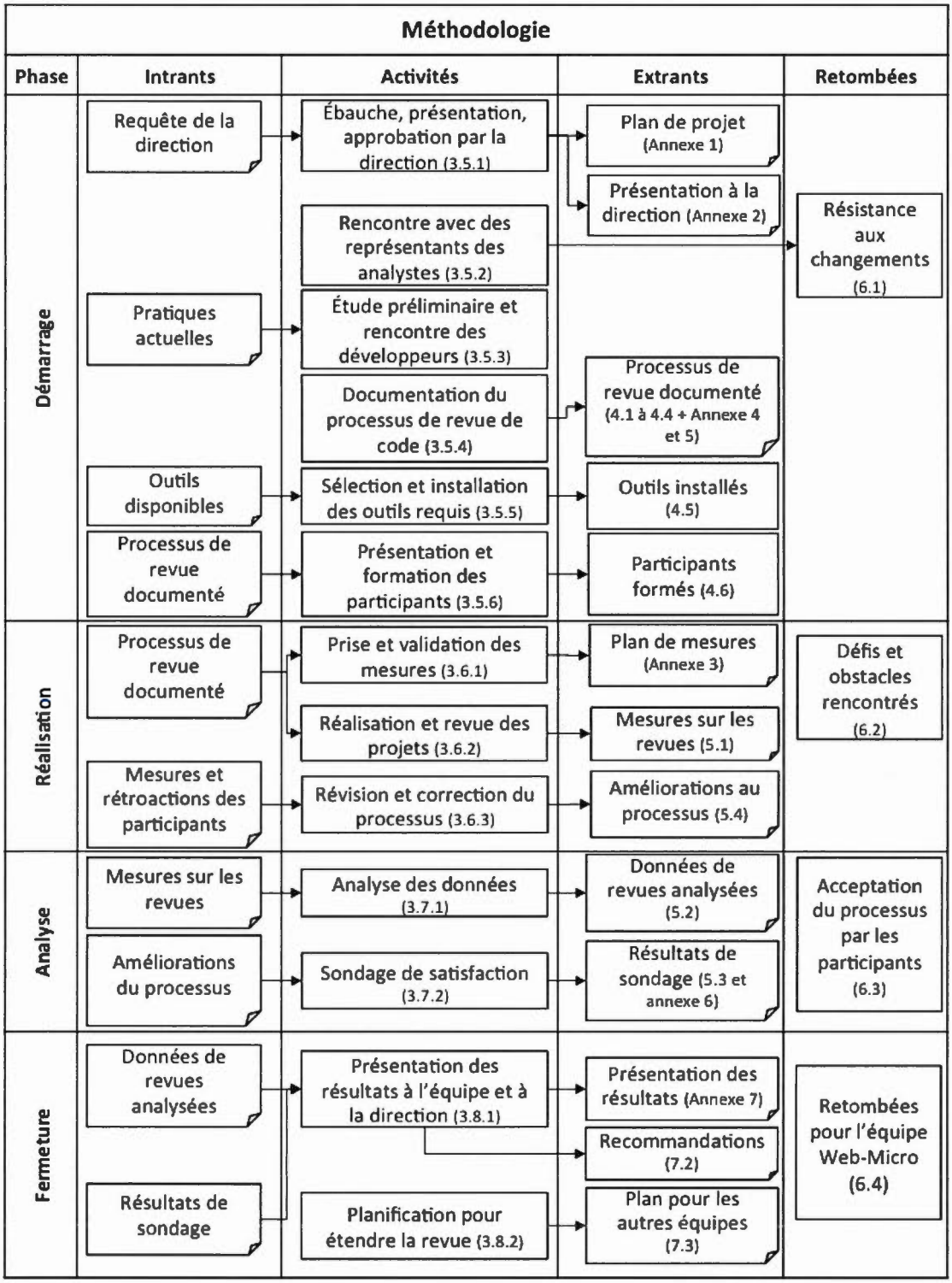


Figure 3.3 Méthodologie utilisée pour l'initiative

3.5 Phase de démarrage

3.5.1 Ébauche, présentation et approbation par la direction

La première étape du projet consiste à préparer une ébauche de l'initiative pour la direction du CIRX. Cette ébauche prendra la forme d'un document de vision (disponible à l'ANNEXE I) énonçant les grandes lignes du projet soit la vision du projet, les objectifs du projet, les critères de succès, les leviers, la gestion des risques, l'échéancier et les hypothèses. En complément du document de vision, une présentation sera préparée et le vice-président adjoint Développement des systèmes d'information ainsi que les directeurs de l'équipe Web-Micro y seront convoqués (disponible à l'ANNEXE II). Le but de cette rencontre est de leur faire une description du projet de revue par les pairs, leur présenter les objectifs, l'échéancier et obtenir leur approbation. Le projet pourra alors officiellement démarrer.

3.5.2 Rencontre avec des représentants des analystes

La méthodologie du CIRX comporte déjà une phase de révision des documents d'analyse. Cependant, aucune norme officielle n'est donnée sur la façon de réaliser ces revues, que ce soit leur fréquence, leur durée, le nombre de participants, la documentation des commentaires et des recommandations, le suivi des rencontres, les mesures, etc. Pour cette raison, une rencontre avec un ou des représentants des analystes est prévue pour en savoir un peu plus sur le processus de revue qui est actuellement appliqué. À partir de celle-ci, une proposition de processus de revue des documents d'analyse sera faite en tenant compte des pratiques reconnues d'inspection, mais en tenant compte aussi des habitudes actuelles, dans l'objectif d'en favoriser l'adoption et de réduire de possibles résistances au changement.

Suite à certaines réticences des analystes et pour pouvoir débuter l'initiative plus rapidement, la direction a décidé le limiter la revue au code dans un premier temps. Toutefois, les données recueillies pourront permettre de valider si la revue des documents devrait être mise en place.

3.5.3 Étude préliminaire et rencontre des représentants des développeurs

Comme pour les analyses, la méthodologie du CIRX comporte une phase de revue de code. Ici aussi, aucune norme officielle n'est donnée sur la façon de faire. Les développeurs étant plus nombreux que les analystes, une rencontre avec seulement des représentants des développeurs ne semble pas suffisante pour obtenir un point de vue global sur la façon dont sont menées les revues. Une rencontre avec tous les développeurs semble également problématique puisque le grand nombre d'intervenants rendrait la conversation difficile. Afin de mieux comprendre comment les développeurs réalisent les revues, ceux-ci seront plutôt encouragés à remplir des fiches de revue papier dans le but de capturer certaines informations comme le nombre de revues réalisées, le temps consacré aux revues (nombre de classes inspectées en combien de temps), les principaux défauts trouvés. Ils sont aussi encouragés à suggérer des points importants à inclure dans le futur processus de revue de code. Le but premier de cet exercice est de comprendre les façons de faire pour mettre en place un processus de revue qui sera le plus près possible de la réalité de l'équipe, tout en permettant d'atteindre les objectifs. Les résultats de ces fiches de revue seront compilés et alors présentés à des représentants des développeurs qui pourront confirmer si les conclusions tirées de la compilation des fiches de revue sont bonnes et si elles peuvent servir de base pour l'élaboration du nouveau processus de revue.

3.5.4 Documentation du processus de revue de code

Une fois les rencontres complétées, l'étape suivante consistera à documenter le processus de revue pour les artéfacts à réviser. Ce document détaillera les différentes étapes du processus et pour chacune d'entre elles, expliquera les tâches à réaliser, comment les réaliser et identifiera les intervenants requis pour l'étape. Cette documentation devra être assez détaillée pour permettre à un nouveau membre de l'équipe de bien comprendre le processus et être en mesure de l'appliquer rapidement. De plus, la procédure sera adaptée aux outils que l'équipe aura décidé d'utiliser et fera ainsi office de guide d'utilisation.

La documentation comprendra également des détails sur les éléments à inclure dans les revues comme des listes de vérification (*checklists*), des descriptions de techniques et des outils d'analyse. Pour ce qui est des revues de code, ces listes de vérification seront initialement construites à partir des défauts récurrents identifiés dans les fiches de revue des développeurs. Pour ce qui est des documents d'analyse, les listes de vérification seront construites à partir de rencontres avec les analystes qui se serviront de leurs expériences pour déterminer les éléments prioritaires à inclure. Par la suite, les listes de vérification seront révisées périodiquement pour inclure de nouveaux défauts qui tendent à revenir régulièrement et à retirer ceux qui se produisent rarement ou dont l'impact est minime. Le but de cet exercice est de conserver une liste relativement concise qui tient compte des priorités de l'équipe et qui peut facilement être comprise et appliquée par tous.

Le guide comprendra également une section pertinente sur les comportements à adopter et à éviter par chacun des participants et parties prenantes aux revues. Un participant aux revues pourrait accueillir les défauts soulevés favorablement ou pas, en fonction des comportements des autres participants (Gilb and Graham 1993 (chapitre 8.11)).

3.5.5 Sélection et installation des outils requis

Une fois le processus documenté, les outils requis devront être installés par l'équipe technique du CIRX et configurés par l'équipe Web-Micro pour répondre à ses besoins et ses façons de faire. Les outils à installer et à adapter comprennent entre autres :

- Un outil pour la gestion du code source
- Un outil pour demander et faire les revues de code.
- Un outil pour documenter et suivre l'évolution des défauts.
- Un outil de mesure de la qualité du code.
- Un outil pour consigner le temps passé à effectuer des revues
- Un outil pour documenter et mesurer les défauts trouvés après la revue.
- Un outil pour documenter les défauts récurrents et les améliorations potentielles au processus de revue.

La priorité pour les outils sera de minimiser l'impact sur le travail des participants. Si l'entreprise possède déjà un outil pour satisfaire le besoin, celui-ci sera priorisé. Sinon, l'équipe évaluera les outils disponibles sur le marché et les évaluera selon une liste de critères déterminés.

3.5.6 Présentation et formation des participants aux revues

Finalement, avant le démarrage officiel de l'initiative, les différents groupes de l'équipe seront rencontrés pour leur expliquer le projet et ce qu'on attend d'eux. Comme les processus de revue risquent d'être différents pour les analystes et les développeurs, il est probable que ces deux groupes seront rencontrés séparément. Si, une fois le processus clairement détaillé, la quantité d'information à transmettre semble trop importante, des sessions de formation en petits groupes pourraient être organisées. Il est important à cette étape que toutes les personnes de l'équipe aient une bonne compréhension du processus pour pouvoir demander ou réaliser des revues.

3.6 Phase de réalisation

3.6.1 Prise et validation des mesures

Pour déterminer que les objectifs de l'entreprise sont atteints ou non, des mesures seront prises tout au long des revues. Les principales mesures nécessaires sont :

- L'effort passé à faire des revues (en heures) afin d'en dériver les coûts.
- L'effort de correction des défauts (*rework*) (en heures).
- Le nombre de défauts trouvés en test et en production.
- L'évolution de la qualité du code (indices de qualité calculés par SonarQube)
- La productivité de l'équipe (en heures/PFC).

Le plan de mesure détaillé peut être consulté à l'ANNEXE III.

L'architecte, responsable de l'initiative, fera des vérifications périodiques pour s'assurer que les données nécessaires aux mesures sont correctement entrées dans les différents outils et des ajustements au processus seront faits si nécessaires. Les informations colligées seront agrégées et tout contenu nominatif est retiré pour éviter toute tentative d'utilisation des revues comme outils d'évaluation individuelle des membres de l'équipe. Il s'agit d'un exercice collectif et les mesures devront refléter les tendances de l'équipe.

3.6.2 Réalisation et revue des projets

Une fois le plan de mesure mise en place, l'initiative peut commencer. Les développeurs seront mandatés pour revoir le code de leurs collègues en utilisant les listes de vérification produites lors de la phase de démarrage. Ils seront aussi responsables de consigner correctement le temps passé à faire des revues dans l'outil de feuille de temps ainsi que le temps passé sur des tâches de *rework*.

3.6.3 Rétroaction, révision et correction du processus

À la fin d'un projet, d'une demande de changement ou d'une itération (tout dépendant de la durée du projet), les défauts trouvés lors des tests ou en production seront analysés pour déterminer si ceux-ci auraient pu être détectés par les listes de vérification. Si c'est le cas et que c'est un problème récurrent, la raison à l'origine du défaut pourrait être incluse dans la liste de vérification. Si le défaut est détectable grâce à un analyseur statique de code, une règle pourra également être ajoutée pour intercepter ce genre de problème automatiquement lors des analyses futures.

Les participants aux revues (auteurs comme réviseurs) auront également accès à un outil de type « wiki » pour proposer des améliorations au processus. Les améliorations proposées seront alors analysées par les directeurs pour s'assurer qu'elles ne vont pas à l'encontre des objectifs et si elles semblent intéressantes, elles seront débattues lors des rencontres d'équipe.

Les listes de vérification et les techniques de revues pourraient également être révisées lors de changements technologiques qui pourraient demander de nouvelles vérifications ou au contraire, rendre certaines anciennes règles caduques.

3.7 Phase d'analyse

3.7.1 Analyse des données

Étant donné qu'il s'agit d'une nouvelle initiative, toute une série de mesures sera prise tout au long du projet, tel que décrit ci-dessus. Comme il n'existe pas à ce jour de mesures pour tous les éléments d'informations nécessaires, il n'y a pas de valeurs de référence (*baseline*) avec lesquelles comparer les résultats obtenus durant le projet. La progression de l'équipe vers les objectifs attendus sera plutôt analysée durant le projet. Comme on veut mesurer plusieurs projets à plusieurs moments de leur cycle de développement, l'hypothèse est qu'au fur et à mesure des apprentissages et de l'expérience acquise, les données recueillies permettront de constater si les objectifs sont en voie d'être atteints.

Par l'analyse de ces données, la direction cherche à obtenir des réponses aux questions suivantes :

- Est-ce que l'augmentation du temps de revue diminue l'effort de *rework* et si oui, dans quelle proportion?
- Est-ce que la diminution du temps de *rework* est supérieure au temps passé en revue?
- Est-ce que l'augmentation du temps de revue augmente l'indice de maintenabilité du projet (et dans quelle proportion)?
- Est-ce que l'augmentation de l'indice de maintenabilité augmente la productivité de l'équipe (en heures/PFC)?

C'est seulement après avoir répondu à ces questions que la direction pourra décider si les objectifs ont été atteints.

3.7.2 Sondage de satisfaction

Dans les objectifs du projet, le transfert de connaissances et de compétences a été identifié comme étant crucial à la réussite de l'initiative. Il est cependant plutôt difficile de mesurer quantitativement ces éléments. C'est pourquoi ils seront plutôt mesurés à partir d'un sondage de satisfaction envoyé aux participants du projet à la fin de l'initiative dans le but d'obtenir leur point de vue sur le projet, mais également sur leur expérience en tant qu'auteur et réviseur sur les apprentissages qu'ils ont faits et les connaissances qu'ils ont pu partager avec des collègues. Voici un exemple de questions qui seront incluses dans le sondage de satisfaction :

- À combien d'inspection avez-vous participé (comme réviseur et auteur)?
- Quelle a été votre degré de satisfaction en tant que réviseur (selon une échelle de 0 à 10 : 0 étant très insatisfait et 10 étant très satisfait)?
- Quelle a été votre degré de satisfaction en tant qu'auteur (selon une échelle de 0 à 10 : 0 étant très insatisfait et 10 étant très satisfait)?
- Est-ce que les revues vous ont aidé à améliorer vos compétences techniques? Dans quelle mesure (0 à 10 : 0 étant pas du tout, 10 étant énormément)?
- Est-ce que les revues vous ont permis de partager vos connaissances techniques avec vos collègues? Dans quelle mesure (0 à 10 : 0 étant pas du tout, 10 étant énormément)?
- Quelle est votre satisfaction du processus actuel de revues (de 0 à 10 : 0 étant pas du tout, 10 étant énormément)
- Qu'avez-vous appris de votre expérience?
- Avez-vous des suggestions pour améliorer les revues?

3.8 Fermeture du projet

3.8.1 Présentation des résultats à l'équipe et à la direction

Une fois les données compilées, analysées, les sondages réalisés et interprétés, les résultats de l'initiative seront présentés à l'équipe. Cette présentation aura pour but de leur permettre de voir le fruit de leurs efforts et constater la progression (ou non) de l'équipe durant le projet. Cette rencontre sera également une occasion de valider avec eux si les conclusions tirées des mesures et des sondages sont justes selon leur perspective et sinon, quels ajustements dans l'interprétation des données doivent être faits pour refléter plus fidèlement la réalité, avant de les présenter à la direction. Comme le projet se veut une mise en place des revues, l'objectif est que l'équipe continue d'appliquer les méthodes apprises. Il est donc primordial d'obtenir leur collaboration et leur approbation des conclusions avant la publication à la direction.
La présentation à la direction sera similaire à celle faite aux membres de l'équipe. Ce sera une occasion de leur démontrer les travaux réalisés, les résultats obtenus et les travaux futurs si nécessaire. Idéalement, la fin de l'initiative ne sera pas la fin du processus de revue, mais plutôt une étape dans le processus d'amélioration continue de l'équipe.

Suite aux rencontres avec les membres de l'équipe et la direction, des recommandations seront faites pour proposer des solutions aux irritants rencontrés en cours de réalisation. Des stratégies seront élaborées pour mieux les comprendre et les éliminer ou à tout le moins, en atténuer les effets. Dépendant de la nature de ces recommandations, un plan de mise en place pourra être détaillé, selon les désirs de la direction et des membres de l'équipe. Si des travaux futurs ont été identifiés lors des rencontres, la mise en œuvre de ceux-ci pourra également faire l'objet de recommandations.

3.8.2 Planification pour étendre la revue aux autres équipes et aux autres artéfacts

L'initiative de revue de code pour l'équipe Web-Micro se veut un premier pas vers une mise en place plus généralisée de la revue par les pairs au CIRX. À partir des résultats obtenus lors de l'initiative, la direction désire obtenir un plan de déploiement du processus de revue aux autres équipes du CIRX. Ce plan devra entre autres expliquer les parties du processus qui peuvent s'appliquer à toutes les équipes uniformément et les parties qui devront faire l'objet d'adaptations pour se coller à la façon de travailler des équipes.

La revue des autres artéfacts de développement – notamment les documents d'exigences et de spécifications – devra également être évaluée. Par mesure d'efficacité, elle avait été exclue de la portée initiale du projet, mais il faudra y revenir, dans l'éventualité que l'analyse des défauts trouvés révèle que les documents d'analyse en sont régulièrement la source.

CHAPITRE 4

PROCESSUS DE REVUE PAR LES PAIRS

Le processus de revue par les pairs présenté dans le présent chapitre est orienté sur la revue du code. Il a été développé en collaboration avec les membres de l'équipe Web-Micro à la suite de rencontres et de production de fiches de revues papier qui ont été réalisées durant l'année précédant l'initiative. L'objectif de ces rencontres et de ces analyses préliminaires était de bien comprendre comment les éventuels participants concevaient le processus de revue de code. Il a été ainsi possible de concevoir un processus qui se collait bien aux réalités de l'équipe et qui était adapté aux défis et objectifs identifiés par la direction.

Le processus de revue utilisé s'inspire du processus utilisé dans les projets de logiciel libre (*open source*) tel Chromium de Google (Bergeron 2016). Dans ce type de revue, tout se fait de façon asynchrone, c'est-à-dire que l'auteur fait une demande de revue à travers un outil et le réviseur effectue la revue à son rythme et selon ses disponibilités. L'auteur et le (ou les) réviseur(s) n'ont donc pas besoin de trouver physiquement au même endroit durant la revue. Toutefois, les rencontres sont encouragées, principalement après la revue pour discuter des défauts trouvés et des améliorations à apporter.

Un document sur le processus adapté pour l'équipe a été produit et partagé aux participants aux revues. Ce document peut être consulté à l'ANNEXE V. Les grandes lignes du processus de revue seront détaillées dans les sections à venir.

4.1 Élément à réviseur lors de la revue

La revue de code a plusieurs utilités. Durant l'inspection, on cherche à détecter des défauts réels ou potentiels du code inspecté dans le but d'en améliorer la qualité pour le bien de

l'équipe. Les éléments suivants sont entre autres évalués à l'aide de listes de vérification (voir la section 4.1.1) :

- La sécurité du code.
- La structure des projets .NET et des fichiers.
- Le respect des normes de codage (clarté, lisibilité et maintenabilité).
- L'utilisation de pratiques reconnues et acceptées par l'équipe (architecture et patrons).
- La présence et la qualité des tests (unitaires et d'intégration).

Lors de la revue, le code (client et serveur) produit par les développeurs sera inspecté. Il n'y a pas d'inspection du code généré par les outils, des librairies externes ou qui ne font pas partie du projet. Par contre, une mauvaise utilisation de ces librairies peut constituer un défaut à reporter. Pour détecter ces défauts, plusieurs outils ont été mis à la disposition des réviseurs.

4.1.1 Listes de vérification

Dans le but de porter une attention particulière sur les règles importantes identifiées lors des analyses préliminaires, une liste de vérification (*checklist*) d'inspection a été produite. Elle contient une série d'éléments à vérifier lors des inspections et chaque élément y est détaillé afin de s'assurer de leur bonne compréhension par les développeurs. Il s'agit d'un document de départ qui évoluera avec le temps : si des défauts de même nature sont régulièrement découverts, ils pourront être ajoutés à la liste de vérification. Au contraire, si certains défauts ne sont plus décelés parce que la technologie ou les techniques utilisées rendent cette règle caduque, elle pourra être retirée de la liste. Cette liste demeure toutefois un point de départ : elle contient les éléments à valider de façon plus précise, mais elle n'est en aucun cas exhaustive et les défauts d'une autre nature détectés durant la revue devraient être consignés même s'ils ne font pas partie de la liste de vérification. La liste de vérification est disponible à l'ANNEXE IV.

4.1.2 Outil d'analyse de mesures de code

Une partie de l'analyse du code peut être automatisée à l'aide d'outils. Par exemple, l'utilisation d'un outil de calcul de mesures permet d'obtenir en quelques secondes la liste des classes et des méthodes potentiellement problématiques. Certaines lignes directrices et valeurs limites sur les mesures ont été déterminées. Ces seuils sont adaptés au langage de programmation principalement utilisé par l'équipe (C# de Microsoft, qui est un langage de programmation orienté objet). Ils sont tirés de différentes références (telles (McConnell 2004) et (Shatwani 2010)) et de l'expérience de l'équipe.

Attention

Il s'agit de suggestions et non pas de valeurs absolues. Les seuils aident surtout à identifier les problèmes potentiels et à savoir où porter une attention particulière. Il faut en vérifier le contenu, faire preuve de jugement et ne pas faire (ou suggérer) un remaniement (*refactoring*) important du code simplement pour se conformer aux seuils suggérés. Il faut que le changement ait un impact positif sur la qualité du code (lisibilité, testabilité, maintenabilité, sécurité, etc.).

Seuils suggérés :

Classes :

- Nombre de méthodes par classe : entre 1 et 10.
- Niveau d'héritage (*depth of inheritance*) : moins de 5.
- Couplage : le plus bas possible. Un couplage supérieur à 10 indique un problème potentiel de maintenance à long terme.
- Lignes de code : moins de 500 lignes.

Méthodes :

- Couplage : le plus bas possible. Un couplage supérieur à 4 indique un problème potentiel de maintenance à long terme.

- Lignes de code : idéalement, moins de 50 lignes, mais un maximum de 200 lignes. Un truc souvent utilisé est de tenter de contenir l'ensemble du code de la méthode dans un seul écran. La lisibilité est ainsi grandement améliorée puisque le développeur n'a pas à se déplacer constamment pour lire l'ensemble de la méthode.
- Complexité cyclomatique :
  - Moins de 5 : La complexité est faible, la méthode est correctement factorisée.
  - Entre 5 et 15 : La complexité est moyenne. Il faut porter une attention particulière à ces méthodes, il y a un potentiel de problèmes à la maintenance, mais un *refactoring* n'est pas nécessairement requis.
  - Plus de 15 : La complexité est élevée. Il faudrait penser sérieusement à un *refactoring* pour extraire une partie de la logique dans d'autres méthodes ou d'autres classes. Exception notable à cette règle : une méthode avec un gros *switch/case* pourrait avoir une complexité élevée (un chemin par cas testé), mais demeurer facile à comprendre. En cas de doute, les développeurs sont encouragés à demander l'avis de collègues.

4.1.3 Analyse statique du code

Les outils d'analyse statique du code font tout un travail pour aider à identifier les problèmes potentiels dans le code. L'outil peut d'ailleurs être configuré pour s'exécuter en arrière-plan dans l'interface de développement (*IDE*) et faire des suggestions d'amélioration en temps réel. Lorsqu'un élément dans le code source (un fichier par exemple) est modifié, il peut être intéressant de consulter la liste des suggestions faites par l'outil et faire les corrections proposées si elles ont du sens dans le contexte. En effet, il peut arriver que la suggestion soit bonne en général, mais qu'elle ne s'applique pas dans la classe ou la méthode actuelle à cause du patron utilisé ou d'un algorithme particulier.

Attention

L'outil permet d'appliquer certaines modifications « en lot » dans l'ensemble du code source de l'application. Cette façon de faire n'est pas recommandée à moins d'être convaincu que c'est nécessaire, car cela va générer une grande quantité de changements d'un seul coup ce qui compliquera la tâche du réviseur et pourrait également demander une grande quantité de tests pour s'assurer que le changement appliqué n'a pas d'impact négatif sur le fonctionnement de l'application. Avant de se lancer dans ce genre de changement massif, il serait pertinent de consulter les collègues d'abord et d'inclure ces *refactorings* dans une révision distincte du code pour ne pas avoir à départager ces améliorations de nouvelles fonctionnalités ajoutées en même temps. Cela facilitera grandement le travail du réviseur.

Une technique intéressante est la règle du scout (« *the boyscout rule* » (Martin 2009)) : lorsqu'un changement est fait dans un fichier, il doit être laissé dans un meilleur état qu'avant la modification. Le développeur s'étant assuré que le code qu'il a ajouté ne viole aucune des règles de l'analyseur de code, il peut en profiter pour régler certaines violations déjà présentes dans le fichier modifié. Les violations doivent être attaquées en ordre de sévérité soit celles de niveau critique d'abord, suivi des erreurs, des avertissements et finalement les suggestions. Comme toujours, il faut savoir faire preuve de jugement et n'accepter les suggestions que si elles apportent un gain concret sur l'amélioration de qualité du code dans le contexte actuel.

4.2 Le réviseur

La revue de code doit être faite par un développeur qui n'est pas l'auteur du code. Le directeur responsable du projet affectera un ou des réviseurs au projet au démarrage de celui-ci et idéalement, ils resteront les mêmes pour la durée du projet. Cela améliore l'efficacité et simplifie le suivi des corrections. Ce peut être quelqu'un qui travaille sur le même projet ou non, pourvu qu'il travaille sur une autre section du code que celle à inspecter. Les deux situations comportent des avantages et des inconvénients :

- Un développeur au projet prendra moins de temps à comprendre le contexte du projet. Par contre, il pourrait ne pas remarquer certains problèmes de clarté du code, car il est au courant du « langage » du projet (abréviations, structure, concept d'affaires).
- Inversement, un développeur externe au projet prendra plus de temps pour comprendre le contexte au début, mais ce faisant, pourra plus facilement identifier les parties du code difficiles à comprendre ou qui manque de documentation.

Dans un cas comme dans l'autre, les problèmes tendent à s'atténuer avec le temps.

4.3 Fréquence et durée des revues

La revue de code est une activité qui demande du temps et de la concentration. Pour une question d'efficacité, elle devrait se faire à intervalle régulier. Il ne faut surtout pas attendre la fin du projet (à moins bien sûr qu'il s'agisse d'une petite requête de quelques jours) sans quoi la tâche est beaucoup trop lourde et il est souvent trop tard pour corriger des problèmes de design importants dans l'application qui demanderait un lourd *refactoring*.

Diverses études sur le sujet, présentées dans le chapitre 2, en sont venues à la conclusion que le rythme d'inspection idéal pour le code se situe entre 250 et 1000 lignes de code par heure (ce nombre exclut le code généré automatiquement par les outils), dépendant de la complexité du code inspecté. Un rythme plus rapide ne permet généralement pas de bien comprendre le code et d'y déceler un maximum de défaut. Il faut inspecter à un rythme assez lent pour avoir le temps de réfléchir et se demander non seulement ce qu'il y a d'incorrect dans le code inspecté, mais aussi ce qu'il manque (validations, gestion d'erreurs, traces, etc.). Une omission peut être un problème aussi important qu'un défaut présent dans le code.

Il a été également démontré qu'à partir d'un certain point, les inspections trop longues ne permettent pas de trouver plus de défauts. Comme la fatigue mentale se fait sentir après un certain temps et que cela amène une baisse du niveau de vigilance de l'inspecteur, il est recommandé de limiter les séances de revue à une heure puisque le nombre défauts trouvés tant à plafonner après 60 minutes (voir Figure 4.1).

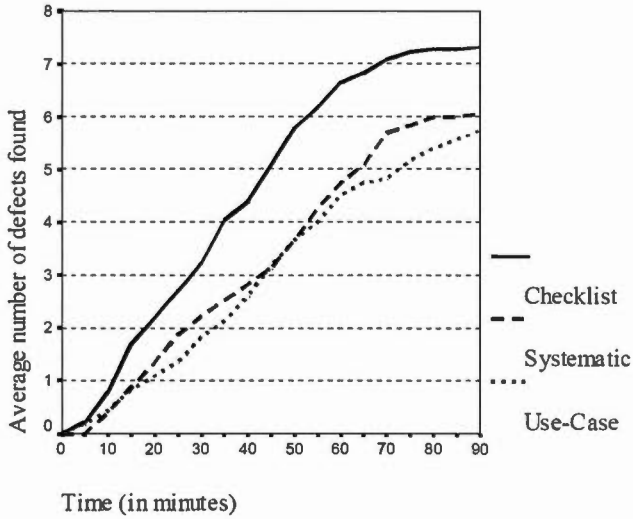


Figure 4.1 Évolution de l'efficacité de la revue dans le temps (Cohen, Teleki et al. 2013)

Il peut être une bonne idée de planifier la plage horaire consacrée à la revue pour éviter les sollicitations et les distractions. Le réviseur devrait idéalement choisir une période de la journée où il est le plus alerte. Certaines personnes préfèrent tôt le matin alors que d'autres sont plus performants plus tard dans la journée.

L'intervalle idéal entre les revues dépend donc de la vélocité du projet : il devrait représenter le nombre de jours nécessaires pour que le code ait suffisamment changé (entre 250 et 1000 lignes de code ajoutées/modifiées, selon les standards d'équipe ou la complexité du code) pour justifier une revue. Dans un processus de développement Agile, la recommandation est généralement d'au moins une revue par sprint. Cependant, comme l'équipe utilise un processus de développement qui se rapproche davantage d'un processus en cascade, une revue hebdomadaire devrait être suffisante dans la plupart des cas, mais dans certains projets où le code change rapidement comme un très gros projet ou en début de projet, il se pourrait qu'il faille faire des revues plus fréquemment. Peu importe l'intervalle choisi, il est important de faire au moins une revue en profondeur en début de projet, car des problèmes initiaux de conception dans l'architecture peuvent être coûteux et difficiles à corriger en cours de projet. Il faut aussi s'assurer d'en faire au moins une avant chaque livraison (soit à la fin du projet,

d'un module ou d'un sprint, dépendant du processus de développement utilisé et de la taille du projet) pour en assurer la qualité.

Bien sûr, il s'agit d'une recommandation minimale et il n'est pas nécessaire de s'en tenir strictement à cela. L'auteur pourrait demander une revue supplémentaire sur une petite portion de code complexe s'il a des doutes. L'inspecteur pourrait également prendre l'initiative de faire des revues un peu plus fréquente en début de projet pour s'assurer que l'application part sur de bonnes bases, spécialement si l'auteur est un développeur junior ou s'il s'agit d'une nouvelle technologie. En cas de doute, c'est une bonne idée de demander l'opinion des collègues ou d'en discuter avec le responsable du projet.

4.4 Étapes de la revue

La revue comme telle est composée de sept étapes. Ces sept étapes ont été inspirées de la méthode de revue par les pairs proposée par le CRIM (Trudel 2005). La Figure 4.2 démontre ces étapes qui seront par la suite décrites en détail. Notez également que les couleurs de remplissage utilisées servent à indiquer le ou les responsables pour chaque étape.

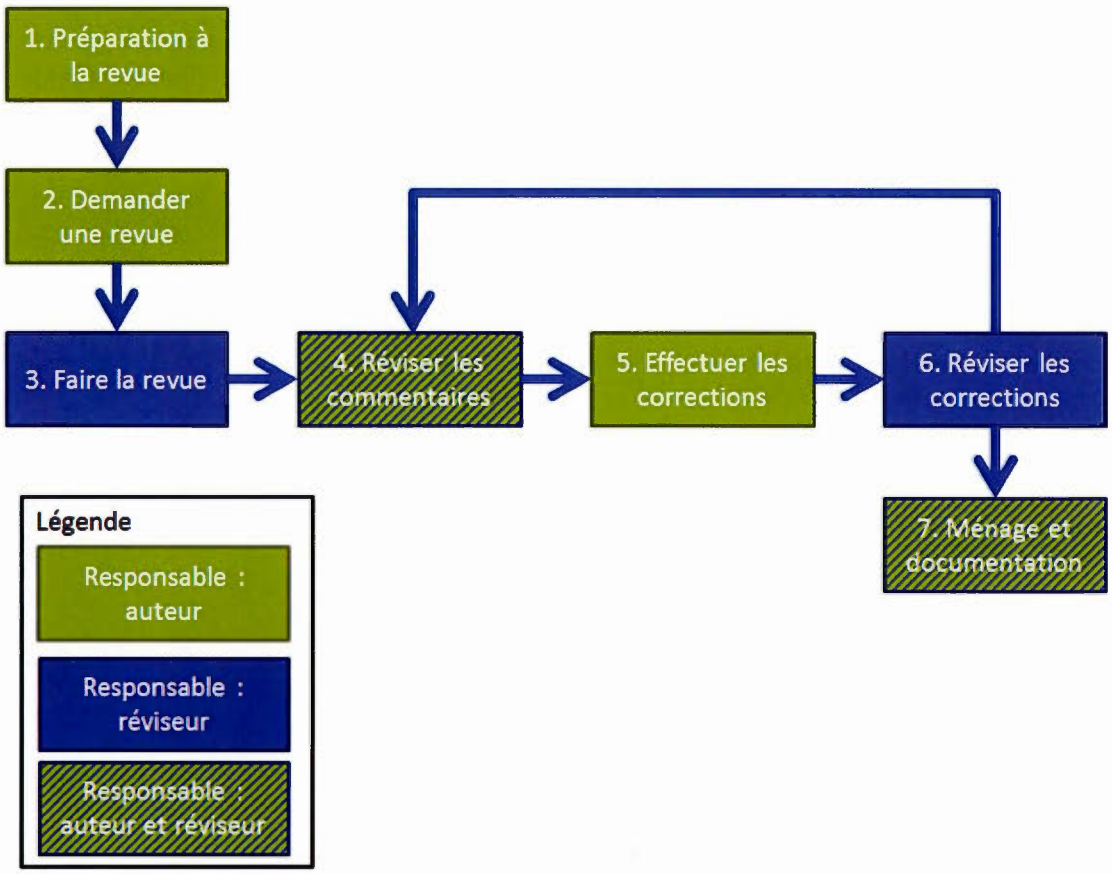


Figure 4.2 Étapes du processus de revue de code avec les responsables pour chacune des étapes

4.4.1 Préparation de la revue

Durant cette étape, l'auteur prépare le code à inspecter pour la revue. Cette préparation consiste à :

- inclure dans une branche du gestionnaire de code source les changements à inspecter
- consulter la liste de vérification, les outils d'analyse statique de code et les mesures décrites ci-dessus pour éliminer le maximum de défauts pouvant être détectés automatiquement avant même la soumission à l'inspection.

— S'assurer que le code respecte les normes minimales de qualité définies par l'équipe c'est-à-dire qu'il doit compiler, comprendre des tests unitaires automatisés vérifiant le code inclus dans la modification et ces tests doivent tous réussir.

C'est seulement une fois tous ces critères réunis que l'auteur peut demander une revue de code.

4.4.2 Demander une revue

La demande de revue se fait par l'outil de développement utilisé par l'équipe qui comprend déjà des fonctionnalités facilitant les revues par les pairs (voir la section 4.5.2). L'auteur démarre une demande en spécifiant la branche de code qu'il veut faire inspecter et en indiquant la liste des réviseurs qui seront affectés à l'inspection. Comme décrit précédemment, les réviseurs auront été préalablement désignés par le directeur responsable du projet. Les réviseurs seront informés par courriel ou par l'interface de leur outil de développement qu'une demande de révision leur a été assignée.

4.4.3 Faire la revue

Par la suite, les réviseurs s'attaquent à la révision comme telle. Cette révision se fait sur les fichiers inclus dans la demande de revue. Les éléments à inspecter ont déjà été détaillés précédemment. Lorsque des défauts sont découverts, les réviseurs les identifient directement dans l'outil de révision. Une boîte de texte leur permet de commenter et d'expliquer le problème. La revue se voulant un exercice le plus objectif possible, il est important de décrire non seulement le défaut, mais de faire référence à la règle non respectée à l'origine de ce défaut.

Une fois la révision terminée, le réviseur doit assigner un statut général sur la demande de revue :

— Acceptée;

— Acceptée avec suggestions;

- En attente de l'auteur. Ce statut est utilisé dans le cas où le réviseur ne peut compléter la revue puisque des explications supplémentaires de la part de l'auteur sont nécessaires;
- Refusée.

4.4.4 Réviser les commentaires

Lorsque le statut de la revue est assigné, l'auteur est automatiquement informé par courriel que la revue a été complétée et peut alors aller consulter les commentaires laissés par le ou les réviseurs. Généralement, la révision des commentaires se fera individuellement par l'auteur. Toutefois, si plusieurs défauts similaires ont été trouvés ou si des explications supplémentaires semblent nécessaires, le réviseur pourrait prendre l'initiative d'organiser une courte rencontre pour discuter des points problématiques, surtout s'il note qu'une notion ne semble pas bien comprise. Il est important que l'auteur comprenne la nature et la raison des changements demandés, non seulement pour corriger le code problématique dans la revue en cours, mais aussi pour pouvoir l'appliquer par la suite. Il se peut aussi que l'auteur ait une raison légitime d'avoir fait le code de cette façon dans quel cas, le réviseur pourra en tenir compte et par le fait même, découvrira peut-être une façon de faire qui lui était inconnue. La revue est un outil d'apprentissage, tant pour l'auteur que le réviseur. Ces discussions doivent être constructives et non une bataille d'égo entre deux façons de faire qui sont peut-être aussi bonnes une que l'autre. Si les deux techniques utilisées sont équivalentes au niveau fonctionnel et en qualité, la technique de l'auteur devrait être priorisée. En cas de doute, il peut être approprié de consulter d'autres collègues pour avoir leur opinion sur le sujet.

4.4.5 Effectuer les corrections nécessaires

L'auteur retourne alors dans son code pour faire les corrections nécessaires. Une fois celles-ci complétées, il n'a pas besoin de faire une nouvelle demande de revue, car tant que celle-ci n'est pas complétée, c'est-à-dire acceptée par tous les réviseurs, tous les changements dans la branche du code seront automatiquement inclus dans la demande de revue et chaque chan-

gement au code dans le gestionnaire de code source sera signalé au(x) réviseur(s). L'outil permet également à l'auteur d'inscrire un statut sur chaque demande de changement (corrigé ou non), facilitant le suivi des corrections.

4.4.6 Réviser les corrections

Le réviseur peut alors s'assurer que les corrections apportées sont satisfaisantes pour éliminer le défaut identifié. Le réviseur ne portera son attention que sur les demandes qui sont au statut « Corrigé », car si ce n'est pas le cas, c'est que l'auteur n'a peut-être pas complété toutes les corrections. Si le réviseur est d'accord avec le changement effectué, il changera le statut de la demande de changement à « Fermé ». Le processus se poursuit jusqu'à ce que toutes les demandes soient au statut fermé. À ce moment, le réviseur met la demande révision au statut accepté et la révision est terminée.

4.4.7 Ménage et documentation

La dernière étape du processus de révision est faite à la fois par l'auteur et le réviseur. Le travail de l'auteur est de faire le ménage dans le code source, c'est-à-dire intégrer le code qui a été accepté dans le tronc commun du code utilisé par tous les développeurs et la branche utilisée pour la révision est détruite puisqu'elle n'est plus utile.

Dans le cas des réviseurs, ceux-ci ont la responsabilité de maintenir au besoin la liste des défauts récurrents et des améliorations potentielles. Selon les cas trouvés, ils peuvent demander des ajustements à la liste de vérification ou les règles de l'outil d'analyse statique pour identifier plus facilement les cas similaires dans le futur. Ces suggestions devront être consignées dans l'outil de documentation de l'équipe (décrit à la section 4.5.7) et feront l'objet de discussions lors des rencontres de développeurs ou de rencontres particulières au besoin.

4.5 Choix et installation des outils pour soutenir la revue

Comme décrit dans la section 3.5.5, la mise en place du processus de revue demande l'évaluation et l'installation d'outils nécessaires pour supporter les différentes activités du processus. Les outils sont évalués selon différents critères :

Tableau 4.1 Critères d'évaluation des outils

Critère	Description	Valeurs possibles
Familiarité	L'équipe utilise ou est familière avec un outil permettant d'effectuer la tâche	— Oui — Non
Complétude	L'outil permet d'effectuer efficacement toutes les activités requises	— Oui — Non
Coût	Le coût d'acquisition, d'installation et d'utilisation de l'outil	Montant en $ US
Viabilité	Selon la réputation de l'entreprise et du produit, la probabilité que le produit existe et continue d'être maintenu durant les cinq prochaines années.	— Forte — Moyenne — Faible
Intégration	La facilité d'intégration de l'outil dans le processus de développement de l'équipe	— Automatique : Intégration faite automatiquement par le fournisseur du produit — Facile : Intégration possible à faire par l'équipe avec des connecteurs déjà existants — Moyenne : Intégration possible à faire par l'équipe, mais des connecteurs doivent être développés par l'équipe — Impossible : Aucune intégration possible

Voici ci-après une liste des outils choisis et du processus de sélection qui a mené l'équipe à ces choix.

4.5.1 Outil de gestion de code source

L'équipe utilise l'outil Team Foundation Server (TFS) de Microsoft depuis de nombreuses années. TFS, en plus d'être un gestionnaire de code source centralisé, offre un éventail de fonctionnalités telles que le suivi des items de travail (tâches, bogues, *product backlog items*) et un système de personnalisation des informations à inclure dans ces items de travail. Il supporte également l'intégration continue grâce à un système de compilation automatisé déclenché à chaque modification du code source ou selon un horaire prédéfini.

TFS permet aussi d'automatiser certaines tâches de tests unitaires et fonctionnels. Finalement, il intègre un module pour l'automatisation et l'approbation des déploiements d'applications dans les différents environnements utilisés durant le cycle de développement. Comme l'outil est connu de tous les membres de l'équipe et qu'il ne comporte pas de défauts importants, l'équipe ne voit pas l'intérêt de changer d'outil pour la gestion du code source. Elle a toutefois décidé de faire une mise à jour de TFS à la version la plus récente (TFS 2015 update 2 au moment de l'installation) puisque cette version apporte des nouveautés pour faciliter la gestion des branches en supportant le système de gestion de version Git. Elle offre également la possibilité d'intégrer une revue au processus de fusion (*merge*) des branches. Plus de détails sur cette fonctionnalité seront donnés dans la section suivante.

4.5.2 Outil pour demander et faire la revue de code

Avant le début de l'initiative, l'équipe n'utilisait pas d'outil spécifique pour la revue de code. Elle se faisait à l'aide de formulaire papier et toutes les communications se faisaient par courriel ou en personne. Les fonctionnalités attendues pour l'outil de revue sont :

- Permettre à l'auteur d'identifier un ensemble de changements (*commits* ou branche) à revoir.
- Permettre à l'auteur de désigner des réviseurs et de les informer qu'une revue est à faire.
- Permettre au réviseur de consulter facilement les changements effectués par l'auteur.
- Permettre au réviseur d'identifier les défauts et d'y inclure des commentaires.
- Permettre à l'auteur de visualiser les défauts à corriger.
- Permettre au réviseur d'approuver ou de refuser le changement.

Une recherche a été faite pour inventorier les outils spécialisés pour la revue de code dans l'environnement de développement Microsoft. Deux se démarquaient et ont été évalués : TFS de Microsoft (Microsoft 2017) et *Collaborator* de SmartBear (SmartBear 2017). Voici comment les deux outils se comparent selon les critères établis :

Tableau 4.2 Comparaison des outils TFS 2015 et SmartBear Collaborator

Critère / produit	Microsoft TFS 2015	SmartBear Collaborator
Familiarité	Oui	Non
Complétude	Oui	Oui
Coût	0 $ (l'équipe utilise déjà l'outil, le coût est donc déjà assumé)	1 500 $ US / an pour l'équipe
Viabilité	Forte	Moyenne
Intégration	Automatique	Facile

Compte tenu de cette analyse, il a été décidé d'utiliser TFS 2015 étant donné que le coût est déjà assumé et que l'intégration à la suite d'outils déjà utilisés par l'équipe est automatique. La fonctionnalité de revue de TFS 2015 n'est pas parfaite (certaines fonctionnalités comme l'identification des changements à réviser pourraient être plus flexibles), mais l'équipe considère que l'outil est suffisamment mature pour l'utilisation qu'elle compte en faire. De plus, comme l'outil évolue constamment, l'équipe demeurera à l'affût des nouveautés qui pourraient corriger certains des irritants rencontrés. La figure suivante démontre comment l'outil choisi permet de visualiser les changements dans un fichier et d'annoter le code à l'aide de commentaires pour indiquer et expliquer les défauts trouvés :

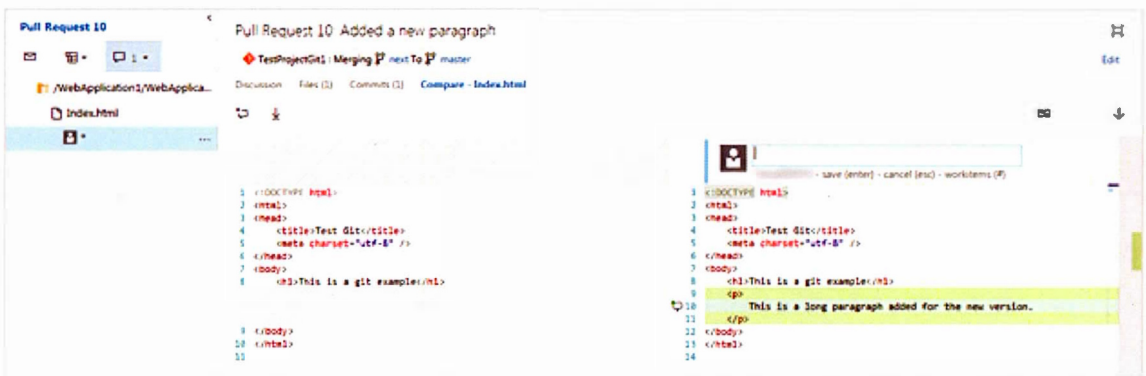


Figure 4.3 Outil de révision de changements de TFS 2015

4.5.3 Outil pour documenter et suivre l'évolution des défauts identifiés lors des revues

Ici aussi, la discussion a été de courte durée. TFS inclut déjà un module de documentation et de suivi des défauts à même l'outil de revue de code. L'équipe n'a donc pas vu le besoin d'évaluer d'autres outils alors que celui-ci faisait déjà très bien le travail, est efficace et permet de faire le suivi des corrections de défauts identifiés lors des revues. À noter que la documentation des défauts trouvés lors de la revue est beaucoup plus légère que celle des défauts trouvés suite aux revues, c'est-à-dire ceux qui sont passés entre les mailles du filet. Alors qu'un défaut découvert en revue est généralement documenté en une ou deux phrases expliquant la nature du défaut et la règle de validation qui a été violée, un défaut découvert suite à la revue sera beaucoup plus détaillé pour permettre à l'équipe d'en analyser plus facilement la source. La section 4.5.6 explique la documentation des défauts découverts après les revues.

4.5.4 Outil de mesure de la qualité du code

La qualité du code peut-être mesurée de différentes façons : la complexité cyclomatique par rapport à la taille de l'application, le couplage entre les classes, la cohésion, etc. L'équipe était à la recherche d'un outil permettant d'obtenir l'évolution de la qualité dans le temps afin de déterminer si les revues permettaient une amélioration mesurable de la qualité du code.
Trois outils ont été évalués pour répondre à ce besoin : Visual Studio 2015 de Microsoft, SonarQube de SonarSource et NDepend de Zen Program. Voici le résultat de l'évaluation des produits selon les critères établis :

Tableau 4.3 Comparaison des outils de mesure de la qualité du code.

Critère / produit	Microsoft Visual Studio 2015	SonarSource SonarQube	Zen Program NDepend
Familiarité	Oui	Non	Non
Complétude	Non	Oui	Oui
Coût	0 $ (l'équipe utilise déjà l'outil, le coût est donc déjà assumé)	0 $ (Open source)	363 $ US/an par développeur + 857 $ US/an pour le serveur d'intégration continu = 6302 $ US/an
Viabilité	Forte	Moyenne	Moyenne
Intégration	Facile	Facile	Moyenne

Après une analyse et une preuve de concept, l'outil retenu est SonarQube. C'est un outil de mesure de qualité très populaire dans les projets « open source », principalement dans le monde du développement Java. Il existe des connecteurs simples à configurer et à utiliser pour l'analyse du code C# et JavaScript, qui sont les deux langages principalement utilisés par l'équipe. De plus, SonarQube permet l'intégration des règles de validation de qualité de plusieurs outils sur le marché dont entre autre ReSharper, un outil de *refactoring* et d'analyse statique qui est déjà utilisé par les développeurs. Cette intégration permet aux développeurs d'utiliser localement sur leur poste de travail les mêmes règles de validations que sur le serveur d'intégration continue. Il n'y a donc pas de surprises, les règles sont connues de tous et affichées en temps réel dans l'environnement de développement intégré (IDE) du développeur.

Aussi, SonarQube permet des personnalisations intéressantes par exemple, des niveaux de qualité cible (*quality gates*) permettant à l'équipe de déterminer le niveau de qualité acceptable pour qu'une application puisse être déployée dans les environnements d'intégration et de production. Cette fonctionnalité et sa convivialité le démarquent clairement des deux autres outils à cet égard.

Il offre également plusieurs outils de visualisation pour aider à l'identification rapide des modules les plus problématiques de l'application selon différents critères (taille, complexité, niveau de dette technique). Par exemple, la Figure 4.4 permet de visualiser rapidement l'état actuel et l'évolution de la qualité du code pour un projet :

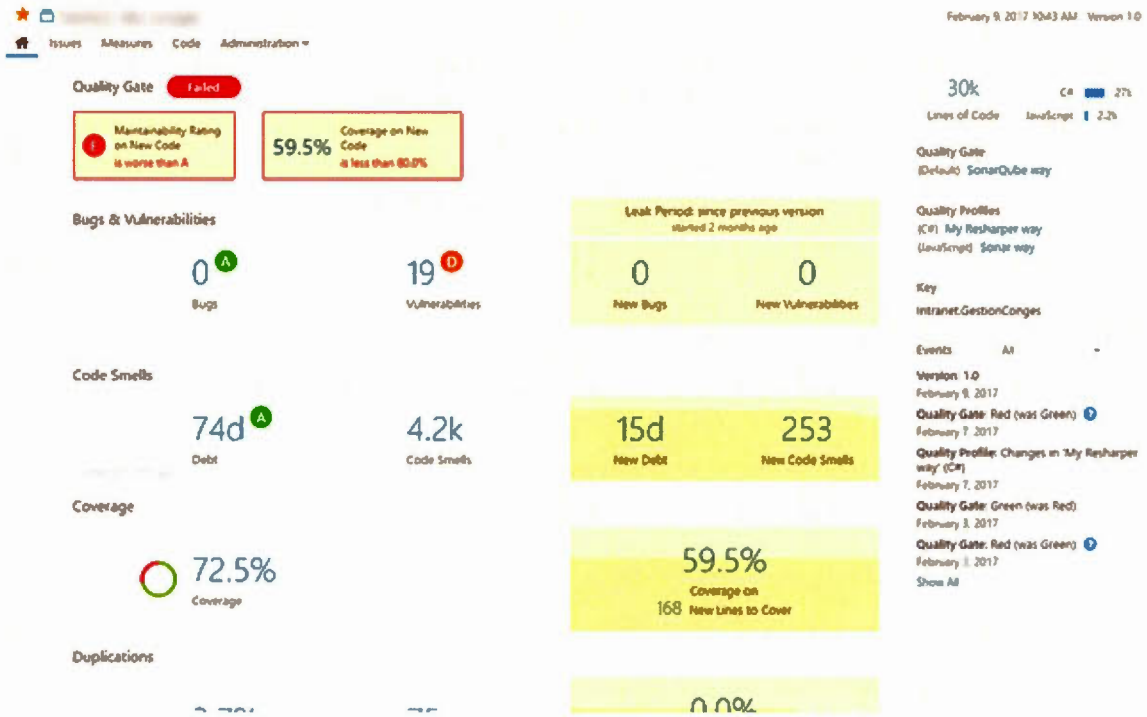


Figure 4.4 Aperçu de l'état actuel et de l'évolution de la qualité du code dans SonarQube

Un autre exemple d'outil de visualisation intéressant est une représentation graphique de la dette d'un fichier par rapport à sa taille. Cette façon de présenter ces mesures permet de repérer rapidement les classes les plus problématiques du projet, tant par leur taille que par leur faible niveau de qualité :

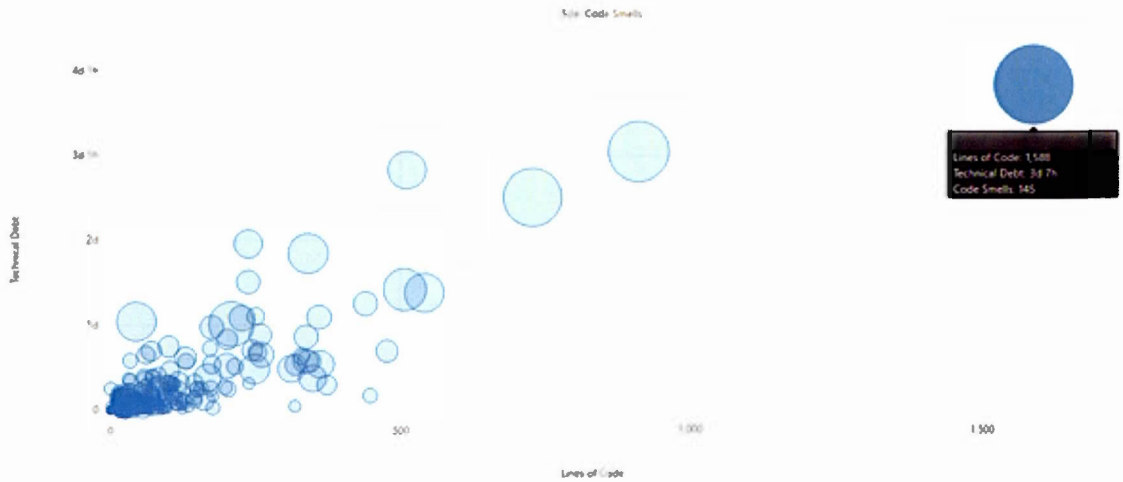


Figure 4.5 Visualisation de la dette technique d'une classe par rapport à sa taille dans SonarQube

Un des risques du produit est le fait qu'il ne soit supporté par aucune entreprise majeure, car c'est un logiciel libre. Toutefois, la communauté de développeurs et d'utilisateurs est importante et active donc cela amoindrit le risque de voir le produit disparaître ou de voir son évolution compromise.

Finalement, le produit est assez populaire pour que des connecteurs soient fournis par Microsoft pour intégrer SonarQube au processus d'intégration continue de TFS 2015, facilitant d'autant plus son utilisation.

4.5.5 Outil pour consigner le temps passé à effectuer des revues

La consignation du temps passé à faire de la revue sera faite par l'outil de feuille de temps, Kronos, déjà utilisé par l'entreprise. Une nouvelle activité, « Revue par les pairs », a été ajoutée et est incluse dans tous les projets. Les développeurs ont été conscientisés à y consigner le temps passé à effectuer de la revue de code. Ces données seront plus tard utilisées pour les mesures d'efficience du processus de revue.

4.5.6 Outil pour documenter et mesurer les défauts trouvés après la revue

Pour consigner, documenter les défauts résiduels suite à la revue et pouvoir mesurer le temps de *rework*, il a été décidé rapidement que l'outil nécessaire était déjà utilisé par l'équipe. En effet, comme elle utilise déjà TFS pour consigner les défauts et les tâches à faire, il était naturel de poursuivre avec cet outil. Toutefois, il a fallu utiliser les capacités de personnalisation de TFS pour définir de nouveaux attributs pour la documentation des défauts nécessitant un *rework*. Les attributs ajoutés à un item de travail de type bogue (qu'on utilise pour documenter les défauts) sont les suivants :

Tableau 4.4 Attributs ajoutés aux items de travail pour documenter les défauts.

Attribut	Description	Valeurs possibles
Étape de détection	Étape de la détection du défaut dans le processus de développement	• Exigences • Spécifications • Développement • Tests d'assurance qualité (QA) • Production
Source	Source du défaut	• Exigences • Spécifications • Développement
Version	Version de l'artéfact dans lequel le défaut a été trouvé	Numéro, identifiant ou itération de l'artéfact, selon la nature de l'artéfact

Une fois ces éléments ajoutés à la fiche de documentation des défauts, le formulaire a l'aspect illustré à la Figure 4.6.

Figure 4.6 Fiche de documentation des défauts de TFS 2015 adaptée aux besoins de l'équipe Web-Micro

De plus, comme l'item de travail comporte déjà un attribut pour consigner le temps qui a été nécessaire pour effectuer la correction, les participants ont simplement reçu l'instruction de cumuler le temps total de *rework* dans cet attribut. Comme le temps de *rework* comprend le temps nécessaire pour documenter, examiner, corriger et vérifier la correction du défaut (arrondi aux 15 minutes près par souci d'efficacité et de simplicité), l'exemple du Tableau 4.5 a été fourni à l'équipe.

Tableau 4.5 Exemple d'effort de *rework* consigné par item de travail.

Activité	Responsable	Temps requis	Temps cumulé
Documenter le défaut (création de l'item de travail)	Personne qui a découvert le défaut	10 minutes (arrondi aux 15 minutes)	0,25 h
Rechercher la cause du défaut	Développeur ou analyse à qui le défaut a été assigné	30 minutes	0,75 h
Correction du défaut (incluant, si nécessaire, les tests unitaires, l'intégration et la documentation)	Développeur ou analyse à qui le défaut a été assigné	1 heure	1,75 h
Vérification de la correction	Personne qui a découvert le défaut	30 minutes	2,25 h
Temps total de *rework*			**2,25 h**

Si la correction n'est pas satisfaisante pour la personne ayant identifié le défaut, l'effort de *rework* se poursuit (les nouvelles étapes sont identifiées par une mise en forme *italique* dans le Tableau 4.6) :

Tableau 4.6 Exemple d'effort de *rework* consigné avec des corrections insatisfaisantes.

Activité	Responsable	Temps requis	Temps cumulé
Documenter le défaut (création de l'item de travail)	Personne qui a découvert le défaut	10 minutes (arrondi aux 15 minutes)	0,25 h
Rechercher la cause du défaut	Développeur ou analyse à qui le défaut a été assigné	30 minutes	0,75 h
Correction du défaut (incluant, si nécessaire, les tests unitaires, l'intégration et la documentation)	Développeur ou analyse à qui le défaut a été assigné	1 heure	1,75 h
Vérification de la correction	Personne qui a découvert le défaut	30 minutes	2,25 h
Documenter le résultat du test et la nature du défaut résiduel	*Personne qui a découvert le défaut*	*15 minutes*	*2,50 h*
2ième correction	*Développeur ou analyse à qui le défaut a été assigné*	*30 minutes*	*3,00 h*
Vérification de la 2ième correction	*Personne qui a découvert le défaut*	*30 minutes*	*3,50 h*
Temps total de *rework*			**3,50 h**

En le faisant de cette façon, l'équipe juge que la mise en place de la méthode de cumul des heures devrait se faire avec plus de facilité étant donné qu'elle ne sera pas additionnée de l'apprentissage d'un nouvel outil.

4.5.7 Outil pour documenter les défauts récurrents et les améliorations potentielles au processus de revue

Comme décrit dans le processus de revue de code, une des responsabilités des réviseurs est de documenter les cas récurrents et les améliorations au processus. Ces suggestions pourront alors servir lors des révisions périodiques du processus de revue, des listes de vérification ou à la calibration des règles de validations dans les outils d'analyse du code.

Pour documenter ces informations, l'équipe a décidé d'utiliser l'outil de gestion documentaire déjà en place dans l'entreprise, Microsoft Sharepoint. Une liste de discussion a été créée spécifiquement pour cette activité et les participants aux revues sont encouragés à y émettre leurs commentaires et à en discuter. Le contenu de ces discussions sera utilisé lors des rencontres d'équipe mensuelles où les décisions sur les priorités des améliorations à mettre en place seront prises.

4.6 Formation des participants

Pour s'assurer que tous les participants aux revues sont au courant du processus et qu'ils en comprennent les subtilités, des sessions de formation ont été données. Les sessions de formation données à l'interne par l'architecte se sont déroulées en plusieurs étapes afin de donner l'information pertinente à chaque participant selon son rôle dans l'équipe. Voici un aperçu des formations qui ont été données dans le cadre de l'initiative :

- Les développeurs ont d'abord été formés sur le processus de revue lors d'une rencontre d'équipe d'environ une heure. La formation donnait de l'information sur les concepts du processus de revue ainsi que sur la partie technique de la revue c'est-à-dire l'utilisation des outils nécessaires à la revue. Cette session de formation a été donnée à tous les développeurs de l'équipe en même temps dans le cadre de la rencontre mensuelle des développeurs. Toutefois, comme il est attendu que tous les développeurs ne seront peut-être pas tous impliqués dans des revues de code durant la durée de l'initiative, des sessions de rappel pourront être données plus tard pour ceux

qui auraient oublié certains concepts ou techniques. De plus, ces sessions pourront permettre de partager l'expérience des développeurs qui ont utilisé le processus de revue aux autres.

- Par la suite, les développeurs ont été formés en petits groupes de trois à quatre personnes sur le nouveau gestionnaire de code source de TFS 2015, Git. Les développeurs sont déjà à l'aise avec TFS, mais le changement de gestionnaire de code source change plusieurs de leurs habitudes de travail. Il semblait donc important de s'assurer que tout le monde comprend bien l'outil. Ces formations ont été aussi l'occasion de répondre aux questions et craintes de certains des membres de l'équipe. De plus, certaines des questions ont permis d'identifier des améliorations à faire dans la documentation et la configuration de l'outil.
- Une fois les développeurs formés, ce fut au tour des analystes. Ils ont également été formés sur le processus de revue en général afin d'être au courant de ce que font les développeurs durant les revues. Pour les analystes, l'accent a été mis sur le suivi et la documentation des défauts puisque c'est la partie du processus qui risque de les impacter le plus dans leurs tâches quotidiennes. De plus, comme pour les développeurs, ils ont reçu une courte formation sur les nouveautés de TFS, en particulier les outils de tests, de documentation des défauts et de mesure du temps de *rework*.

CHAPITRE 5

RÉSULTATS

Durant l'initiative, un ensemble de mesures ont été prises sur 11 projets réalisés par l'équipe Web-Micro. Ces projets ont été réalisés dans la période de janvier à juillet 2017 et représentent un échantillon varié par leur taille et leur contexte d'affaires (Internet, intranet). Les prochaines sections serviront à présenter les résultats, à les analyser et à les mettre en contexte pour tenter d'en tirer des conclusions. Par la suite, les résultats du sondage de satisfaction envoyé aux développeurs seront présentés et décortiqués. Finalement, la section 5.4 présentera les améliorations qui ont été faites en cours de projet suite aux premières mesures.

5.1 Mesures sur les revues

Pour déterminer les progrès de l'équipe dans le processus de revue par les pairs, différentes mesures ont été prises. Après quelques semaines de mise en route où les membres étaient appelés à se familiariser avec le processus de revue, des données ont été recueillies à la fin des semaines du 8 mai, 2 juin, 9 juin et 26 juin. Ce sont les données de cette dernière prise de mesure qui seront présentées.

Aux fins de présentation, les projets ont été classés selon leur coût relatif par PFC, du moins coûteux au plus coûteux. Cependant, comme ces projets ont été démarrés ou complétés avant la mise en place du plan de mesure, certaines des valeurs sont manquantes, car l'information requise n'était pas disponible à ce moment. Lorsque c'était possible, les valeurs ont été estimées avec l'information disponible et elles sont identifiées par un trait de soulignement dans le tableau des résultats. La méthode d'estimation utilisée sera décrite dans l'analyse des données. Aussi, certaines mesures recueillies ne semblaient pas correspondre à la réalité vécue

durant les projets, jetant un doute sur la fiabilité de certains résultats obtenus. Pour indiquer d'une mesure ne semble pas conforme à la réalité, elle a été inscrite avec une police de caractère en *italique*. Les projets ont été cotés selon un indice de fiabilité tel que détaillé dans le Tableau 5.1. Cet indice est basé sur le nombre de mesures manquantes qui ont dû être estimées ou qui sont suspectes :

Tableau 5.1 Critères pour l'indice de fiabilité des mesures

Indice de fiabilité des mesures	Critère
A	L'ensemble des mesures sont disponibles et semblent fiables
B	Une des mesures a été estimée ou sa valeur est considérée comme suspecte
C	Deux des mesures ont été estimées ou leurs valeurs sont considérées comme suspectes
D	Plus de deux des mesures ont été estimées ou leurs valeurs sont considérées comme suspectes. Le nombre de défauts étant une mesure très importante pour l'analyse, tout projet où la valeur était manquante ou suspecte s'est vu attribuer automatiquement la cote de fiabilité D.

Aussi, pour clarifier le vocabulaire utilisé dans les résultats, les défauts trouvés lors des revues utiliseront le terme *problème potentiel (PP)* tandis que le terme *défaut* sera réservé aux défauts détectés en dehors du processus de revue (par exemple, lors du QA ou en production).

Les mesures ont été regroupées en six catégories, contenant chacune certaines mesures de base et certaines mesures dérivées. Voici une description de ces catégories et des mesures qu'elles contiennent :

- Informations sur les projets : informations génériques sur les projets comme le code de projet, le type, la taille de projet et l'indice fiabilité des mesures. Quant à la taille des projets, elle a été déterminée par le nombre de PFC selon les critères suivants :
    - 0 à 50 PFC : petit

- 51 à 200 PFC : moyen
- 201 PFC et plus : gros

— Taille et effort : données sur la taille (en PFC) des projets et sur l'effort réel de réalisation pour le projet. Le coût relatif du projet (en heures/PFC) est dérivé du nombre d'heures totales de réalisation sur la taille du projet.

— Défauts détectés : Décompte des défauts trouvés. On considère comme un défaut tout problème trouvé dans le logiciel qui n'a pas été détecté par le processus de revue. Autrement dit, il s'agit de tout ce qui est passé au travers des mailles du filet. La densité de défaut de l'application est dérivée du nombre de défauts relatif à la taille de l'application (en PFC). Une caractérisation des défauts (phase d'injection et de détection) a également été faite et elle sera traitée séparément dans une section subséquente.

— Données sur les revues : ici, plusieurs mesures ont été recueillies ou dérivées.

- Heures de revues : Nombre d'heures de revue réalisées dans le cadre du projet.
- Problèmes potentiels détectés : Nombre de problèmes potentiels identifiés par les réviseurs lors des différentes revues.
- Efficacité : proportion des problèmes potentiels trouvés en revue relative à la somme des problèmes potentiels et des défauts détectés. Cette mesure permet de déterminer si le processus de revue permet de déceler une proportion importante des défauts qui se seraient retrouvés dans le logiciel sans les revues.
- Problèmes potentiels par heure de revue : ratio permettant de déterminer le taux de détection des problèmes potentiels par rapport à l'effort.
- Problèmes potentiels par PFC : ratio permettant de déterminer la densité de problèmes potentiels détectés par rapport à la taille de l'application.
- Heures de revue par PFC : ratio permettant de déterminer l'effort relatif de revue par rapport à la taille de l'application.
- Proportion de l'effort de revue par rapport à l'effort total de réalisation : ratio permettant de déterminer l'effort relatif de revue par rapport au temps de réalisation total pour le projet.

- Données de *rework* : informations relatives à l'effort de *rework*
    - Heures de *rework* : nombre d'heures passées en *rework* c'est-à-dire à corriger des défauts qui n'ont pas été préalablement détectés en revue.
    - Proportion de l'effort de *rework* sur l'effort total de projet
    - Heure de *rework* par PFC : ratio permettant de déterminer l'effort de rework par rapport à la taille du projet
    - Temps moyen de correction des défauts : aussi appelé en anglais *Mean Time To Repair* (MTTR), il s'agit d'une mesure dérivée qui consiste à déterminer combien de temps est nécessaire pour corriger un défaut en divisant le temps de *rework* par le nombre de défauts corrigés.
- Qualité : information recueillie grâce à l'outil SonarQube. L'outil donne trois indices de qualité du code soit la sécurité, la fiabilité et la maintenabilité. Ces trois indices peuvent prendre les valeurs A à E. Une fois que l'analyseur a parcouru le code et inventorié les problèmes potentiels, ceux-ci sont caractérisés selon un type (sécurité, fiabilité, maintenabilité) et une sévérité (mineur, majeur, critique, bloquant). C'est à partir des problèmes caractérisés que sont calculés les indices suivants :

Tableau 5.2 Critères pour l'indice de sécurité du code

Indice de sécurité du code	Critère
A	Aucun problème de sécurité détectée
B	Au moins un problème de sécurité mineur détecté
C	Au moins un problème de sécurité majeur détecté
D	Au moins un problème de sécurité critique détecté
E	Au moins un problème de sécurité bloquant détecté

Tableau 5.3 Critères pour l'indice de fiabilité du code

Indice de fiabilité du code	Critère
A	Aucun problème de fiabilité détecté
B	Au moins un problème de fiabilité mineur détecté
C	Au moins un problème de fiabilité majeur détecté
D	Au moins un problème de fiabilité critique détecté
E	Au moins un problème de fiabilité bloquant détecté

Pour le calcul de la dette technique, SonarQube utilise une méthode de calcul inspirée de la méthode SQALE (Letouzey 2016) qui quantifie la dette technique en calculant l'effort de correction des problèmes potentiels identifiés par rapport à l'effort nécessaire pour une réécriture complète de l'application ou du module contenant le code modifié. Ainsi, une dette technique de 5 % indique que l'effort de correction représente 1/20 de l'effort d'une réécriture complète.

Tableau 5.4 Critères pour l'indice de maintenabilité du code

Indice de maintenabilité du code	Critère
A	Indice de dette technique inférieur à 5 %
B	Indice de dette technique entre 5 % et 10 %
C	Indice de dette technique entre 10 % et 20 %
D	Indice de dette technique entre 20 % et 50 %
E	Indice de dette technique supérieur à 50 %

Pour le tableau de mesures général, les trois indices ont été additionnés (A=1, B=2, C=3, D=4, E=5) pour obtenir une valeur générale de qualité allant de 3 (meilleure qualité) à 15 (plus faible qualité).

Tableau 5.5 Données des mesures de projet

Code	Info Projet			Taille et efforts			Défauts		Revues						Rework				
	Fia.	Type	Size	PFC	HT	HT / PFC	Nb	Nb / PFC	HDR	PP	Eff.	PP/ HDR	PP/ PFC	Vit. Revue	% HDR/ HT	HRw	%HRw / HT	HRw/ PFC	MTTR	QS
A	B	A	G	531	2133	4.02	112	0.21	15	56	33.33%	3.73	0.11	35.40	0.70%	337	15.80%	0.63	3.01	N/A
B	C	B	M	52	253	4.87	18	0.35	2.5	45	71.43%	18.00	0.87	20.80	0.99%	31.00	12.25%	0.60	1.72	3
C	A	A	M	106	521	4.92	22	0.21	14	140	86.42%	10.00	1.32	7.57	2.69%	29.75	5.71%	0.28	1.35	4
D	C	B	P	16	86.25	5.39	4	0.25	0.75	6	60.00%	8.00	0.38	21.33	0.87%	22.75	26.38%	1.42	5.69	10
E	C	B	P	42	227.5	5.42	34	0.81	0	0	0.00%	0.00	0.00	0.00	0.00%	20.71	9.10%	0.49	0.61	9
F	B	B	P	40	222.5	5.56	8	0.20	0	0	0.00%	0.00	0.00	0.00	0.00%	11.75	5.28%	0.29	1.47	N/A
G	D	B	M	85	641.58	7.55	17	0.20	0	0	0.00%	0.00	0.00	0.00	0.00%	96.99	15.12%	1.14	5.71	10
H	D	B	G	455	3854.5	8.47	80	0.18	21	4	4.76%	0.19	0.01	21.67	0.54%	530.66	13.77%	1.17	6.63	10
I	D	B	G	258	2514.3	9.75	140	0.54	31.5	19	11.95%	0.60	0.07	8.19	1.25%	163.45	6.50%	0.63	1.17	4
J	C	B	P	29	396.25	13.66	18	0.62	6.25	0	0.00%	0.00	0.00	4.64	1.58%	51.60	13.02%	1.78	2.87	N/A
K	C	B	M	81	1385.6	17.11	76	0.94	10	61	44.53%	6.10	0.75	8.10	0.72%	232.00	16.74%	2.86	3.05	8
Total	N/A	N/A	N/A	1695	12235	N/A	529	N/A	101	331	N/A	N/A	N/A	N/A	N/A	1528	N/A	N/A	N/A	N/A
Moy.	N/A	N/A	N/A	154	1112.3	7.88	48.09	0.41	9.18	30	28.40%	4.24	0.32	11.61	0.85%	138.9	12.70%	1.03	3.02	7.25

Définitions et acronymes

- Fia. : Côte de fiabilité des mesures (de A à D)
- Type : Type de projet (A = Internet, B = Intranet)
- Size : Taille du projet (P = Petite, M = Moyen, G = Gros)
- PFC : Point de fonction COSMIC
- HT : Heures totales
- Nb : Nombre
- HDR : Heures de revues
- PP : Problèmes potentiels
- Eff. : Efficacité de la revue. Proportion des PP sur le total des défauts et des PP potentiels (PP/(Défauts+PP))
- Vit. Revue : Vitesse de revue. Taille du projet (en PFC) sur le nombre d'heures de revue (PFC/HDR)
- HRw : Heures de *rework*
- MTTR : Temps moyen de correction d'un défaut (Mean Time To Repair)
- QS : Indice de qualité (quality score).
- Moy. : Moyenne
- N/A : Non disponible ou ne s'applique pas

Les défauts détectés ont également été caractérisés par phase d'injection et de détection pour pouvoir juger de l'efficacité du processus de revue actuel et aussi déterminer les phases du processus de développement où des améliorations seraient les plus bénéfiques pour l'amélioration de la qualité des logiciels. Voici le détail des caractérisations :

Tableau 5.6 Caractérisation des défauts

		Défauts détectés par phase						Défauts injectés par phase			
Code	Total	Req	Spec	Dev	QA	Prod	N/D	Req	Spec	Dev	N/D
A	112	0	0	1	111	0	0	0	30	82	0
B	18	0	0	1	13	0	4	3	0	11	4
C	22	0	0	0	22	0	0	0	9	13	0
D	8	0	0	0	8	0	0	0	0	8	0
E	4	0	0	0	0	0	4	0	0	0	4
F	17	0	0	0	0	0	17	0	0	0	17
G	34	0	0	0	0	0	34	0	0	0	34
H	80	0	0	0	0	0	80	0	0	0	80
I	140	0	0	2	44	0	94	1	7	38	94
J	18	0	0	0	18	0	0	4	1	13	0
K	76	0	0	0	0	0	76	0	0	0	76
Totaux	**529**	**0**	**0**	**4**	**216**	**0**	**309**	**8**	**47**	**165**	**309**

Définitions et acronymes

Code : Code de projet

Total : Total des défauts détectés

Req : Exigences (requirements)

Spec : Spécifications

Dev : Développement

QA : Tests (Quality Assurance)

Prod : Production

N/D : Information non disponible

5.2 Données de revues analysées

5.2.1 Mesures sur la taille et l'effort

La fiabilité des mesures d'effort de réalisation est généralement bonne, car l'information provient des feuilles de temps remplis par tous les membres de l'équipe. Ce sont ces données qui sont utilisées par les gestionnaires pour suivre l'avancement des projets et facturer les clients. Par contre, pour ce qui est de la mesure de l'effort estimé, la situation est un peu moins claire.

Depuis quelques années, l'équipe utilise la mesure de taille fonctionnelle en point de fonction COSMIC pour estimer les projets. Selon le modèle d'estimation de l'équipe, le coût estimé d'un point de fonction est de 5,4 heures par PFC. Cependant, il s'est avéré que plusieurs estimations de projet rataient leur cible, et parfois avec plus de 50 % d'écart. L'équipe cherchant à identifier le problème a mené de front deux initiatives pour tenter de remédier à la situation :

- Les projets de type Internet de grande taille sont actuellement estimés grâce à la méthode du *poker planning*. Il s'agit d'un essai fait par l'équipe pour vérifier si cette méthode serait plus précise pour estimer correctement ce type de projet. Dans la liste des projets de l'initiative, le projet A a été estimé de cette façon. Pour rendre la base de calcul plus homogène entre les projets, l'estimation en heure déterminée lors du des séances de *poker planning* a été ramenée en PFC. Jusqu'à présent, la mesure semble assez juste, car le coût en heure ramené en PFC équivaut à 4,01 h/PFC soit un peu en bas du coût habituel des projets, mais pas par une marge assez importante pour être écarté.
- Pour les projets utilisant la méthode d'estimation COSMIC, l'équipe a constaté des écarts dans la façon de mesurer la taille fonctionnelle. Ces écarts ont plusieurs sources dont :
    - Une interprétation différente du découpage des FUR et des groupes de données d'un mesureur à l'autre.

- Le manque de rigueur dans la mise à jour des mesures de taille fonctionnelle suite à des demandes de changements du client en cours de projet.

Pour remédier à la situation, l'équipe s'est lancée dans une révision de ses pratiques de mesures COSMIC pour assurer une meilleure cohérence lorsque vient le temps d'estimer la taille des projets. Cependant, comme certains projets ont été estimés avant cette révision, certaines des mesures de taille semblent aujourd'hui sous-estimées compte tenu de l'historique des projets et des efforts réels de réalisation. Dans le cas des projets de l'initiative, les projets I et K font partie de cette catégorie. La révision de la méthode d'estimation sera traitée en détail dans la section 5.4.2.

Finalement, l'équipe a pris l'habitude dans les dernières années d'estimer les très petits projets (20 jours et moins) en heure plutôt qu'en PFC, car la méthode COSMIC avait tendance à surestimer le temps de réalisation des très petits projets. Toutefois, pour les besoins de l'initiative, les petits projets (D, E et F) ont été ramenés en PFC. De toute façon, le coût final de réalisation étant plus grand que l'estimation COSMIC de base de 5,4h/PFC, on ne peut pas affirmer que ces projets ont été surestimés. L'équipe a donc décidé d'inclure aussi les petits projets dans la réévaluation de la méthode de mesure COSMIC puisqu'elle n'est plus certaine qu'ils sont réellement surestimés.

La plupart des projets qui ont obtenu une cote de fiabilité de B et C l'ont été à cause des mesures de taille fonctionnelle. Toutefois, la meilleure estimation possible a été faite pour obtenir des mesures acceptables pour l'analyse.

5.2.2 Défauts détectés et caractérisation des défauts

Les défauts détectés représentent les défauts trouvés lors des tests ou en production qui n'ont pas été identifiés durant les revues. Le nombre de défauts provient de l'outil TFS qui sert à les consigner, comme décrit à la section 4.5.6. Dans plusieurs cas, les informations sont fiables et complètes. Quelques exceptions cependant :

- Pour les projets débutés avant la mise en place du processus de revues, certains (ou la totalité) défauts n'ont pas été caractérisés, car ils avaient été ouverts depuis trop long-

temps et les gens concernés ne se souvenaient pas nécessairement de la phase de détection du défaut. La phase d'injection quant à elle pouvait parfois être difficile à déterminer plusieurs mois après sa documentation initiale. De plus, l'effort de tout caractériser à nouveau étant important, il a été décidé de ne pas réévaluer les défauts existants, mais de caractériser seulement les nouveaux défauts. Cela explique que plusieurs phases de détection et d'injection sont non déterminées.

— Pour trois des projets (G, H et I), le nombre de défauts enregistrés dans l'outil de suivi semblait anormalement bas compte tenu de la taille et de l'historique des projets concernés. Après discussion avec les gens impliqués, il a été découvert que dans certains projets, les défauts n'étaient pas tous consignés dans l'outil, mais avaient plutôt été gérés « manuellement » par courriel ou de vive voix. Pour tenter d'obtenir la meilleure mesure possible, le nombre de défauts a été estimé à partir des données de feuille de temps et de l'historique des *commits* dans l'outil de gestion de code source. Cette méthode d'estimation comportant une marge d'erreur assez grande, ces projets se sont automatiquement vus attribuer une cote de fiabilité de D.

Maintenant, que peut-on déduire des données sur les défauts? Quelques informations intéressantes peuvent en être tirées. Tout d'abord, la densité de défauts semble être un moyen efficace de prédire le coût par PFC d'un projet. En effet, une augmentation de la densité de défauts semble entrainer une augmentation du coût relatif par PFC. Si l'on prend tous les projets analysés durant l'initiative, la relation n'est pas évidente à première vue :

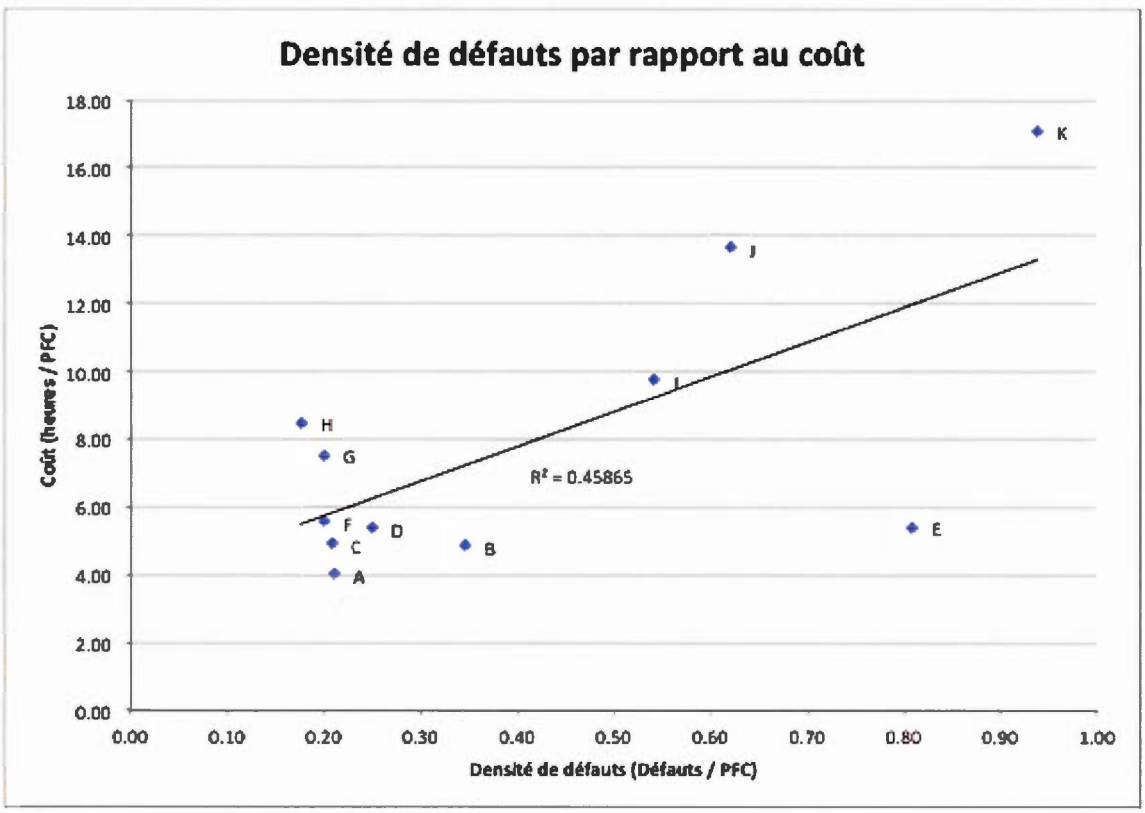


Figure 5.1 Densité de défauts par rapport au coût (tous les projets)

Par contre, si l'on retire le projet E du graphique (qui semble être un cas extrême « *outlier* »), on peut trouver une relation assez importante entre la densité de défaut et le coût du projet :

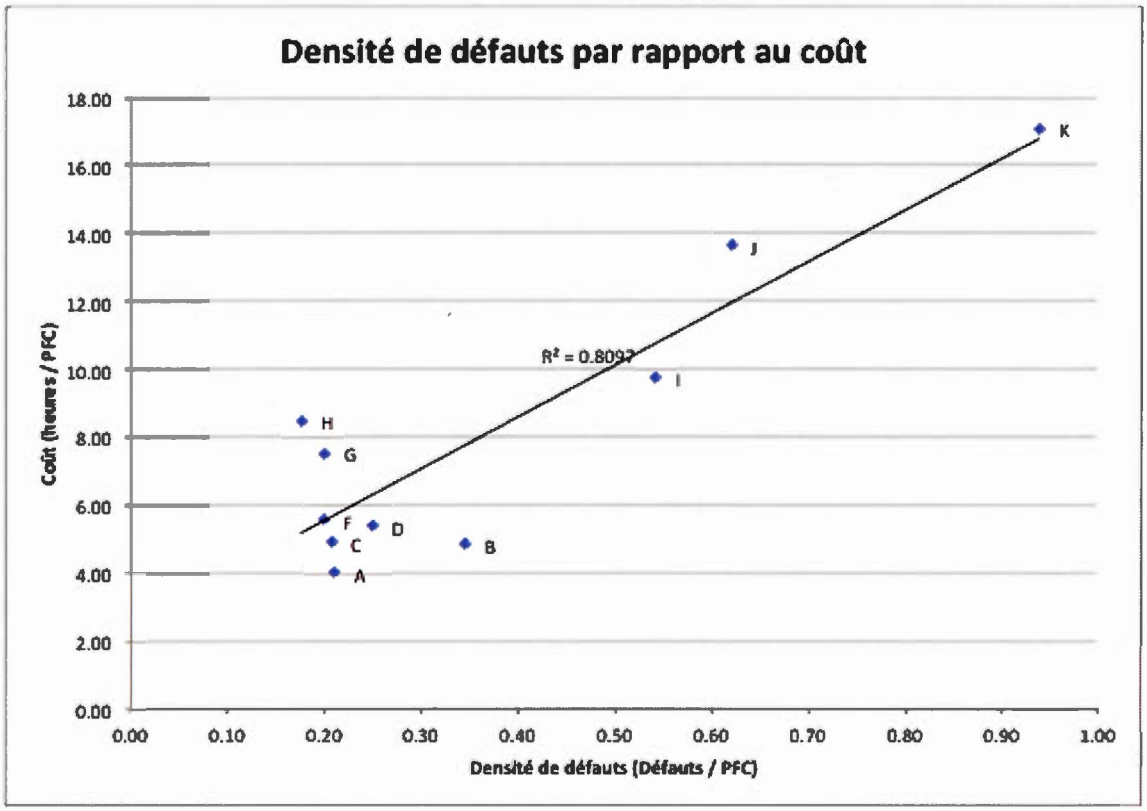


Figure 5.2 Densité de défauts par rapport au coût (projet E exclu)

La relation est encore plus importante lorsqu'on exclut les projets dont la fiabilité des données est faible (D) :

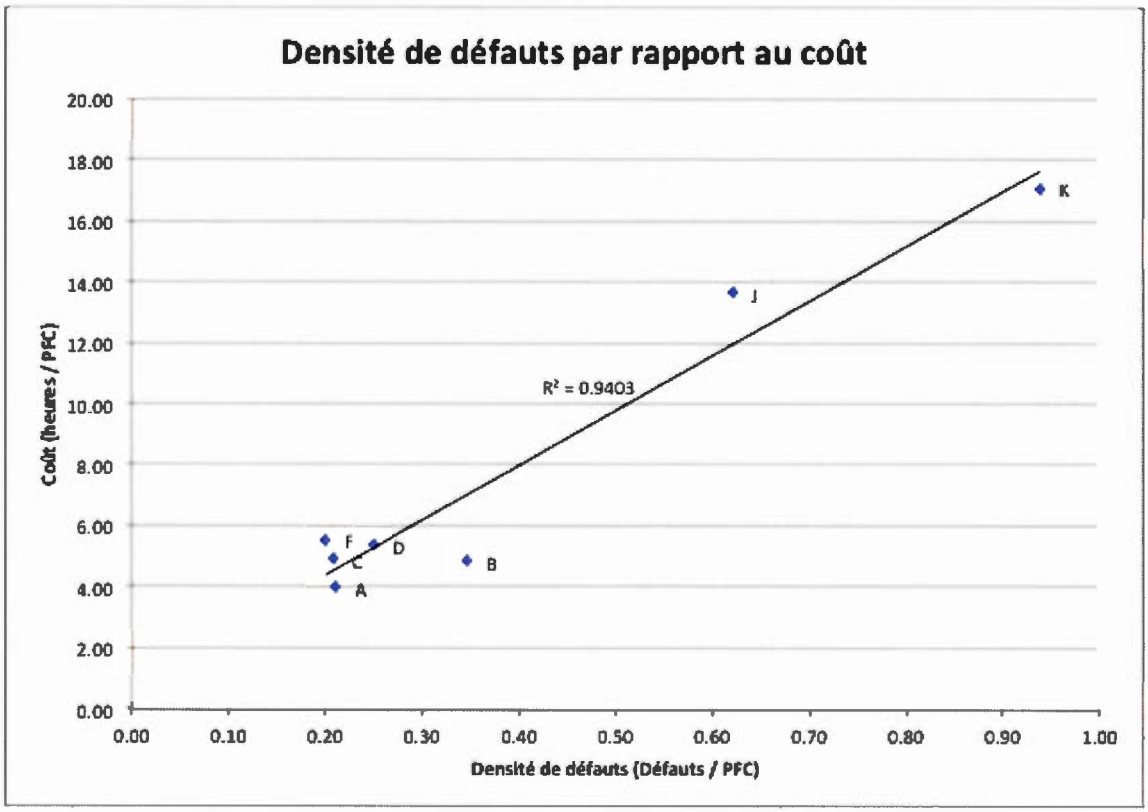


Figure 5.3 Densité de défauts par rapport au coût (faible fiabilité et projet E exclu)

Ces données viennent conforter l'idée que la qualité externe du code (c'est-à-dire une faible densité de défauts lors des tests et de la production) contribue à garder le coût par PFC bas et ainsi permettre de livrer un projet à moindre coût et donc, dans un délai plus court.

La caractérisation des défauts quant à elle a permis d'obtenir les informations suivantes :

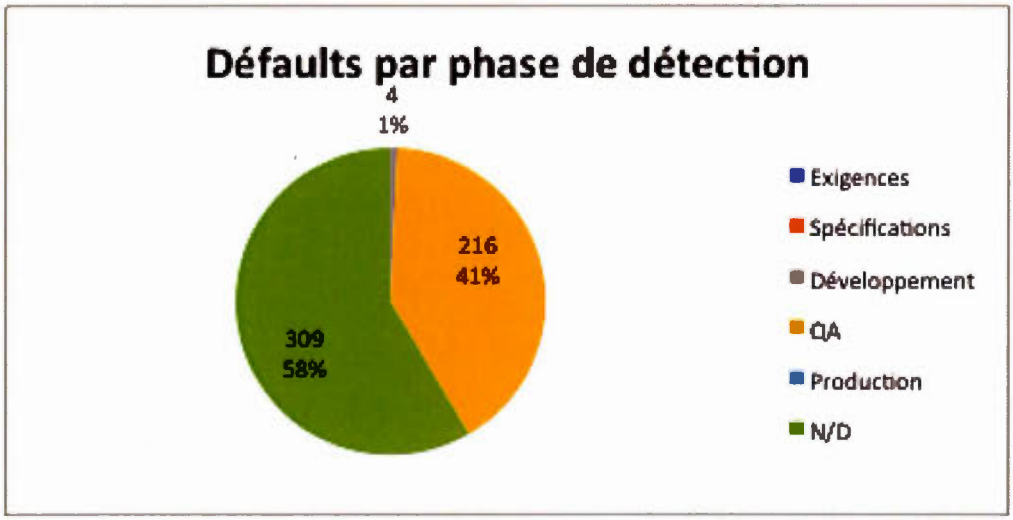


Figure 5.4 Défaut par phase de détection

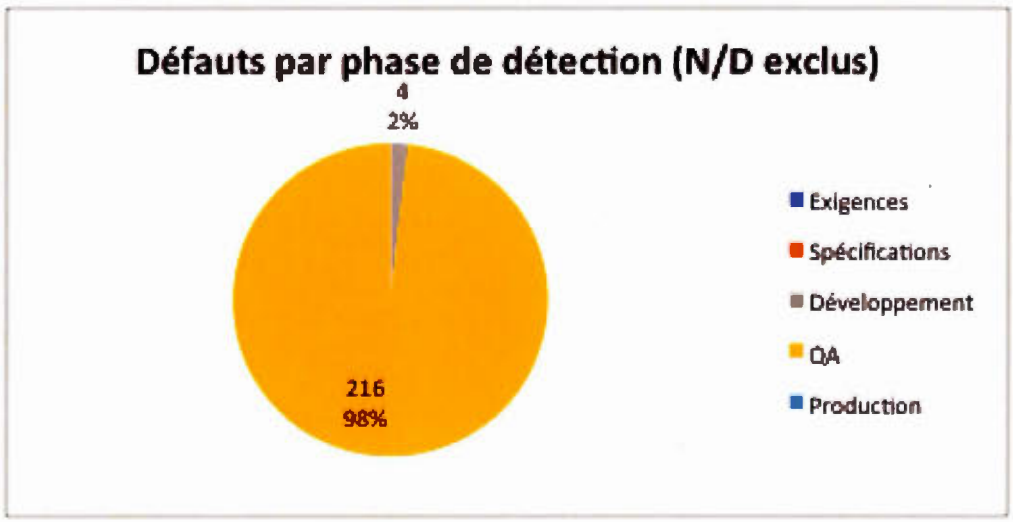


Figure 5.5 Défauts par phase de détection (non déterminés exclus)

Les figures précédentes démontrent de façon éloquente que la très grande majorité des défauts sont trouvés lors de la phase de QA. Ces résultats sont conformes aux attentes.

Par contre, l'analyse de la phase d'injection des défauts donne des résultats plus inattendus :

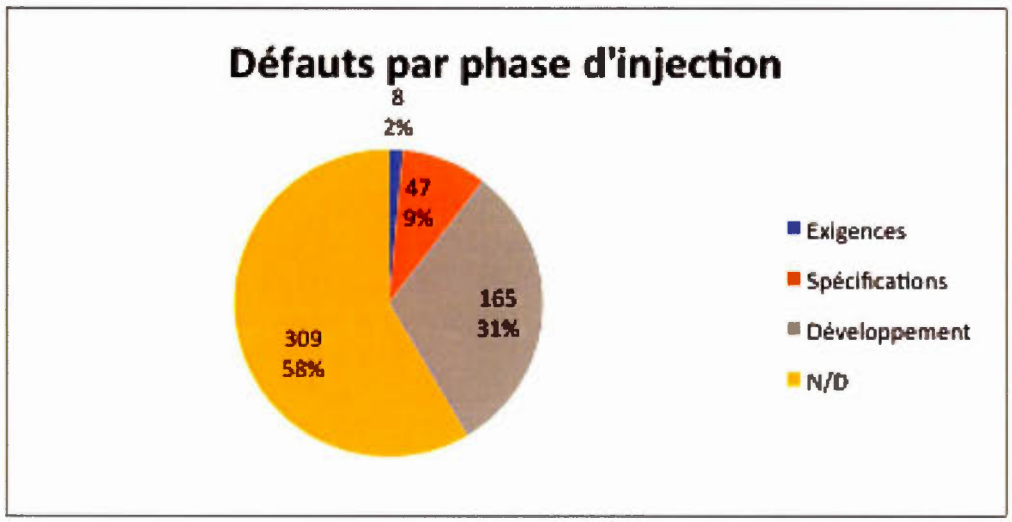


Figure 5.6 Défauts par phase d'injection

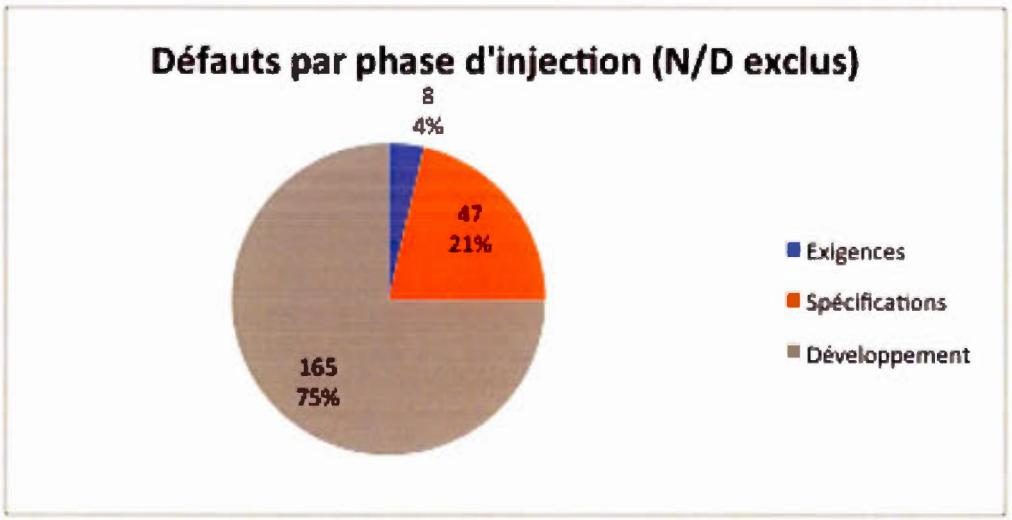


Figure 5.7 Défauts par phase d'injection (non déterminé exclus)

Bien que la majorité des défauts proviennent du développement, un nombre significatif des défauts trouve leur source dans les spécifications et les exigences. Bien qu'exclues en début de projet, ces données apportent un argument favorable à la mise en place de revues des spécifications. Cet aspect sera traité dans les travaux futurs (section 7.2).

5.2.3 Mesures des revues

Les mesures de revues ne démontrent pas de tendances claires. En effet, les données recueillies ne permettent pas d'établir un lien solide entre l'effort de revue et la densité de défauts trouvés suite aux revues. De même, la relation entre l'effort de revue et la densité de problèmes potentiels (PP) détectés n'est pas significative. Plusieurs raisons peuvent expliquer le manque de constance dans les résultats obtenus :

- Une mauvaise estimation de la taille du projet a pu fausser les résultats de densité. Cet aspect a déjà été couvert dans la section 5.2.1.
- Le manque d'expérience des réviseurs qui peut occasionner une détection plus faible des défauts potentiels. Par exemple, le projet J a bénéficié de plusieurs heures de revue, mais aucun problème potentiel n'a été détecté. Bien qu'il puisse s'agir de code parfait ne nécessitant aucune correction, la situation semble pour le moins improbable. D'ailleurs, sur l'ensemble des projets révisés, trois ont un taux de détection inférieur à un PP par heure de revue ce qui semble très peu. Selon les résultats compilés par SmartBear, le taux de détection des PP devraient se situer aux alentours de 15 PP par heure de revue (Cohen, Teleki et al. 2013, p. 87).
- Une vitesse d'inspection trop rapide pour bien détecter le maximum de problème potentiel. Selon le processus de revue documenté utilisé par l'équipe, la vitesse d'inspection ne doit pas dépasser 1000 lignes de code (non commentées) par heure. Cependant, la technique utilisée pour les revues de code c'est-à-dire la vérification des *pull requests* complique un peu la tâche de l'équipe pour évaluer la taille des éléments à revoir. En effet, l'utilisation de l'outil de comparaison fourni par TFS indique clairement les modifications qui ont été faites dans un fichier ce qui fait qu'il n'est pas nécessairement revu en entier. Il devient alors difficile d'évaluer le temps nécessaire pour inspecter 1000 lignes de code. Aux fins d'analyse, un simple ratio des heures de revue par rapport à la taille fonctionnelle des projets permet de constater des écarts importants d'un projet à l'autre, allant de 4,64 à 35,40 PFC revus par heure d'inspection. Avec l'expérience, l'équipe devra trouver une meilleure façon de

déterminer la taille du code à inspecter et ajuster en conséquence le nombre d'heures de revue nécessaire pour compléter la vérification.

Bref, il n'est pas possible de tirer des conclusions empiriques fortes à partir de ces mesures, mais quand même une tendance semble se dessiner : les projets ayant des taux d'efficacité élevé ont généralement un coût par PFC plus bas. Cette constatation est en accord avec celle faite précédemment qui spécifie qu'une densité plus basse de défauts occasionne un coût unitaire plus bas. Dit autrement, chaque problème potentiel détecté diminue le nombre de défauts trouvés en test et en production, diminuant par le fait même le coût.

5.2.4 Mesures de *rework*

Les heures de *rework* ont été les plus difficiles à obtenir. En effet, les premiers cycles de vérification de mesure ont révélé de grands écarts dans la manière dont les heures ont été comptabilisées. De plus, pour les projets qui étaient déjà en cours lors du démarrage de l'initiative, les efforts de *rework* étaient tout simplement manquants. Pour y remédier, les ajustements suivants ont été apportés :

- Pour les projets où les efforts de *rework* étaient présents mais semblaient erronés, les analystes responsables des projets ont été rencontrés individuellement pour comprendre comment ils avaient comptabilisé les heures et, lorsque nécessaire, les mesures antérieures ont été ajustées pour tenir compte des écarts possibles. Bien que les explications aient été données à chacun des analystes verbalement, la documentation sur le processus de revue par les pairs devra être bonifiée pour clarifier la collecte des efforts de *rework*.
- Pour les projets où l'effort de *rework* était manquant, une estimation a été faite avec la meilleure information disponible. Il a été estimé à partir des feuilles de temps, des défauts identifiés lors du QA et des commentaires sur les *commits* dans l'outil de gestion de code source. La logique d'estimation était la suivante : tout *commit* dans le gestionnaire de code source qui suivait la documentation d'un défaut et dont le commentaire spécifiait qu'il s'agissait d'une correction de défaut s'est vu attribuer un

nombre d'heures. Ce nombre était basé sur l'effort de développement dans la semaine où la correction a été faite. Par exemple, pour une certaine semaine, 60 heures de développement ont été entrées dans l'outil de feuille de temps et durant cette même semaine, six *commits* ont été faits dans le gestionnaire de code source. De ceux-ci, deux concernaient des corrections de défauts. Ces deux défauts se sont vus octroyer 10 heures de *rework* chacun pour un total de 20 heures de *rework* pour cette semaine. Cette méthode est très approximative, mais elle est la plus fiable qu'a trouvée l'équipe pour estimer les données de *rework* manquantes.

Malgré les efforts d'estimation, la relation directe entre l'effort de revue et l'effort de *rework* n'est pas évidente. Les données disponibles ne permettent donc pas à ce jour de confirmer qu'une augmentation de l'effort de revue diminue l'effort de *rework.* Toutefois, il est possible que ce manque de corrélation provienne du manque de constance dans la mesure ou du manque d'expérience de l'équipe dans les révisions. L'effort de *rework* étant directement lié au nombre de défauts qui ont passé l'étape de revue sans être détectés, l'amélioration des compétences des membres de l'équipe dans la détection des problèmes potentiels devrait logiquement diminuer d'autant l'effort de *rework.*

Une chose qui est cependant claire, c'est qu'une diminution de l'effort de *rework* diminue le coût par PFC du projet. Même si cette information peut sembler évidente, il est toutefois intéressant de la voir confirmée par les mesures. Tout comme le nombre de défauts (traité dans la section 5.2.2), la relation est particulièrement forte si l'on retire les projets avec une faible qualité de données :

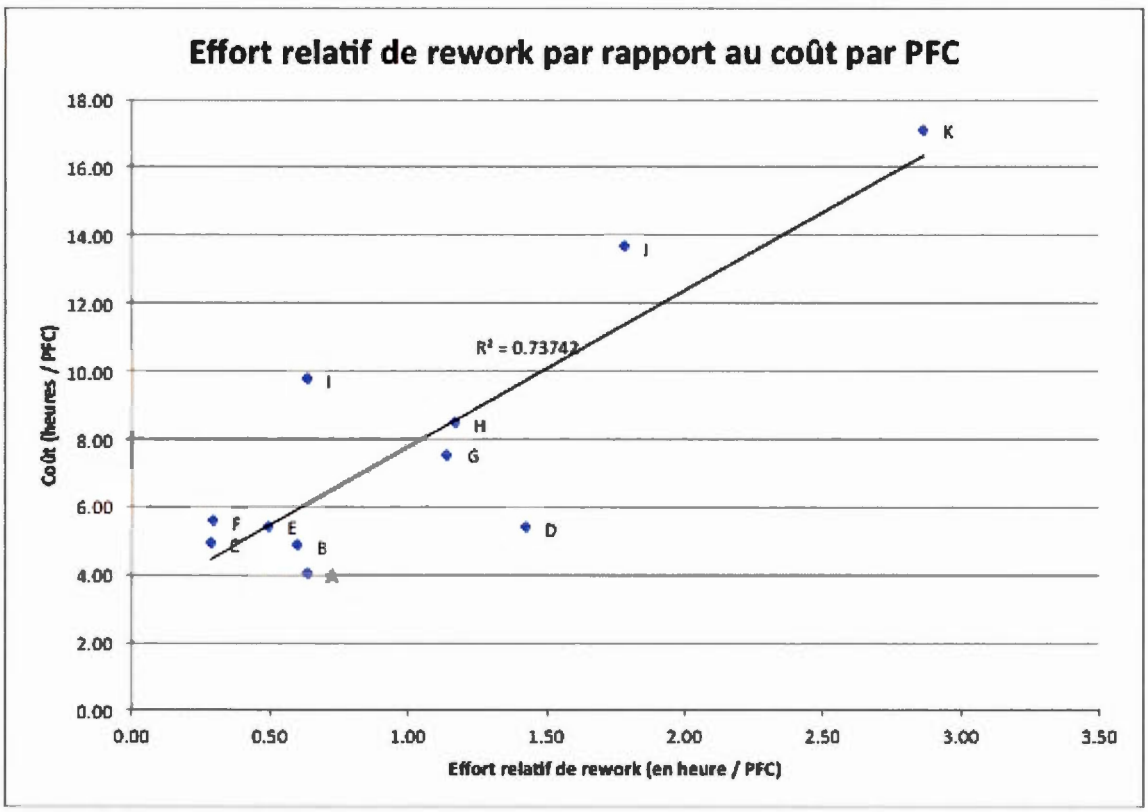


Figure 5.8 Effort relatif de *rework* par rapport au coût par PFC (tous les projets)

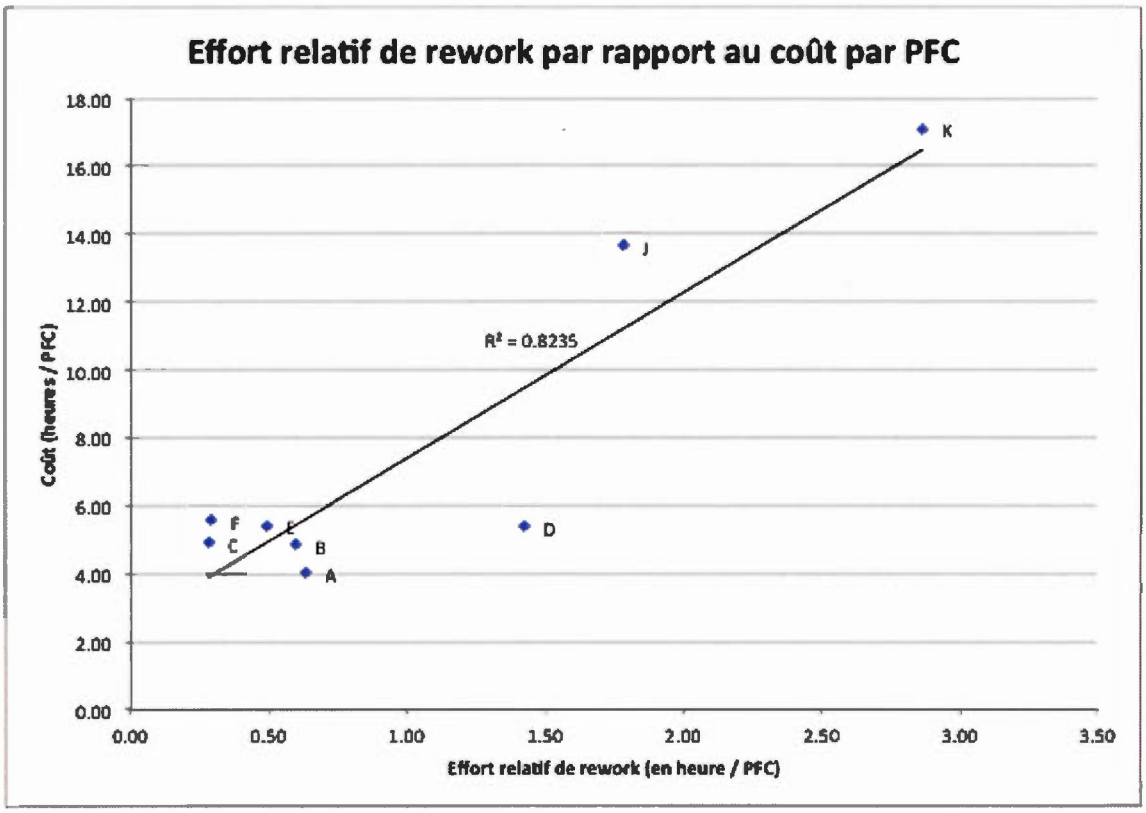


Figure 5.9 Effort relatif de *rework* par rapport au coût par PFC (projets avec faible fiabilité exclus)

5.2.5 Mesure de qualité

La mesure de la qualité interne du code est quelque chose de nouveau dans l'équipe. En effet, avant cette initiative, l'équipe n'avait pas en place l'outillage nécessaire pour quantifier la qualité interne des applications. L'initiative ayant été d'une durée assez courte, il est encore trop tôt pour déterminer l'impact des revues sur les indices de qualité du code, mais le simple fait de rendre l'information disponible donne déjà des résultats concrets. Grâce à l'analyse faite par SonarQube, l'équipe a identifié plusieurs problèmes potentiels et certains plus importants ont déjà été évalués et corrigés.

L'introduction de SonarQube donne aussi un éclairage nouveau sur des informations inconnues jusqu'alors. Par exemple, même avec des données fragmentaires, il semble y avoir une certaine relation entre la qualité du code et le temps moyen de correction.

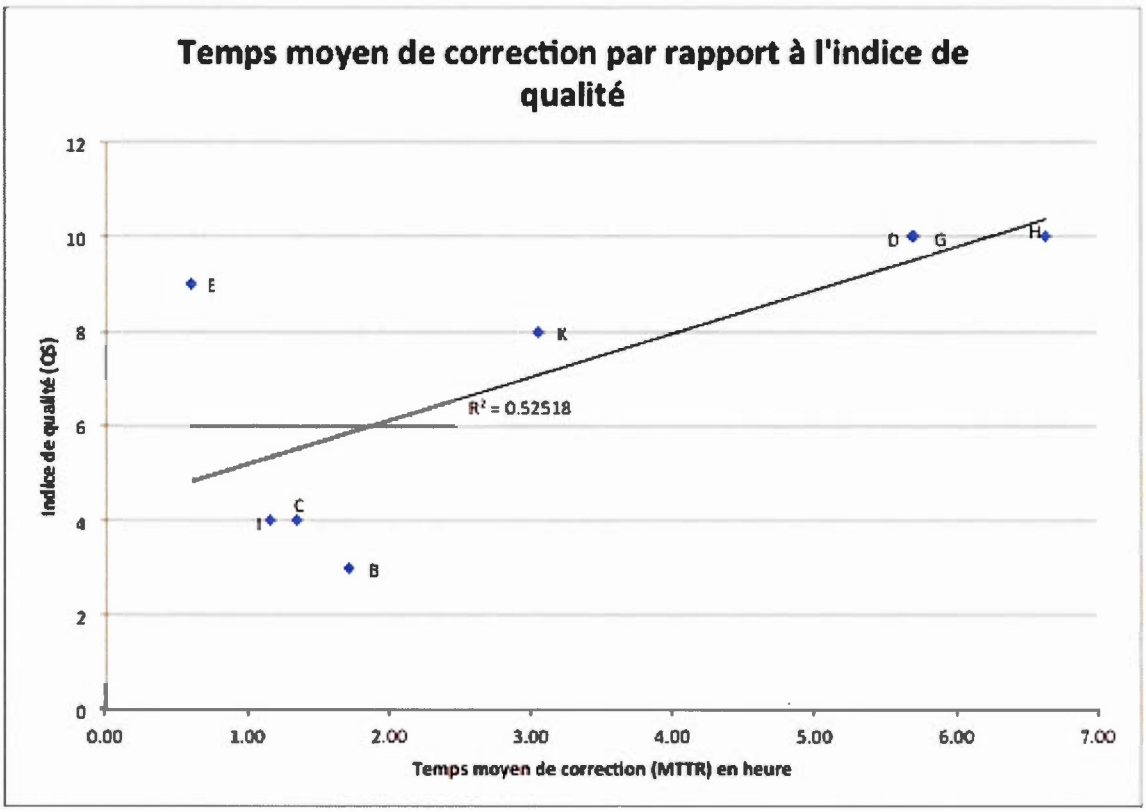


Figure 5.10 Relation entre le temps moyen de correction (MTTR) et l'indice de qualité.

Comme à la section 5.2.2, le projet E semble être dans une classe à part et ne pas suivre la tendance générale. Si on l'exclut de l'analyse, la relation est beaucoup plus évidente :

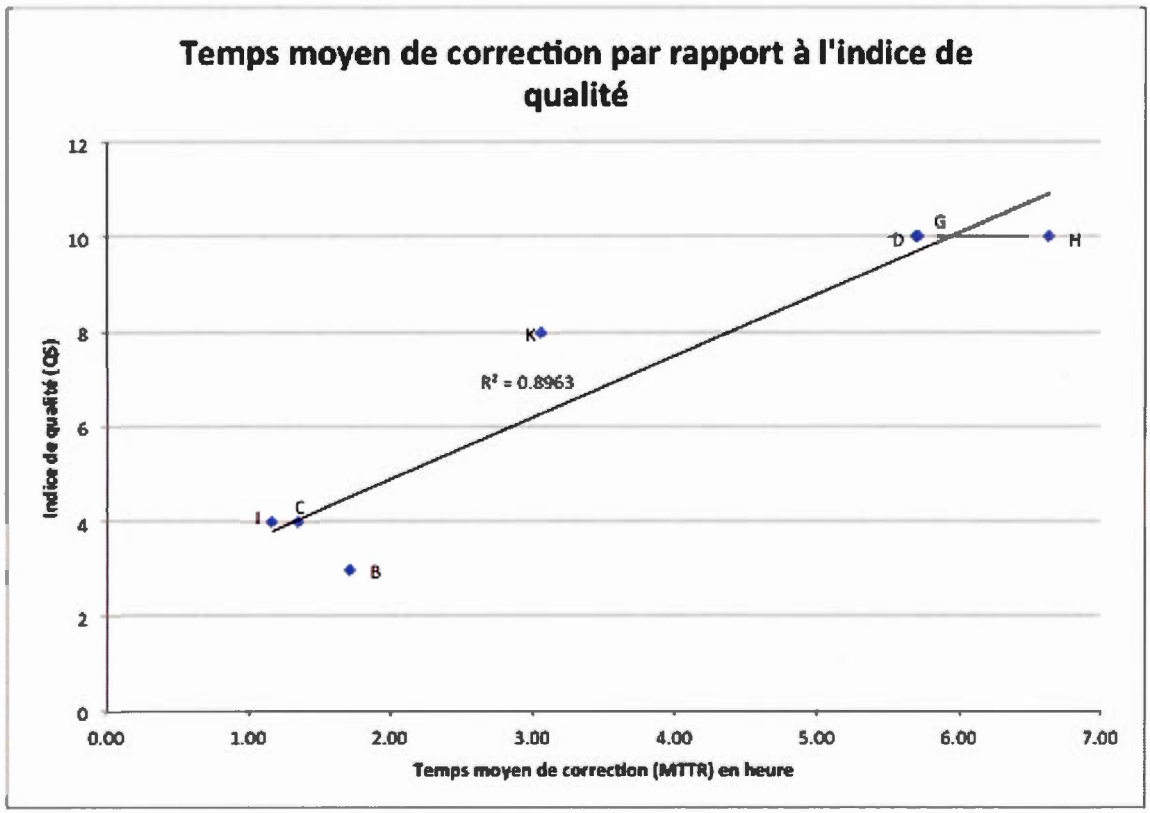


Figure 5.11 Relation entre le temps moyen de correction (MTTR) et l'indice de qualité (projet E exclu)

Cette tendance est conforme à ce qui est décrit dans la littérature sur le sujet soit qu'un code de plus grande qualité sera généralement plus simple à comprendre et donc plus rapide à corriger. Il est aussi important de rappeler que la dette technique doit éventuellement être « remboursée » d'une façon ou d'une autre (soit volontairement par des initiatives de correction, soit par l'obligation d'effectuer du *refactoring* d'urgence pour corriger un défaut ou introduire une nouvelle fonctionnalité). La surcharge occasionnée par la dette technique empiète sur le temps disponible pour développer de nouvelles fonctionnalités et répondre aux besoins des clients. Ken Power a d'ailleurs dédié un article intéressant sur le sujet (Power 2013). Dans cet article, il fait la distinction entre la dette technique, une dette volontaire occasionnée

par des compromis réfléchis, et la dette de qualité, involontaire, qui est liée à un manque de qualité du code. Toutefois, SonarQube traite les deux comme un seul et même concept et l'équipe Web-Micro fait de même.

5.3 Sondage de satisfaction

Pour juger de la satisfaction de participants avec le processus de revue et leur permettre de s'exprimer sur les apprentissages faits durant l'initiative, un sondage a été réalisé auprès des développeurs durant les semaines du 9 et 16 juillet 2017. Il a été envoyé à 16 personnes et 12 y ont répondu soit un taux de réponse de 75 %. Le sondage comportait 12 questions à choix de réponses et deux questions ouvertes. Il était anonyme, a été fait en ligne grâce à l'outil Google Forms et peut être consulté ainsi que la représentation graphique des résultats obtenus à l'ANNEXE VI. Dans les sections suivantes, les résultats du sondage seront détaillés et analysés. Les tableaux de la section 5.3.1 donnent les résultats détaillés des 12 questions avec un choix de réponses. Les réponses aux questions ouvertes (13 et 14) seront traitées dans l'analyse des résultats.

5.3.1 Résultats du sondage

Tableau 5.7 Résultats de sondage (Sections « Expérience des réviseurs et Expérience des auteurs »)

Participant	Réviseurs				Auteurs			
	1. Nb	2. Complexité	3. Partage technique	4. Apprentissage application	5. Nb	6. Complexité	7. Apprentissage technique	8. Partage application
1	0	Ne sais pas / je n'ai pas fait de révision	Ne sais pas / je n'ai pas fait de révision	Ne sais pas / je n'ai pas fait de révision	4	Simple	Souvent	Occasionnellement
2	0	Ne sais pas / je n'ai pas fait de révision	Ne sais pas / je n'ai pas fait de révision	Ne sais pas / je n'ai pas fait de révision	7	Simple	Souvent	Occasionnellement
3	6	Simple	Occasionnellement	Occasionnellement	3	Simple	Souvent	Occasionnellement
4	5	Ni simple, ni complexe	Occasionnellement	Souvent	2	Simple	Occasionnellement	Occasionnellement
5	3	Très simple	Souvent	Très souvent	10	Très simple	Très souvent	Souvent
6	10	Ni simple, ni complexe	Occasionnellement	Occasionnellement	10	Très simple	Occasionnellement	Occasionnellement
7	2	Très simple	Occasionnellement	Occasionnellement	3	Très simple	Occasionnellement	Occasionnellement
8	2	Très simple	Occasionnellement	Occasionnellement	2	Très simple	Occasionnellement	Occasionnellement
9	4	Ni simple, ni complexe	Occasionnellement	Occasionnellement	5	Ni simple, ni complexe	Occasionnellement	Occasionnellement
10	4	Ni simple, ni complexe	Occasionnellement	Souvent	2	Complexe	Souvent	Occasionnellement
11	3	Ni simple, ni complexe	Souvent	Souvent	3	Simple	Souvent	Souvent
12	5	Très simple	Très souvent	Souvent	2	Très simple	Très souvent	Souvent

Tableau 5.8 Résultats de sondage (Section « Général »)

Participant	Général							
	9. Satisfaction TFS 2015	10. Satisfaction SonarQube	11. Satisfaction checklist	12.1 Uniformité checklist	12.2 Uniformité documentation défauts	12.3 Uniformité Processus de revue	12.4 Uniformité Modèle app.	12.5 Uniformité outils
1	Ne sais pas / je ne l'ai pas utilisé	Ne sais pas / je ne l'ai pas utilisé	Ne sais pas / je ne l'ai pas utilisé	Ne sais pas / je ne l'ai pas utilisé	Ne sais pas / je ne l'ai pas utilisé	Ne sais pas / je ne l'ai pas utilisé	Ne sais pas / je ne l'ai pas utilisé	Ne sais pas / je ne l'ai pas utilisé
2	Satisfait	Satisfait	Satisfait	Énormément	Moyennement	Énormément	Ne sais pas / je ne l'ai pas utilisé	Énormément
3	Satisfait	Ne sais pas / je ne l'ai pas utilisé	Satisfait	Beaucoup	Un peu	Moyennement	Moyennement	Beaucoup
4	Très satisfait	Ne sais pas / je ne l'ai pas utilisé	Très satisfait	Énormément	Beaucoup	Beaucoup	Beaucoup	Beaucoup
5	Satisfait	Neutre	Satisfait	Beaucoup	Moyennement	Beaucoup	Énormément	Moyennement
6	Satisfait	Satisfait	Satisfait	Énormément	Moyennement	Beaucoup	Ne sais pas / je ne l'ai pas utilisé	Beaucoup
7	Très satisfait	Ne sais pas / je ne l'ai pas utilisé	Très satisfait	Beaucoup	Beaucoup	Beaucoup	Beaucoup	Ne sais pas / je ne l'ai pas utilisé
8	Très satisfait	Très satisfait	Très satisfait	Énormément	Énormément	Énormément	Énormément	Énormément
9	Satisfait	Neutre	Neutre	Moyennement	Un peu	Beaucoup	Un peu	Beaucoup
10	Très satisfait	Très satisfait	Satisfait	Énormément	Énormément	Énormément	Énormément	Énormément
11	Satisfait	Satisfait	Satisfait	Beaucoup	Beaucoup	Beaucoup	Énormément	Beaucoup
12	Très satisfait	Neutre	Neutre	Ne sais pas / je ne l'ai pas utilisé	Énormément	Énormément	Énormément	Énormément

5.3.2 Analyse des résultats du sondage

Section « Expérience comme réviseur »

Pour ce qui est de l'expérience des réviseurs, elle semble avoir été positive dans l'ensemble. Sur les 12 personnes ayant rempli le sondage, 10 ont participé à une revue comme réviseur. Par conséquent, tous les résultats qui suivent ne vont inclure que les réponses de ces 10 participants.

Les réviseurs ont réalisé entre 2 et 10 revues durant l'initiative avec une moyenne de 3,66 revues par participants. Aux fins de validation, la médiane a également été vérifiée et elle se situe très près de la moyenne (3,5).

Parmi les réviseurs, 50 % ont répondu que le processus, du point de vue du réviseur est simple ou très simple. L'autre 50 % des réviseurs ont qualifié le processus de revue de ni simple, ni complexe. À première vue, il serait facile d'émettre l'hypothèse que le manque d'expérience peut expliquer le résultat et que le processus se clarifie au fil du temps. Cela semble probable étant donné que seulement deux participants ont été réviseurs à plus de cinq occasions. Cependant, en regardant les résultats de plus près, on remarque que l'expérience ne semble pas diminuer la perception de complexité du processus. Au contraire, la réponse « Ni simple, ni complexe » est la plus populaire parmi les développeurs qui ont réalisé quatre révisions ou plus (moyenne de 5,2 révisions pour les participants ayant donné cette réponse). Quelques possibilités pour expliquer ce résultat :

- En participant à plus de revues, les gens sont peut-être confrontés à plus de cas limites, inconnus ou non documentés.
- Le fait de réviser plusieurs projets différents en peu de temps peut rendre le contexte plus difficile à comprendre puisque le participant doit fréquemment se remettre en tête l'application à réviser. Cela pourrait être particulièrement vrai lorsqu'il s'agit d'un projet différent de celui sur lequel le réviseur travaille au moment de la revue.

- Les développeurs qui ont plusieurs révisions en cours simultanément ont un effort de gestion plus important pour faire le suivi des problèmes potentiels identifiés, ajoutant une surcharge cognitive qui peut être plus difficile à gérer.
- Comme certains projets n'avaient pas encore été migrés à TFS 2015, il fallait pour certaines revues continuer d'utiliser les fiches de revue papier, ce qui est une tâche beaucoup plus fastidieuse que de faire une révision grâce à l'outillage moderne fourni par TFS 2015.

Dépendant des raisons réelles qui expliquent ce résultat, de la formation, des améliorations dans la documentation, le processus de revue ou l'assignation des réviseurs sur les projets pourraient faire partie des pistes à explorer pour diminuer la complexité (réelle ou perçue) du processus. Ce point mérite d'être discuté lors des prochaines rencontres de développeurs ou lors de la présentation des résultats de l'initiative afin de mieux le comprendre. Pour conserver la confidentialité des résultats, un deuxième sondage pourrait aussi être envoyé pour obtenir plus de précisions.

Pour ce qui est du partage de compétences techniques, tous les participants ont partagé des compétences techniques au moins à l'occasion (70 %), mais parfois souvent (20 %) ou très souvent (10 %). Il semble donc que le transfert de connaissance se fasse comme souhaité.

La même chose est aussi vraie pour l'acquisition de connaissance sur les applications. Comme le nombre d'applications supportées par l'équipe est élevé, il est très important que le plus grand nombre de personnes possibles soient à l'aise avec les applications pour faciliter les efforts de maintenance et de support. Ici également, tous les participants ont acquis des connaissances sur les applications au moins à l'occasion (50 %), mais parfois souvent (40 %) ou très souvent (10 %). Il sera intéressant de voir dans le futur si le fait d'avoir plusieurs nouveaux développeurs au courant du fonctionnement d'une application permettra de réduire le coût des prochains efforts de maintenance sur ces applications.

Section « Expérience comme auteur »

Pour ce qui est des auteurs, les 12 personnes ayant participé au sondage ont vu leur travail faire l'objet de revue à deux reprises ou plus. L'analyse pour cette partie du sondage sera donc faite sur l'ensemble des résultats obtenus.

Pour ce qui est de la complexité du processus de revue en tant qu'auteur, 10 personnes sur 12 ont jugé le processus simple ou très simple. Il semble donc que les auteurs soient généralement à l'aise avec la façon dont les commentaires sur l'artéfact sont communiqués et qu'ils aient une bonne compréhension du mécanisme de correction et de suivi des problèmes potentiels identifiés. Un des participants a toutefois indiqué que le processus de revue en tant qu'auteur était complexe. Étant donné la nature confidentielle du sondage, il est impossible d'aller recueillir des précisions auprès de cette personne. Cela pourra faire l'objet de discussions lors des rencontres de développeurs, de la présentation des résultats ou encore d'un sondage de suivi tel que décrit précédemment.

Pour l'apprentissage de compétences techniques, les résultats obtenus sont positifs. 100 % de participants affirment avoir appris de nouvelles compétences techniques grâces aux revues, que ce soit par les commentaires ou les discussions avec le ou les réviseurs et de ceux-ci, 58,4 % des participants affirment acquérir de nouvelles compétences souvent (41,7 %) ou très souvent (16,7 %). C'est une bonne nouvelle pour l'amélioration des connaissances de l'équipe en général.

Pour ce qui est de la transmission d'informations sur les applications, les résultats sont plus mitigés. Même si 100 % des auteurs affirment avoir partagé des connaissances avec les réviseurs, seulement 25 % affirment l'avoir fait souvent alors que 75 % affirment l'avoir fait occasionnellement. Cela contraste avec les résultats obtenus des réviseurs où 50 % d'entre eux affirment avoir appris de nouvelles connaissances sur les applications souvent ou très souvent. Cette différence dans les résultats pourrait s'expliquer par le fait que les auteurs ont une connaissance plus approfondie des applications sur lesquelles ils travaillent. Ils n'ont pas nécessairement conscience qu'en expliquant aux réviseurs l'usage et l'architecture interne

des modules qu'ils ont modifiés, ils se trouvent à les former sur le fonctionnement de l'application. Cette information peut alors être utile aux réviseurs qui, en plus de pouvoir mieux comprendre le code et ainsi faire un meilleur travail de révision, peuvent également être plus efficaces lors du support.

Section « Général »

La partie générale du sondage comprend trois groupes de questions : un sur la satisfaction avec l'utilisation des outils mis en place, un sur l'impact de ces outils et d'autres techniques sur l'uniformité du processus de développement et une dernière section avec des questions ouvertes sur leur expérience et sur leurs suggestions pour améliorer les revues.

Pour ce qui est de la satisfaction avec les outils, les résultats sont généralement positifs, mais quelques points méritent une attention plus particulière. Pour ce qui est de TFS 2015, 100 % des participants l'ayant utilisé, soit 11 sur 12, se sont décrits comme satisfait ou très satisfait donc peu de chose à dire à ce sujet. Les résultats avec SonarQube sont plus mitigés. D'abord, un tiers des participants (4 sur 12) au sondage ne l'ont pas utilisé, ce qui est possible étant donné que tous les projets n'y sont toujours pas intégrés. Cinq participants sur les huit restants sont satisfaits ou très satisfaits de l'outil alors que trois sont neutres. En plus de l'intégration non complétée de l'outil, une autre possibilité pour expliquer ces résultats est le manque de formation et d'information sur le fonctionnement de l'outil et sur les possibilités qu'il offre. D'ailleurs, certains des commentaires dans les questions ouvertes du sondage faisaient mention du manque de connaissances sur SonarQube. C'est un point à considérer. Pour finir avec la satisfaction des outils, 75 % des participants sont satisfaits ou très satisfaits avec la liste de vérification mise en place. Seule ombre au tableau, un participant affirme ne pas savoir ou ne pas avoir utilisé la liste de vérification, ce qui ne devrait pas être le cas. En effet, tous les répondants ont participé à une revue comme auteur ou réviseur. Comme cette liste sert autant à l'auteur pour préparer son code à la révision qu'au réviseur pour identifier les points à surveiller, il aurait dû y avoir 100 % de réponse sur cette question. Il sera probablement nécessaire d'insister sur l'importance de la liste de vérification lors des prochaines rencontres.

Les gens ont également été sondés pour connaitre leur opinion sur l'influence des outils mis en place dernièrement pour uniformiser le processus de développement et le rendre plus prévisible. Les participants devaient noter sur une échelle allant de « pas du tout » à « énormément » l'impact d'un outil pour favoriser l'uniformité du processus de développement. En compilant les réponses des participants ayant donné la réponse « beaucoup » et « énormément » pour un outil, on obtient le classement suivant (en ordre décroissant) :

1. Un processus de revue documenté (10 réponses)
2. Une liste vérification (9 réponses)
3. Les nouveaux outils de développement TFS 2015 et SonarQube (9 réponses)
4. Un modèle d'application (7 réponses)
5. Un standard dans la documentation des défauts (6 réponses)

Ces résultats sont de bon augure puisqu'ils semblent confirmer que les participants saisissent l'importance du processus de revue comme une étape pour uniformiser les pratiques de développement. De plus, les outils mis en place sont également perçus comme un pas dans la bonne direction dans l'atteinte de cet objectif.

Le cas du modèle d'application est intéressant : suite aux commentaires recueillis durant l'initiative, certains participants ont exprimé le souhait d'avoir un modèle d'application intégrant les bonnes pratiques d'architectures et d'organisation du code tel que décrit dans la liste de vérification ; bref, une « implémentation » de la liste de vérification. Le but de ce modèle était de fournir une base commune pour le démarrage de nouveau projet ou de point de référence pour les nouveaux modules. Cela ne faisait pas partie de l'initiative au départ, mais suite à la demande, une première version du modèle d'application a été mise en place en avril soit au milieu du projet et utilisée à quelques reprises. À première vue, le résultat de 7 le classe en queue de peloton, mais le modèle d'application arrive ex æquo en tête des outils ayant obtenu le plus grand nombre de mention « énormément » (42 %) et en première place lorsqu'on exclut les participants qui ne l'ont pas utilisé (56 %). Autrement dit, ceux qui en ont fait l'expérience le trouvent très utile et en sont très satisfaits. Donc, bien que cet outil n'était pas prévu au démarrage de l'initiative de revue par les pairs, cela il démontre

comment la rétroaction avec les gens de l'équipe peut mener à des résultats imprévus et positifs et c'est pourquoi il a été décidé de l'inclure dans le sondage de satisfaction et dans l'analyse des résultats.

Finalement, quelques mots sur les commentaires obtenus grâce au sondage. Deux questions ouvertes ont été incluses dans le sondage. La question 13 était : « Qu'avez-vous appris de votre expérience avec le processus de revue? » Voici les résultats de la question 13 (tel quel) :

- Le partage de connaissances est intéressant. À la fois pour le développeur que pour le *reviewer*.
- Intéressant de voir une autre perspective sur notre code ou sur le code d'un collègue.
- Le processus m'a permis d'accélérer ma démarche pour rendre mon travail moins "brouillon" et dispersé.
- Une fois que l'égo est mis de côté il n'y a que du positif.
- Qu'il y avait plusieurs choses que je ne faisais pas de la bonne façon. Ça été très bénéfique pour moi.
- Plus efficace si fait sur une base régulière en cours de développement et l'utilisation de tfs2015 rend l'expérience plus dynamique.

La question 14 était : « Avez-vous des suggestions à faire pour améliorer le processus de revue (à tous points de vue)? » Voici les résultats de la question 14 (tel quel) :

- Le processus doit toujours être présenté comme étant bidirectionnel.
- SonarQube doit être mis de l'avant plus en détail. L'outil présente beaucoup d'informations intéressantes à différents degrés, mais il semble négligé. Les *code smells* moins triviaux que les références non utilisées sont généralement significatifs.

Que peut-on tirer que ces commentaires? Quelques informations intéressantes :

- Le partage de connaissances semble être un des aspects les plus appréciés de l'expérience. Un des commentaires suggère d'insister sur le caractère bidirectionnel

des revues et c'est une excellente suggestion qui devra être adressée lors des rétroactions en fin de projet.

- Que l'aspect humain de la revue demeure un enjeu important à considérer (gestion des égos).
- Que des revues courtes et fréquentes sont préférables à de longues revues, ce qui est très bien décrit dans la littérature sur le sujet et abondamment traité dans la section 4.3.
- Poursuivre la formation sur le processus de revue dans son ensemble. Un commentaire suggère d'améliorer les connaissances sur SonarQube, ce qui sera nécessaire, mais les autres étapes et outils du processus devront également être abordés.

5.4 Améliorations au processus de revue

Durant le projet, suite aux premières prises de mesures et aux discussions avec les participants, plusieurs opportunités d'amélioration ont été suggérées ou découvertes en cours de route. Voici un aperçu des améliorations au processus qui ont été réalisées en cours de projet ou qui sont en voie de l'être.

5.4.1 Applications modèles

Comme discuté dans la section 5.3.2, des demandes ont été faites d'avoir un exemple concret d'application qui respecterait les bonnes pratiques d'architecture, d'organisation du code et des nomenclatures documentées dans la liste de vérification. Après discussion avec les gestionnaires, ils ont approuvé la mise en place d'une application modèle pour servir de base aux nouveaux projets ou comme référence pour les projets existants. Cependant, comme cette partie du projet n'était pas prévue au départ, peu de temps a pu y être consacré. Le modèle d'application est donc incomplet c'est-à-dire que certains des modules (comme la gestion des données et certains standards d'interface utilisateur) ne sont pas encore en place. La documentation est aussi très sommaire. Toutefois, le modèle était suffisamment avancé pour servir de base à quelques nouvelles applications et déjà, quelques itérations du modèle ont été réalisées à partir des commentaires des développeurs qui l'ont utilisé. L'expérience est posi-

tive jusqu'à maintenant et l'équipe envisage de continuer de bonifier le modèle d'application dans le futur.

5.4.2 Uniformiser les mesures de taille fonctionnelle pour améliorer les estimations

Le problème de dépassement de coût des projets est déjà connu depuis un moment dans l'équipe. L'écart du coût relatif en heures/PFC étant important d'un projet à l'autre (et 4,01 à 17,11 pour les 11 projets de l'initiative), l'équipe cherchait à identifier la source de ces dépassements de coûts. Elle a d'abord voulu vérifier si les dépassements de coût provenaient d'une estimation incorrecte au départ ou de fluctuations dans le temps de réalisation.

Les premières mesures ont démontré que des lacunes étaient présentes dans la mesure de la taille fonctionnelle des projets. Plusieurs méthodes de mesure différentes étaient utilisées : certains projets étaient mesurés avec la méthode COSMIC, certains étaient estimés en heures (sans mesure de taille réelle) et d'autres étaient estimés en heures avec la méthode du *poker planning*. Le *poker planning* a été introduit pour expérimenter si cette méthode d'estimation allait donner de meilleures estimations que la méthode COSMIC pour les projets de type Internet de grande taille. Les résultats sur la fiabilité des estimations à partir du *poker planning* devraient être connus bientôt. Cependant, même les projets estimés avec COSMIC, des écarts importants demeurent entre les estimations et le coût réel.

L'équipe a d'abord tenté de déterminer si le coût par PFC actuel (5,4 h/PFC), calculé il y a quelques années, était toujours adéquat. Cependant, comme les données actuelles de projets ne démontraient pas de tendance claire, l'équipe a commencé à inspecter les données de taille fonctionnelle plus en détail. Après vérification et discussion, l'équipe a remarqué des différences d'interprétation dans la façon de mesurer les projets. Dépendant du mesureur, il pouvait y avoir des différences allant parfois du simple au double. Avec de tels écarts, il est difficile de savoir si le décalage entre l'estimation et le coût réel provient d'une mauvaise mesure de taille fonctionnelle, d'un coût par PFC désuet ou d'une autre source. De plus, un cal-

cul de taille fonctionnelle erroné fausse d'autres mesures puisqu'elle sert de point de comparaison pour plusieurs d'entre elles (h/PFC, HDR/PFC, densité de défaut, etc.).

Pour tenter de rectifier la situation, un groupe de travail a été mis sur pied pour documenter des règles de mesures, tant pour l'offre de service que pour les spécifications, avec pour objectif premier d'obtenir un mesurage uniforme, quel que soit le mesureur. Une fois les règles comprises, documentées et révisées, il a été déterminé que toutes les mesures pour les projets de plus de 20 jours seraient systématiquement revues par le groupe de travail jusqu'à nouvel ordre pour les valider et pour identifier des patrons de mesure qui pourront alors être ajoutés au document de règles d'estimation. Les résultats sont à venir, mais l'équipe espère ainsi pouvoir vérifier si cela uniformisera le coût relatif final par PFC ou si d'autres variables sont en cause dans les dépassements de coût.

5.4.3 Catégorisation de la sévérité des défauts (revue et bogues)

Une des mesures manquantes actuellement est la caractérisation de la sévérité des problèmes potentiels et des défauts. Il est actuellement difficile de savoir si les PP trouvés lors des revues préviennent des défauts importants ou sont surtout concentrés sur des éléments de qualité de code moins critique au bon fonctionnement de l'application. La sévérité des défauts n'est pas systématiquement caractérisée non plus. Il est donc impossible de savoir si les défauts trouvés suite aux revues sont mineurs ou majeurs. Des propositions ayant déjà été faites en ce sens, de la documentation et de la formation sont prévues sous peu.

5.4.4 Qualité des données d'effort

Les premières mesures ont démontré un manque de rigueur dans la documentation des données d'effort, que ce soit l'effort de revue dans les feuilles de temps ou l'effort de *rework* documenté dans l'outil de suivi de défauts. Les participants ont été informés de la situation, le suivi se poursuit et des rappels périodiques sont faits lorsque des écarts ou des informations manquantes sont détectés. La documentation du processus de revue a été bonifiée pour

mieux expliquer la collecte de données d'effort et des exemples illustrés ont été fournis pour couvrir les différents scénarios rencontrés par les participants.

5.4.5 Amélioration de la documentation du processus de revue

Les mesures prises en cours de projet et les résultats de sondage ont permis d'identifier des sections de documentation du processus de revue manquantes ou dont la clarté pouvait être améliorée. Plusieurs ajustements ont déjà été faits en cours de projet, tant sur la documentation du processus de revue que sur les listes de vérification. D'ailleurs, d'autres corrections sont à venir. De toute façon, ces documents vont continuer d'évoluer au fur et à mesure des apprentissages, des expériences vécues et des changements dans l'environnement de l'équipe, qu'ils soient technologiques, humains ou d'affaires.

5.4.6 Formation des développeurs et des analystes

Les résultats de sondage ont mis en lumière des manques dans la formation des participants, particulièrement avec l'outil SonarQube, mais aussi avec certains modules de TFS 2015. Par exemple, la documentation adéquate des défauts (bonne caractérisation des phases d'injection, sévérité et effort de *rework)* est à revoir. Ces formations sont à prévoir dans un avenir rapproché.

CHAPITRE 6

DISCUSSION

En cours de réalisation, certaines situations se sont présentées et méritent qu'on s'y attarde dans le but d'en saisir les implications sur le projet comme tel et sur la suite des choses.

6.1 Résistance aux changements

Un des éléments dont il faut tenir compte lors de l'introduction de revue par les pairs, c'est l'éventualité d'une résistance aux changements. En effet, lorsque les membres d'une organisation ou d'une équipe ont des façons de faire en place depuis plusieurs années, la mise en place d'un changement aussi fondamental que la revue par les pairs peut intimider certaines personnes. Dans le cadre de la mise en place dans l'équipe Web-Micro, certaines résistances ont été rencontrées. La source de ces résistances n'a cependant pas été celle anticipée au départ.

Les travaux de Tom Gilb identifient plusieurs sources de résistance possibles. Une des sources fréquemment rencontrées est le développeur « superstar » : il ne veut pas voir son travail inspecté ou critiqué par les autres, car il considère que son travail est de la plus haute qualité et il ne conçoit pas que des défauts (ou des problèmes potentiels) puissent y être décelés (Gilb and Graham 1993, p. 188). L'équipe Web-Micro compte plusieurs développeurs d'expérience, mais aucun cas d'égo problématique n'y est heureusement présent. Les gens responsables de déployer les revues dans l'équipe craignaient quand même une résistance de ces personnes. Or, il s'est avéré qu'au contraire, les développeurs d'expérience étaient pour la plupart de grands partisans de la mise en place des revues. Une des causes possibles de cette situation inattendue est que les développeurs séniors sont souvent les experts des applications les plus complexes et lorsque la qualité du code diminue, ils sont souvent ceux qui vont devoir « réparer les pots cassés » en cas de problèmes urgents en production ou de dé-

fauts complexes à détecter dans des modules critiques des applications. Ils sont donc ceux qui ont un grand intérêt à voir le code revu (incluant le leur) pour réduire le nombre d'incidents demandant des corrections d'urgence. Dans l'ensemble, aucun des développeurs n'a explicitement opposé de résistance à la mise en place des revues et la garantie de confidentialité des informations semble avoir mis à l'aise tout le monde.

Par contre, la réaction a été différente du côté des analystes. Lors de rencontres préliminaires à la mise en place des revues, des représentants du groupe ont émis des réserves sur certaines parties du processus. Avant le démarrage de l'initiative, les analystes réalisaient déjà à l'occasion des revues de spécifications en groupe pour identifier les possibles omissions ou les manques de clartés de certaines parties. Tous semblaient d'accord pour une revue plus systématique des documents. Toutefois, l'ambiance de la rencontre a changé lorsqu'est venu le temps de parler de documenter les problèmes potentiels détectés. Cette prise de mesure n'a pas été bien reçue, car dans le processus de revue de documentation alors en place, les documents étaient souvent corrigés durant la rencontre d'inspection ou après, mais aucune trace sur la présence d'un problème potentiel dans le document original n'existait. De plus, la garantie de confidentialité ne semblait pas rassurer le groupe, car les analystes sont peu nombreux (cinq personnes) et bien que les mesures soient anonymes, il leur semblait relativement facile d'identifier l'auteur des documents et donc le « responsable » d'une spécification avec un nombre important de problèmes potentiels. Pour toutes ces raisons, et compte tenu de l'échéancier déjà serré pour la mise en place des revues, celle-ci s'est concentrée sur les développeurs et le code.

Tout cela n'est cependant que partie remise, car le fait qu'au moins 25 % des défauts identifiés trouvent leur source dans les documents d'exigences ou de spécifications mérite qu'on s'y attarde un peu plus. Comme suggéré par les études à la base de la Figure 3.1 (p. 27), un problème détecté avant le développement est beaucoup moins coûteux à corriger. Avec ces résultats en main, il faudrait rencontrer à nouveau les analystes pour leur faire part des avantages pour eux à mettre en place un processus documenté et mesuré des documents et trouver avec eux de solutions possibles à leurs craintes et interrogations. Une section des travaux

futurs (section 7.2) y est d'ailleurs consacrée. Une bonne nouvelle pour terminer : il semble déjà y avoir une plus grande acceptabilité de la revue qu'au départ de la part des analystes, particulièrement depuis la mise en place des révisions des estimations (voir section 5.4.2) donc le fruit parait mûr pour un redémarrage des discussions à ce sujet.

6.2 Défis et obstacles rencontrés

Dans l'ensemble, la réalisation du projet s'est bien déroulée. Toutefois, comme dans ce monde rien n'est parfait, quelques difficultés rencontrées en cours de route ont retardé, ralenti ou complexifié la mise en place des revues ou la prise de mesures.

6.2.1 Installation des outils nécessaires

L'étape de réalisation nécessitait l'installation d'outils pour assurer son bon déroulement. Or, les pratiques de l'entreprise ne permettent pas aux gens des équipes de développement de provisionner des serveurs virtuels, de réserver de l'espace de stockage et d'y installer des outils. De plus, une fois ces outils installés, l'équipe doit également faire appel à l'équipe d'infrastructure, via un système de requêtes informatiques pour ajuster certaines configurations ou inspecter le registre d'erreurs (*error logs*) lorsqu'un problème survient. Les équipes d'infrastructure étant déjà surchargées de travail, elles ne sont pas toujours disponibles rapidement pour les requêtes, particulièrement lorsque celles-ci ne concernent pas des problèmes de production. Cela a occasionné quelques retards au démarrage, lors de l'installation de TFS 2015 et particulièrement pour la prise de mesure de qualité, SonarQube n'ayant été installé en production qu'à la fin d'avril 2017.

6.2.2 Migration des applications vers TFS 2015

Le deuxième frein au bon déroulement du projet était la migration des applications de l'ancien outil de gestion de cycle de vie logiciel (TFS 2010) vers TFS 2015. L'équipe ne pouvait pas profiter d'une migration automatique de ses projets, car elle était plusieurs versions en retard dans son installation de TFS. De plus, comme elle voulait profiter de la migra-

tion pour corriger certains problèmes d'organisation avec son ancienne structure de projets, chaque application devait donc être migrée manuellement. Initialement, l'équipe pensait que la migration serait terminée avant le début du projet de revue, mais ce ne fut pas le cas. Le processus de transfert n'est pas très complexe, mais il requiert quand même quelques heures et surtout, il demande qu'il n'y ait pas de développement actif de l'application pendant quelques jours, le temps que le ménage prémigration soit fait et d'effectuer les vérifications post-migration. À cause de cette contrainte, certains projets n'ont pu être migrés vers le nouvel outil (spécifiquement les projets A et F). Cela a occasionné un processus de revue plus fastidieux, car, ne profitant pas des facilités de revue de TFS 2015, les participants devaient utiliser les fiches de revue papier (tout en utilisant toutefois la même liste de vérification) rendant la révision, le suivi des corrections et la prise de mesures plus complexes.

6.2.3 Arrivée tardive de SonarQube et dépendance avec TFS 2015

Comme pour TFS 2015, chaque projet devait être configuré dans SonarQube pour permettre d'obtenir les mesures de qualité des projets. Du temps a aussi dû être investi pour calibrer l'outil, c'est-à-dire déterminer les règles d'analyse et les critères de qualité que l'équipe voulait adopter. De plus, comme l'analyse de projets par SonarQube est faite via une extension (*plug-in*) dans TFS2015, les projets qui n'ont pas été migrés (A et F) n'ont pas non plus été intégrés à SonarQube ce qui explique les mesures de qualité manquantes pour ces projets. Aussi, certains projets n'ont pas pu être configurés à temps (projet J) et les autres n'ont pas bénéficié d'assez de cycles de développement pour constater des tendances sur l'amélioration des mesures de qualité de ces projets. Cette information sera disponible dans quelques mois.

6.2.4 Déterminer les priorités pour les listes de vérification

La création des listes de vérification a demandé un exercice de synthèse et de priorisation plus important que prévu. Dans un monde idéal, on voudrait que tous les développeurs aient en tête toutes les bonnes pratiques de développement logiciel et puissent les détecter et les identifier facilement. Or, personne n'a cette capacité et c'est pour cela que les listes de vérification existent. Toutefois, pour être efficace, il est recommandé d'utiliser les listes les plus

courtes possible et de limiter le nombre d'items à 20 (Cohen, Teleki et al. 2013, p. 112). L'exercice devient donc un jeu d'équilibre entre vouloir donner un maximum d'information tout en produisant une liste avec le minimum d'items possibles. Dans le cas de l'équipe, l'objectif était de faire tenir la liste de vérification sur une page. Malgré les efforts de synthèse, la liste finale comporte près du double d'éléments que le nombre maximum recommandé soit 38 items. L'objectif sera donc, lorsque les membres de l'équipe seront plus expérimentés avec la revue, de retirer de la liste certains éléments moins importants, désuets ou qui peuvent être détectés d'une autre manière (outils d'analyse de code, librairies standards, application modèle, etc.).

6.2.5 Changement dans la classification des activités de feuille de temps

Un irritant à la prise de mesure a été le changement en 2016 des codes pour les activités de réalisation de projets dans l'outil de feuille de temps. Il fallait donc, pour comptabiliser les temps de développement, de revue, de QA et de *rework*, utiliser différents codes d'activité selon le moment où le projet avait démarré. Certains projets chevauchaient même la période de transition. Il fallait donc combiner différentes activités pour obtenir le temps total par activité pour le projet. Cela devra être pris en compte lors de l'automatisation des prises de mesures. Détail cocasse : la nouvelle liste d'activités de projet n'incluait aucun code pour la revue par les pairs alors que trois codes d'activité existaient dans la liste précédente (un pour la revue des analyses fonctionnelles, un pour la revue des analyses détaillées et un pour la revue de code). Ce n'était pas très commode pour l'initiative de revue qui était sur le point de démarrer! Toutefois, la situation a été corrigée rapidement.

6.2.6 Disponibilité et charge de travail des participants

Finalement, bien que le projet de revue par les pairs était une demande de la direction, il s'agissait d'un projet interne à CIRX qui devait donc avoir un minimum d'impact négatif sur les projets en cours et les demandes des clients. Les priorités d'affaires changeant régulièrement et l'équipe réagissant aux urgences quotidiennes, il a été difficile de trouver le temps nécessaire pour réaliser les activités nécessaires à la bonne marche du projet comme la do-

cumentation, les rencontres, la préparation des formations, les suivis et les prises de mesures. Ces tâches étaient évidemment non prioritaires par rapport aux opérations courantes comme les problèmes de production, le démarrage, la réalisation et le suivi des projets et toutes les autres tâches connexes. Cela a retardé l'échéancier prévu de quelques mois, mais en fin de compte, le budget proposé à la direction pour le projet a été respecté. Les différentes étapes ont simplement été distribuées sur une plus longue période.

6.3 Acceptation du processus par les participants

Selon les résultats obtenus lors du sondage et les discussions informelles avec les participants, ceux-ci semblent satisfaits avec le processus de revue en place. Une rencontre avec les directeurs de l'équipe confirme qu'ils sont aussi contents du processus en place et qu'il ne reviendrait pas en arrière même si certains irritants demeurent.

À cause de leur charge de travail, les développeurs ont parfois du mal à trouver le temps de faire les revues qui leur sont assignées. Les directeurs sont d'accord que du travail reste à faire pour libérer le temps nécessaire aux développeurs pour la revue, particulièrement en fin de projet quand l'échéance approche et que les corrections sont nombreuses. De plus, les taux de détection de problèmes potentiels par heure de revue demeurent faibles, laissant supposer que les revues sont peut-être faites trop rapidement. Il faut cependant laisser le temps aux développeurs de prendre de l'expérience avec la revue. Il est normal qu'il faille un certain temps pour apprivoiser la méthode et obtenir les gains de productivité désirés et il est important de persévérer dans les apprentissages. Wiegers et Beatty le représentent très bien dans leur ouvrage Software Requirements (Wiegers and Beatty 2014, p. 530) et Wiegers en fait également mention dans son livre Peer Reviews in software (Wiegers 2002):

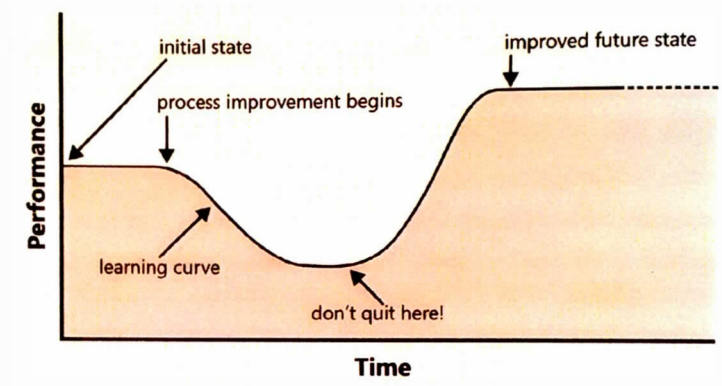


Figure 6.1 Courbe d'apprentissage (Wiegers and Beatty 2014)

Une autre source d'agacement identifiée autant par les développeurs (verbalement et dans le sondage) que les directeurs est la difficulté de déterminer la bonne fréquence et le bon moment pour les revues. Tous s'entendent sur le fait que les longues revues effectuées en fin de projet ne sont pas efficaces et qu'il en faut tout au long du projet, particulièrement au démarrage du projet. Les méthodologies itératives et incrémentales comme Scrum prescrivent des livraisons rapides et fréquentes. Dans ce genre de situation, la préparation de la livraison pour la fin d'un sprint devient un jalon tout indiqué pour réaliser les revues. Or, l'équipe n'utilise pas une méthode itérative, mais plutôt la Métho qui ne requiert pas le découpage du développement en phase. Il devient alors plus difficile de déterminer le « moment idéal » pour la revue, car il y a toujours une petite retouche supplémentaire à faire pour que le code soit « prêt » pour la revue. Il s'agit d'un enjeu important, mais dont la solution n'est pas évidente. Au fur et à mesure que le processus de revue gagnera en maturité, les directeurs pensent que les participants sauront mieux découper leur travail en courtes phases et ainsi déterminer le bon moment pour demander la revue. Dans le cas contraire, il sera peut-être nécessaire de mettre de place des standards ou des lignes directrices (*guidelines*) pour aider les développeurs dans leur prise de décision.

6.4 Retombées pour l'équipe Web-Micro

Un projet comme celui-là est une occasion d'apprentissage qui va souvent au-delà des objectifs prévus au démarrage. Durant la durée de l'initiative, en plus du processus de revue comme tel, d'autres connaissances et compétences transversales ont été acquises. Pour d'autres apprentissages, il s'agit d'une prise de conscience nécessaire qui permettra à l'équipe de progresser dans ses objectifs d'amélioration continue. Voici quelques-unes des retombées qui ont été identifiées.

6.4.1 Rigueur dans la documentation des défauts

Une des constatations lors des premières prises de mesure a été les disparités importantes dans la documentation des défauts d'une personne à l'autre et même d'un projet à l'autre. Dans certains cas, certaines informations étaient manquantes (par exemple, la source du défaut, sa phase de détection, sa sévérité). Dans d'autres cas, les défauts n'étaient carrément pas documentés, car tout le travail de suivi des défauts était fait verbalement ou par courriel. À d'autres occasions, il n'y avait pas de cohérence dans la façon de les documenter les défauts ce qui pouvait fausser les mesures. Par exemple, bien que la méthode de documentation des temps de *rework* soit documentée, il a été découvert en cours de projet que tous ne l'interprétaient pas de la même façon.

Suite à ces constatations, des ajustements ont été faits, la documentation a été bonifiée et les vérifications se poursuivent pour s'assurer que la documentation des défauts est maintenant uniforme. Cela apportera plusieurs bénéfices à court et moyen terme :

- Une réduction de la confusion chez les développeurs, car le processus de documentation et le contenu des défauts seront toujours le même.
- Une meilleure précision des mesures de densité de défaut, de rework.
- Une caractérisation permettant d'identifier les étapes du processus où sont injectés les défauts, permettant la prise d'actions ciblées pour réduire ou éliminer le problème.

6.4.2 Culture de la mesure et son opérationnalisation

La culture de la mesure s'est beaucoup développée CIRX depuis les dernières années. Les mesures sont prises de plus en plus au sérieux et sont utilisées par la direction pour prendre des décisions et juger du bon fonctionnement des processus. Dans l'équipe Web-Micro, les mesures ont longtemps été une sorte de vœu pieux, c'est-à-dire que l'équipe était intéressée par les mesures, mais n'avait pas nécessairement de plan précis sur les données désirées, la façon de les obtenir, les objectifs de ces mesures et sur les actions à prendre une fois les résultats obtenus.

Depuis quelque temps, le désir de quantifier le processus de développement est plus grand, d'abord par la direction, mais aussi par les autres membres de l'équipe et bien que le projet de revue par les pairs ne soit pas le seul responsable de cet appétit pour les mesures, il semble y avoir contribué. Par exemple, une mesure plus précise du temps de *rework* en cours de projet permet déjà de prévoir des écarts entre le temps de réalisation estimé et le temps consommé lorsque celui-ci gonfle plus rapidement que prévu. Même si cela demande un effort supplémentaire de la part des analystes et des développeurs, l'acceptation a été plus rapide que prévu, car tous semblent comprendre l'importance de ces mesures puisque par la suite, elles serviront à améliorer les estimations futures, contribuant ainsi à améliorer leur qualité de vie en minimisant les heures supplémentaires et les urgences (*rush*) imprévues.

De plus, toutes les mesures offertes par les nouveaux outils comme TFS 2015 permettent maintenant aux membres d'un projet de personnaliser des tableaux de bord et ainsi suivre l'avancement du projet :

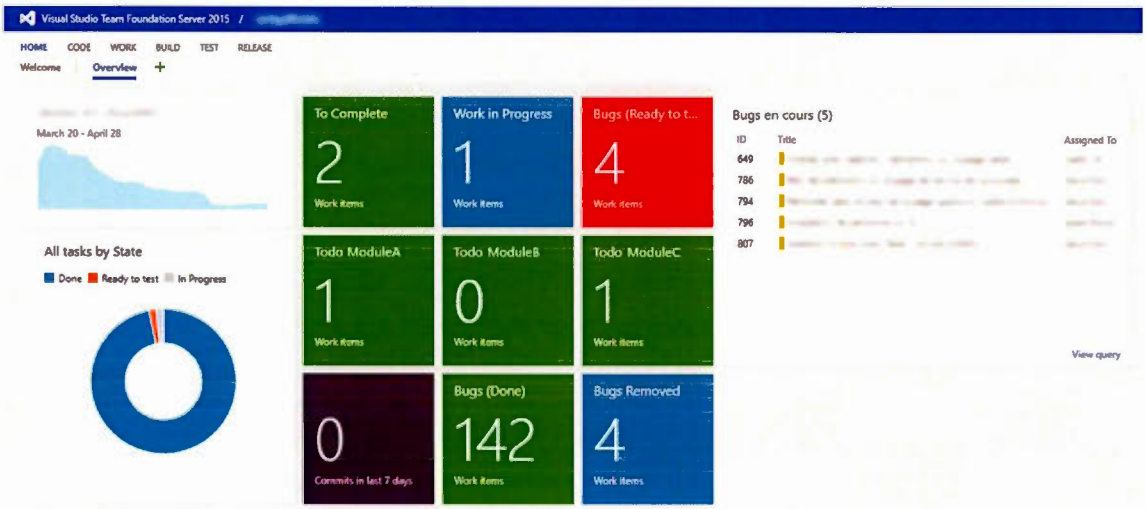


Figure 6.2 Exemple de tableau de bord de projet avec TFS 2015

Finalement, les directeurs ont déjà exprimé le désir de réunir toutes ces nouvelles mesures dans un tableau de bord (*dashboard*) général pour leur permettre d'identifier rapidement les projets qui se déroulent bien et ceux qui doivent faire l'objet d'un suivi plus serré. Ils sont également plus conscients de l'effort nécessaire à des mesures de qualité (suivi, révision, documentation) et semblent plus ouverts à y accorder les ressources humaines, matérielles et monétaires nécessaires.

6.4.3 Revue du processus d'estimation de projet

Une autre des retombées indirectes du projet de mise en place des revues par les pairs est qu'il a apporté une raison supplémentaire de démarrer l'initiative de révision des estimations. La section 5.4.2 traite déjà de ce projet dans le détail, mais il est quand même important de noter que les données nécessaires au processus de revue ont démontré l'ampleur des écarts dans les mesures de taille fonctionnelle entre les projets. Cela a entre autres permis de réali-

ser l'impact de ces différences sur un ensemble de mesures importantes (ex. : densité de défaut, estimation de l'effort de réalisation, impact de la revue sur la qualité) desquelles il sera difficile d'obtenir des réponses tant que la situation ne sera pas corrigée. Les membres de l'équipe, et particulièrement les analystes, sont non seulement plus conscients de l'importance d'un bon calcul de taille fonctionnelle, mais les discussions sur le sujet sont passionnées et elles mettent en lumière plusieurs scénarios de mesure complexes qui sont alors discutés, analysés et documentés. Au fil des échanges, il a également été décidé de mettre en place progressivement une documentation permanente de la taille des applications pour aider les analystes lors des estimations des maintenances futures. Cela devrait permettre d'augmenter la maturité de l'équipe dans plusieurs aspects de son processus de développement.

6.4.4 Désirs exprimés d'un processus plus contrôlé

Dans le but d'obtenir une meilleure acceptation du processus de la part des développeurs, il a été décidé au début de l'initiative que toutes les activités du processus seraient basées sur la confiance, le professionnalisme et la rigueur de chacun. Bien que certains outils en place comme TFS permettent des formes de contrôle (par exemple, impossibilité d'inclure du code dans le gestionnaire de code source sans revue), les responsables de l'initiative étaient d'avis que cela entrainerait de la friction qui pourrait créer des frustrations.

Or, il fut surprenant de constater que plusieurs développeurs ont exprimé le souhait de voir les contrôles *renforcés* pour assurer une meilleure qualité du code. C'est une agréable surprise qui permet de réaliser que bien que ce ne soit pas toujours exprimé explicitement, bien des développeurs ont la qualité des applications à cœur et ils sont prêts à faire des efforts supplémentaires et même à « endurer » un peu de friction pour y parvenir. Cela est une bonne nouvelle pourvu que la direction ait le même souci pour la qualité et n'autorise pas les développeurs à court-circuiter les contrôles mis en place lorsque la pression monte (car c'est généralement dans ces situations que les contrôles sont les plus importants). Pour le moment, aucun mécanisme de contrôle n'a été activé, mais si le besoin se fait sentir, il est intéressant

de savoir que des représentants des développeurs sont prêts à défendre la mise en place de ces points de vérification pour le bien de l'équipe.

CHAPITRE 7

CONCLUSION

La mise en place des revues par les pairs dans l'équipe Web-Micro a été une occasion de revoir certaines activités du processus de développement pour y déceler les points forts et les points faibles. Bien que les mesures prises lors de l'initiative ne peuvent pour l'instant démontrer de façon quantitative l'effet bénéfique de la mise en place des revues sur la qualité du code développé et la productivité de l'équipe, autant les réviseurs, les auteurs que les directeurs sont unanimes sur le fait que la mise en place des revues était nécessaire et qu'il est pertinent de poursuivre l'expérience.

7.1 Revue des objectifs et des hypothèses

7.1.1 Objectifs

En se référant aux objectifs qui ont justifié le démarrage du projet, quelles sont les conclusions qui peuvent en être tirées?

Objectif 1 : Assurer que les bonnes pratiques de sécurité du code soient systématiquement appliquées et vérifiées.

Cet objectif est en partie atteint grâce à la mise en place d'éléments de vérification de sécurité qui doivent être suivis tant par les auteurs que les réviseurs. Durant l'initiative, aucun défaut relatif à la sécurité des applications n'a été découvert en production, ce qui est un bon signe. Toutefois, une plus grande rigueur dans les inspections, surtout par rapport à la vitesse et à la fréquence de celles-ci, contribuera à une meilleure détection des problèmes de sécurité potentiels. Ceux-ci devront également être adéquatement documentés pour être rapidement intégrés à la liste de vérification et faire l'objet d'une attention particulière lors des revues subséquentes.

Objectif 2 : Réduire l'effort de correction de défauts (*rework*).

Comme discuté précédemment, il est impossible pour le moment de conclure quoi que ce soit pour cet objectif étant donné les écarts importants dans la qualité des données nécessaires à la mesure. Bien que l'expérience vécue par les membres de l'équipe semble indiquer que le nombre de défauts est réduit par les revues, seules l'uniformisation des données et la poursuite des mesures pourront confirmer (ou infirmer) cet objectif de façon empirique.

Objectif 3 : Augmenter la qualité interne du code et des documents pour en réduire l'effort de maintenance.

L'initiative ayant été d'une durée relativement courte (environ six mois), peu d'applications ont connu assez de cycles de maintenance pour permettre de confirmer si le coût relatif des maintenances en heure par PFC diminuera avec l'augmentation progressive de la qualité du code. Les premiers résultats sont encourageants, mais ici encore, la poursuite de la collecte de données et des mesures permettront éventuellement de déterminer l'atteinte de cet objectif.

Objectif 4 : Assurer un transfert de connaissances sur les applications.

Suite au sondage réalisé auprès des développeurs vers la fin du projet, le transfert de connaissances au sujet des applications semble effectivement bien se faire grâce aux revues. Comme spécifié dans la section 5.3.2, 100 % des répondants au sondage ayant participé à une revue comme réviseur affirment avoir acquis des connaissances sur le fonctionnement d'applications qui leur étaient alors inconnues ou dont ils ne connaissaient que très peu de choses. À la lumière de ces résultats, il est raisonnable d'affirmer que cet objectif a été atteint.

Objectif 5 : Partager les connaissances et les compétences techniques.

Toujours suite au sondage de satisfaction, 100 % des réviseurs affirment avoir partagé des connaissances techniques avec l'auteur du code. 100 % des auteurs ont quant à eux répondu avoir appris des commentaires reçus. De ceux-ci, 58,4 % affirment en avoir fait l'expérience souvent ou très souvent. De plus, les commentaires du sondage permettent de constater que l'apprentissage de nouvelle façon de faire semble être une des retombées les plus appréciées des participants. Ces données démontrent que le partage de connaissances techniques semble bel et bien se faire entre les membres de l'équipe et donc, que l'objectif peut être considéré comme atteint.

Objectif 6 : Uniformiser les pratiques de développement entre les différents projets.

Finalement, selon les développeurs sondés, les différents processus, outils et documents mis en place lors du projet permettent de rendre le processus de développement plus uniforme. Parmi ces éléments, la liste de vérification et le processus de revue documenté sont ceux qui y contribuent le plus selon le sondage de satisfaction. Toutefois, même si les développeurs considèrent que ces éléments favorisent un processus de développement plus uniforme, l'initiative n'a pas été en mesure de confirmer empiriquement l'atteinte de cet objectif, que ce soit au niveau de la qualité du code (densité de défauts, temps de *rework*) ou de la productivité (heure par PFC). Les prochains projets permettront de vérifier si les outils mis en place auront un effet mesurable sur la prévisibilité et l'uniformité des pratiques de développement.

7.1.2 Hypothèses

À la section 3.3, diverses hypothèses ont été avancées. Est-ce que certaines d'entre elles ont pu être confirmées?

H1. La revue par les pairs diminuera le nombre de défauts trouvés lors des tests intégrés (par l'équipe de QA et lors des tests utilisateurs) et en production.

Comme discuté précédemment, cette hypothèse n'a pas pu être confirmée à partir des données recueillies. La poursuite des mesures sur les projets actuels et les travaux futurs permettront peut-être de confirmer cette hypothèse éventuellement.

H2. L'effort de revue sera compensé au moins entièrement par une réduction du coût de *rework*. La productivité de l'effort de développement et de maintenance (en heure par PFC) devra demeurer stable et idéalement s'améliorer une fois le processus de revue mis en place et après une période d'apprentissage par les membres de l'équipe.

Ici encore, l'hypothèse n'a pas pu être confirmée à partir des mesures. Par contre, les données indiquent que le taux de détection est actuellement faible ce qui pourrait expliquer pourquoi l'effort de revue n'est pas entièrement compensé par une réduction de *rework*. Il est toutefois possible de présumer que lorsque l'équipe aura acquis une plus grande expérience avec les revues, cette hypothèse pourrait se confirmer.

H3. Le partage de connaissances sur les applications et les compétences techniques améliorera la satisfaction des développeurs en leur permettant d'acquérir de nouvelles connaissances qui les rendront plus productifs. De plus, l'apprentissage de nouvelles façons de faire pourra leur donner des idées nouvelles sur la façon d'attaquer et de régler les problèmes qui se présenteront à eux à l'avenir.

Cette hypothèse semble se confirmer. Même si une augmentation de la productivité suite au transfert de connaissance n'a pas encore été constatée, les résultats de sondage indiquent que les développeurs ont appris et partagés de nouvelles connaissances et compétences grâce aux revues et qu'ils sont dans l'ensemble satisfaits de l'expérience, au point de vouloir la pousser plus loin.

7.2 Recommandations et travaux futurs

De nombreux apprentissages ont été faits durant le projet. Il est important de mettre à profit les succès obtenus et de poursuivre les efforts pour continuer à progresser vers les objectifs qui restent à atteindre. Pour ce faire, les recommandations suivantes sont proposées :

- Continuer de mettre l'accent sur la formation et la sensibilisation de tous les membres de l'équipe sur l'importance de la rigueur dans tous les aspects du processus :
  - réalisation des revues;
  - identification, documentation et classification des défauts;
  - corrections des défauts;
  - feuilles de temps précises avec les activités correctement identifiées.

  C'est seulement grâce à des mesures fiables et régulières que l'équipe pourra identifier les écarts dans le processus, y apporter les corrections nécessaires et en mesurer les effets.
- Donner suite aux commentaires recueillis dans le sondage en préparant des formations sur SonarQube pour mieux en expliquer le fonctionnement, les possibilités et les limites. Il faudra également insister sur l'importance d'utiliser les mesures offertes pour améliorer la qualité des logiciels et enseigner les techniques pour y arriver.
- Mettre en place des revues documentées et systématiques des documents d'exigences et des spécifications. La caractérisation des défauts indique qu'une part significative de ceux-ci provient des spécifications et qu'il faudrait donc s'y attarder. De plus, la mise en place des revues de spécifications s'intègrera avec la revue des estimations déjà en cours.
- Insister auprès des membres de l'équipe sur l'importance de la qualité des artéfacts et de son impact sur leurs coéquipiers, le CIRX et l'entreprise en général. Il faut les encourager à utiliser les bonnes pratiques déjà en place (comme les tests unitaires automatisés) et celles développées lors de l'initiative comme les listes de vérification. Il est important d'insister sur l'importance d'utiliser les outils mis en place (comme SonarQube par exemple) pour identifier de façon proactive les problèmes potentiels de qualité et en prendre acte le plus rapidement possible. Il faut également que la direc-

tion soutienne l'équipe dans la réalisation de cet objectif, non seulement en rappelant son importance, mais aussi en donnant aux membres de l'équipe des moyens de l'atteindre.

- Suivre la progression des mesures de façon régulière (au moins une fois par mois) pour l'ensemble des projets en cours de réalisation ou pour un sous-ensemble de projets qui auront été préalablement identifiés pour s'assurer que les revues sont bien réalisées, que les défauts sont bien documentés et que l'équipe progresse dans la qualité de ses applications.
- Mettre en place des tableaux de bord automatisés. Durant l'initiative, les différentes sources de données nécessaires à la création du tableau de mesure étaient accédées et consolidées manuellement par le responsable de l'initiative. Il s'agit toutefois d'une tâche fastidieuse qui demande quelques heures de travail par mois. Comme pour tout processus manuel, il est arrivé à l'occasion que des erreurs dans la compilation des données soient détectées et qu'il faille recommencer une partie de la cueillette d'information. De plus, l'effort de prise de mesure peut être encore plus important si une grande quantité de projets sont en cours de réalisation en même temps. La mise en place de rapports ou de tableaux de bord automatisés permettrait un accès beaucoup plus simple et rapide aux informations clés des projets. Ces indicateurs seraient très utiles pour toutes les parties prenantes (membre de l'équipe, direction) pour suivre l'évolution des projets dans le temps. Évidemment, comme pour la collecte de données manuelle, la notion de confidentialité des personnes demeure un enjeu à ne pas négliger. Le tableau de bord est un outil de suivi d'amélioration continue, pas un outil d'identification de coupables ou d'évaluation du rendement individuel.
- Réviser le processus de revue régulièrement. Les membres de l'équipe doivent s'exprimer sur le processus de revue, faire des recommandations ou des suggestions pour l'améliorer, le simplifier ou le rendre plus efficace. Ils ont la possibilité de le faire à tout moment grâce à la liste Sharepoint de suggestions d'améliorations aux revues et les suggestions apportées pourront être discutées lors des rencontres mensuelles de développeurs. À noter qu'un irritant majeur, l'arrivée d'un nouvel outil ou une nouvelle technologie pourrait aussi amener l'équipe à faire des améliorations en

tout temps. Le processus n'est pas coulé dans le béton. Au contraire, il doit être dynamique et doit pouvoir évoluer rapidement si l'équipe le désire. Celle-ci doit rester en contrôle de son processus de revue et elle doit être consultée avant la mise en place de changements importants. Il faut conserver l'acceptabilité du processus sans quoi, les risques de contournement ou de désaffection sont élevés.

- Poursuivre la révision des mesures d'estimation COSMIC qui a été amorcée pour améliorer la constance des mesures de taille fonctionnelle entre les projets et les analystes. C'est un prérequis essentiel pour les mesures de productivité et de qualité et c'est seulement une fois ce travail complété qu'il sera possible de vérifier si l'impact du processus de revue est significatif. Comme cité plus haut, cette étape pourra être intégrée à un processus plus global de revue des documents de spécifications.
- Évaluer la possibilité d'assigner un minimum de deux développeurs sur chaque projet, exception faite des très petites maintenances. L'interaction avec un collègue sur un projet permet généralement une meilleure revue puisque moins de temps est consacré à la mise en contexte. Le réviseur peut alors mieux se concentrer sur la détection de problèmes potentiels. En ayant plus d'une personne sur un projet, celles-ci peuvent travailler ensemble pour résoudre des problèmes ou discuter des solutions mises en place pour s'assurer qu'elles sont adéquates. Évidemment, les contraintes opérationnelles peuvent rendre cette recommandation plus difficile à appliquer, car la conséquence immédiate serait un moins grand nombre de requêtes traitées en parallèle par l'équipe. Est-ce qu'un moins grand nombre de projets de front avec plus de personnes par projet serait plus efficace pour les revues, les apprentissages et le transfert de connaissances? Il serait intéressant d'évaluer cette option lors de projets futurs.

7.3 Plan de déploiement des pratiques de revues pour les autres équipes du CIRX

Un des besoins exprimés par la direction lors du démarrage de l'initiative est la généralisation des pratiques de revues à toutes les équipes de développement de l'entreprise. Suite aux expériences acquises lors de la mise en place des revues par l'équipe Web-Micro, voici une liste des différentes étapes nécessaires pour appliquer les revues dans une nouvelle équipe de développement :

1. Rencontrer les gens de l'équipe concernée et déterminer le problème à résoudre. Est-ce que l'équipe veut régler un problème de qualité, de productivité, de partage d'information? Recueillir leurs commentaires ou leurs suggestions. Déterminer des objectifs qu'ils aimeraient atteindre. Autant que possible, ces objectifs devraient être mesurables.
2. Mettre en place un plan de mesure afin de déterminer les différents indicateurs nécessaires pour déterminer l'avancement de l'équipe vers l'objectif.
3. Définir le processus de revue. Pour obtenir du succès, celui-ci doit :
    a. Être simple d'application
    b. Rigoureux
    c. Connu et approuvé par tous les membres de l'équipe
    d. Se faire dans le respect le plus total
    e. Être documenté…
    f. … et mesuré
4. Présenter le processus aux membres de l'équipe pour recueillir leurs commentaires, leurs suggestions et obtenir l'approbation de ceux-ci. C'est une étape capitale à la réussite d'une initiative comme celle-là.
5. Si c'est nécessaire (à déterminer selon le processus établi), rechercher, acquérir et installer les outils qui pourraient être nécessaires au bon fonctionnement du processus de revue. Si de nouveaux outils sont mis en place, bien documenter les fonctionnalités requises pour le processus de revue.

6. Former les différents intervenants (développeurs, analystes, directeurs) sur les tâches qu'ils auront à accomplir. La formation doit s'accompagner de documentation écrite puisqu'il arrive régulièrement qu'une personne ne mette en application les connaissances acquises que plusieurs semaines (ou même plusieurs mois) après la formation. Dans certains cas, une séance pour rafraichir les mémoires pourrait être nécessaire. Il est important qu'une partie de la formation soit consacrée au savoir-être et à l'aspect humain de la revue, car c'est généralement ce qui déterminera si la mise en place des revues sera un succès ou non.
7. Utiliser les données recueillies lors des revues pour mettre à jour les différents indicateurs de façon régulière. Cela permet non seulement de visualiser la progression de l'équipe vers leurs objectifs, mais aussi de déceler des problèmes dans la mise en place du processus de revue (mesures irrégulières, processus mal compris, problème avec les outils, etc.).
8. Après quelque temps (environ six mois), faire une rétroaction pour recueillir les commentaires de l'équipe sur l'expérience vécue avec le processus de revue et faire les ajustements nécessaires.

Bien que ces étapes puissent sembler très génériques, il est malheureusement difficile de réussir la mise en place d'un processus aussi épineux que la revue si celui-ci n'est pas adapté aux valeurs et aux façons de faire de l'équipe. Tout l'outillage du monde ne remplacera pas le respect des opinions de chacun et la collaboration sincère des individus pour obtenir des résultats positifs. Il est aussi important de se rappeler que les revues sont une pratique du processus de développement qui a fait ses preuves et les équipes qui veulent obtenir du succès de façon récurrente devraient considérer les mettre en place. Chaque objectif atteint devient alors une occasion d'en définir de nouveaux, différents ou plus ambitieux, afin de permettre à l'entreprise de demeurer compétitive dans un environnement technologique aux changements toujours plus rapides. Dans un monde agile où il faut souvent faire plus avec moins, il s'agit d'une condition essentielle pour permettre à CIRX de continuer de répondre aux attentes de ses clients maintenant et pour les années à venir.

ANNEXE I

PLAN DE PROJET

Plan de projet

1.1 Vision du projet

Le but de cette initiative est d'évaluer le processus de revue de code par les pairs. Par cette évaluation, la direction du CIRX cherche à déterminer ce qui fonctionne bien dans le processus actuel, ce qui est déjà fait, mais qui pourrait être amélioré et si des étapes dans le processus sont manquantes.

1.2 Les objectifs du projet

Réviser le processus de revue de code dans le but :

- de réduire le nombre de défauts trouvés par les analystes lors des tests intégrés et les utilisateurs finaux lors de la livraison du produit
- d'augmenter la qualité interne du code pour réduire les efforts de maintenance.
- Transfert de connaissance sur les applications et les compétences techniques
- Uniformiser les pratiques de développement

La direction espère pouvoir obtenir des résultats tangibles et mesurables de ses améliorations au courant de l'année 2017.

1.3 Les critères de succès

- Mise en place d'un processus de revue simple, efficace, utilisé de façon systématique pour tous les développeurs
- Réduction du nombre de défauts trouvés lors des tests et des livraisons
- Réduction de l'effort de maintenance

1.4 Les leviers

Leviers	

Portée	X
Budget	
Échéancier	

1.5 Gestion des risques

Les risques prévus pour cette initiative sont les suivants :

- Changement dans les priorités de l'entreprise : il se pourrait que des projets plus prioritaires ralentissent le projet. Dans ce cas, il serait bon d'informer la direction de la situation, de leur rappeler l'importance du projet et si nécessaire, négocier un nouvel échéancier
- Manque de collaboration de l'équipe : il se peut que l'équipe soit réfractaire à un changement dans leurs habitudes de travail et il est possible qu'ils voient la revue de code comme une perte de temps ou un manque de confiance en leurs compétences. Il faudra alors expliquer le but de l'initiative, donner le temps à tout un chacun de se familiariser avec le processus et prouver (ou réfuter) à l'aide de mesures que le changement est bénéfique (ou non).

1.6 Échéancier

Voici une liste des différentes étapes de l'initiative avec une estimation des heures requises et un aperçu des échéances prévues pour chacune de ces étapes :

Étape	**Durée estimé (en h)**	**Date échéance prévue**
Démarrage du projet	15h	Fait
Rencontre de la direction pour approbation de la vision du projet, des objectifs et de l'échéancier	5h	Octobre 2016
Analyse des rapports de revues existants	21h	Octobre 2016
Préparer et documenter le processus de revue	70h	Octobre 2016
Terminer l'installation des outils d'analyse	21h	Octobre 2016

Préparer et faire la rencontre des membres de l'équipe pour présenter le processus de revue.	7h	Octobre / Novembre 2016
Réalisation des projets, des revues et collecte des données	Environ 1 heure par projet par semaine	Novembre 2016- mars 2017
Analyse des résultats et révision du processus	35h	Avril 2017
Rédaction du rapport	105h	Avril à juin 2017
Remise du rapport		Septembre 2017
Total estimé	210h + temps de réalisation	

1.6.1 Hypothèses

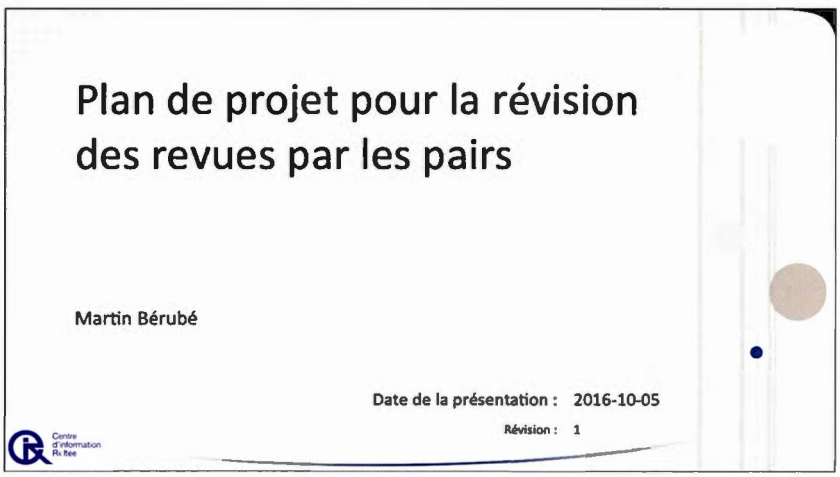
Cet échéancier a été rédigé en tenant compte des hypothèses suivantes :

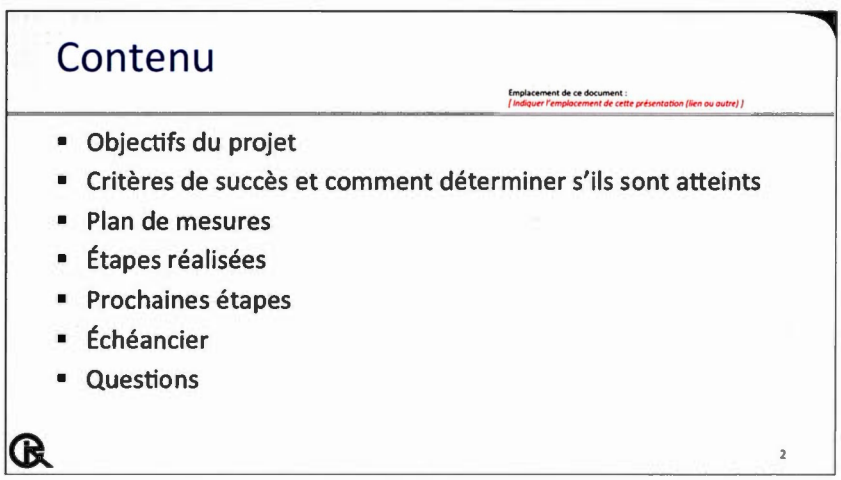
- L'architecte pourra dédier 1 à 2 journées par semaine au début de l'initiative. Durant la phase d'expérimentation, ½ journée par semaine sera nécessaire. Lors de la phase d'analyse et de rédaction, l'effort prévu est d'un à deux jours par semaine.
- La phase d'expérimentation se fera sur au moins 3 projets (idéalement 5) de taille petits à moyen.
- Le temps consacré aux revues par les membres de l'équipe sera probablement plus long que celui actuellement dédié à cette tâche. Le processus sera plus rigoureux et il y aura certainement une courbe d'apprentissage lors des premières revues.

ANNEXE II

PRÉSENTATION À LA DIRECTION

17-08-19





Quelques définitions ...

- Défauts : Élément incorrect dans un artéfact (document ou code)
- Phase d'injection : phase (définition des besoins (I01), analyse fonctionnelle (D16) ou développement) où le défaut a été introduit.
- Phase de détection : phase où le défaut a été détecté. Plus un défaut est détecté « loin » de sa phase d'injection, plus il est coûteux de le corriger.
- Temps de rework : temps nécessaire pour documenter, corriger et retester un défaut.
- PFC : Point de fonction COSMIC

3

Objectifs du projet

- Assurer que les bonnes pratiques de sécurité du code soient systématiquement appliquées et vérifiées
- Réduire l'effort de corrections de défauts trouvés par les analystes lors des tests intégrés et les clients (business et grand public)
- Augmenter la qualité interne du code et des documents pour réduire l'effort de maintenance
- Transfert de connaissances sur les applications et les compétences techniques
- Uniformiser les pratiques de développements (standards et bonnes pratiques)

4

Critère de succès

- Mise en place d'un processus de revue qui soit simple, efficace et systématique
- Réduction du nombre de défauts trouvés lors des tests et des livraisons
- Réduction de l'effort de maintenance (réduire le nombre d'heure par PFC)

5

Comment déterminer si les critères de de succès sont atteints par rapport aux objectifs ?

Objectif général : augmenter la productivité (c'est-à-dire réduire le nombre d'heure par PFC)

Objectifs spécifiques

- S'assurer de la sécurité du code

Pas de mesures spécifiques

- Réduire l'effort de rework :

L'effort de revue doit être inférieur à l'effort de rework évité

- Réduction de l'effort de maintenance

Pas de mesures spécifiques

- Transfert de connaissances sur les applications et les technologies

Augmentation du niveau général de connaissance de l'équipe

- Uniformiser les pratiques de développement

Réduction du nombre de défauts

6

Plan de mesure (1/4)

Mesures de qualité des documents et logiciels

- Nombre de défauts par phase d'injection et de détection (TFS)
- Index de maintenabilité (automatisé)
- Évolution de la couverture de code (automatisé)
- Temps moyen pour la correction des défauts (calculé)

Utilisation :

- Découvrir les endroits où le plus grand nombre de défauts est injecté
- Voir si l'amélioration du processus améliore la qualité du code et réduit le temps de correction et de maintenance

7

Plan de mesure (2/4)

Mesures de qualité des revues

- Ratio du nombre de défaut détectés dans une phase subséquente à la phase d'injection (calculé)
- Nombre d'heures passées en revue par phase (feuille de temps)

Utilisation :

- Découvrir les endroits les revues devraient être amélioré (des défauts sont passés à travers le filtre).
- Calculer la productivité du processus de revue

8

Plan de mesure (3/4)

Mesures de productivité

- Heure par PFC (mesure déjà existante)
- Évolution de l'effort de revue par PFC (calculé)
- Pourcentage de l'effort de revue sur le temps total du projet (calculé)
- Nombre d'heures passés en correction (TFS)

Utilisation :

- Déterminer si l'effort relatif de révision baisse ou augmente avec le temps
- Déterminer si une meilleure qualité de code augmente la productivité générale de l'équipe (heure / PFC)

9

Plan de mesure (4/4)

Mesures du transfert de connaissance

C'est quelque chose de difficile à mesurer quantitativement donc, vers la fin du projet, un cours sondage sera réalisé pour déterminer la satisfaction des membres de l'équipe sur le processus de revue en général mais aussi sur leur impression sur les transferts de connaissances, autant du côté technique que des applications.

Utilisation :

- Déterminer le degré de satisfaction de l'équipe et obtenir des suggestions pour améliorer le processus de revues.

Le plan de mesure détaillé est disponible dans le site de projet (ici).

10

Effort requis par l'équipe pour la prise de mesure

- Documenter correctement (et systématiquement) les bogues lors de la création de ceux-ci en entrant les informations suivantes :
  - Phase d'injection
  - Phase de détection
  - Temps estimé de correction (si connu)
  - Temps de documentation du bogue *
- Documenter le temps de correction lorsque le défaut est corrigé et que le bogue est retourné en test *
- Documenter le temps de vérification passé à tester ou revoir la correction *
- Inscrire le temps passé à faire de la revue dans la feuille de temps

* Le temps est cumulatif. Si le bogue retourne en correction, on continue d'ajouter le temps passé. Tous les temps sont en heure, arrondi au 1S minutes

11

Étapes réalisées

- Plan de mesure élaboré
- Mise en place de TFS2015 pour faciliter la collecte des données nécessaires aux mesures
- Les développeurs qui ont fait des revues dans les derniers mois ont remplis des fiches de revues. Ce sera utile pour déterminer les éléments les plus problématiques à inclure dans le « checklist » des éléments à revoir.

12

Prochaines étapes

- Présentation du plan de projet et approbation par la direction
- Sélection des projets pour tester les hypothèses. Idéalement 6 à 10 projets assez typiques (entre 10-30 PFC). 50% sans le processus de revue, 50% avec le processus de revue
- Documenter le processus de revues et la collecte de données
- Rencontre des membres de l'équipe pour leur présenter la méthode de revue. Dépendant du nombre d'éléments à présenter, il se pourrait que 2 rencontres soient nécessaires (une pour les analystes et une pour les développeurs)
- Terminer l'installation des outils de collecte (SonarQube)
- Réaliser les projets, faire les revues et collecter les mesures
- Analyse des données
- Rédaction et remise du rapport

13

Échéancier

Étape	Durée estimé (en h)	Date échéance prévue
Rencontre de la direction	5h	Octobre 2016
Analyse des rapports de revues	21h (3jrs)	Octobre 2016
Préparer et documenter le processus de revue	70h (10jrs)	Octobre 2016
Terminer l'installation des outils d'analyse	21h (3jrs)	Octobre 2016
Préparer et faire la rencontre des membres de l'équipe pour présenter le processus de revue	7h (1jr)	Octobre / Novembre 2016
Réalisation des projets, des revues et collecte des données	Environ 1h par projet par semaine	Novembre 2016 – Mars 2017
Analyse des résultats	35h (5jrs)	Avril 2017
Rédaction du rapport	70h (10jrs)	Avril – Juin 2017
Remise du rapport, présentation des résultats, suivi		Septembre 2017
Temps total estimé	**210h (+/- 30 jrs) + temps de réalisation**	

14

Questions pour les directeurs

- Création d'un projet pour cette initiative (où dois-je mettre mon temps) ?
- Comment les membres de l'équipe pourront identifier le temps de revue dans leurs feuilles de temps ? Il n'y a plus d'activité « Revue de code » dans Kronos.
- Qui fera l'extraction des temps de feuille de temps (moi ou les directeur) ?

15

Questions

- Commentaires
- Suggestions

16

ANNEXE III

PLAN DE MESURE

3.2 Détail des mesures

3.2.1 Nombre de défauts par phase de détection

Mesure de base					
#	**Mesure**		**Applicabilité**	**UDM**	**Précision**
1	Nombre de défauts par phase de détection		Projet ou DDC	Unité	Unité
Qui mesure ?	**Source de la mesure**	**Où stocker le résultat**	**Outils**	**Moment**	
Responsable du projet (Directeur, architecte ou analyste)	Gestionnaire de bogues (TFS)	Excel ou base de données	Rapport personnalisé dans TFS	À la fin du projet ou de la DDC	
Procédure de collecte		**Assurance qualité**			
Extraire tous les défauts (type d'item de travail == bogue) pour un projet ou une DDC donné, regroupé par phase de détection.		Faire des extractions manuelles lors des premiers rapports pour s'assurer que le rapport est bien calibré et que tous les défauts voulus s'y trouvent			
		Schèmes d'interprétation			
		Idéalement, il faut minimiser le nombre de défauts à la source. Cependant, il est plus optimal de trouver un maximum de défauts dans les premières phases du projet et moins vers les dernières étapes			

3.2.2 Nombre de défauts par phase d'injection

Mesure de base					
#	**Mesure**		**Applicabilité**	**UDM**	**Précision**
2	Nombre de défauts par phase d'injection		Projet ou DDC	Unité	Unité
Qui mesure ?	**Source de la mesure**	**Où stocker le résultat**	**Outils**	**Moment**	
Responsable du projet (Directeur, architecte ou analyste)	Gestionnaire de bogues (TFS)	Excel ou base de données	Rapport personnalisé dans TFS	À la fin du projet ou de la DDC	
Procédure de collecte		**Assurance qualité**			
Extraire tous les défauts (type d'item de travail == bogue) pour un projet ou une DDC donné, regroupé par phase d'injection.		Faire des extractions manuelles lors des premiers rapports pour s'assurer que le rapport est bien calibré et que tous les défauts voulus s'y trouvent			
		Schèmes d'interprétation			
		Idéalement, il faut minimiser le nombre de défauts à la source. Cependant, identifier la phase où sont injectés le plus grand nombre de défauts permet de réviser les processus de test et de revue pour cette phase.			

3.2.3 Évolution de l'index de maintenabilité dans le temps

Mesure de base					
#	**Mesure**		**Applicabilité**	**UDM**	**Précision**
3	Évolution de l'index de maintenabilité dans le temps		Application	Unité	2 décimales
Qui mesure ?	**Source de la mesure**	**Où stocker le résultat**	**Outils**	**Moment**	
Responsable du projet (Directeur, architecte ou analyste)	SonarQUBE maintenability index	Excel ou base de données	SonarQUBE	À la fin du projet, de la DDC et révision globale tous les 3 mois	
Procédure de collecte		**Assurance qualité**			
Utilisation de l'outil pour extraire l'index de maintenabilité des applications en 2 temps : celui en date du jour et celui à la dernière révision (version précédente dans le cas des vérifications en fin de projet, au dernier trimestre dans le cas de la révision globale)		Le calcul des index de maintenabilité étant assez complexe (voir par exemple la façon dont SonarQUBE le calcule[1]), la façon la plus simple de faire l'assurance qualité est de mettre en relation 2 applications : une qui est réputée bien se maintenir et une autre de qualité moindre pour vérifier si l'index de maintenabilité suit la condition de représentation (une application de moindre qualité devrait normalement avoir un index de maintenabilité moindre).			
		Schèmes d'interprétation			
		L'index de maintenabilité est une mesure plutôt abstraite. Ce qu'on souhaite avec cette mesure, c'est d'identifier si la maintenabilité générale de l'application s'améliore ou au minimum reste stable dans le temps suite à la mise en place des revues.			
Notes ou commentaire					
Le client accepte que le produit soit installé sur son infrastructure.					

3.2.4 Évolution de la couverture de tests unitaires du code

Mesure de base					
#	**Mesure**		**Applicabilité**	**UDM**	**Précision**
4	Couverture de tests unitaires du code		Application	Pourcentage	2 décimales
Qui mesure ?	**Source de la mesure**	**Où stocker le résultat**	**Outils**	**Moment**	
Responsable du projet (Directeur, architecte ou analyste)	TFS	Excel ou base de données	TFS	À la fin du projet, de la DDC.	
Procédure de collecte		**Assurance qualité**			
Pour une application donnée, aller chercher le pourcentage de couverture de code de la version précédente pour la comparer à la couverture de la nouvelle version.		La couverture de code est calculée automatiquement par un outil qui a déjà été calibré par l'équipe de développement donc il est tenu pour acquis que cette information est fiable.			
		Schèmes d'interprétation			
		L'idéal est avoir un pourcentage de couverture de code le plus élevé possible, mais il est illusoire de viser 100 %. Il faut plutôt s'assurer d'une couverture globale d'au moins 50 % et d'au moins 80 % dans les logiques d'affaires. Il est souhaitable de voir la couverture rester stable ou croître dans le temps.			

3.2.5 Temps moyen pour la correction d'un défaut

Mesure de base					
#	**Mesure**		**Applicabilité**	**UDM**	**Précision**
5	Temps moyen pour la correction d'un défaut		Toutes les applications	Heures	15 minutes
Qui mesure ?	**Source de la mesure**	**Où stocker le résultat**	**Outils**	**Moment**	
Responsable du projet (Directeur, architecte ou analyste)	TFS	Excel ou base de données	TFS	À tous les trimestres.	
Procédure de collecte		**Assurance qualité**			
Pour l'ensemble des défauts corrigés, aller chercher le temps nécessaire à la correction et faire la moyenne sur l'ensemble des défauts		Le calcul peut être fait manuellement au début pour s'assurer que les valeurs sont valides. Il faudra également vérifier périodiquement que les données de temps de correction sont bien entrées dans l'outil de gestion des défauts.			
		Schèmes d'interprétation			
		Avec l'amélioration de la qualité du code, le temps moyen de correction des défauts devrait diminuer. Comme il a des défauts qui sont plus difficiles à corriger que d'autres, il faut juger de l'amélioration sur un assez grand nombre de défauts pour pouvoir démontrer une tendance.			

3.2.6 Ratio du nombre de défauts détectés dans une phase subséquente à la phase d'injection

Mesure dérivée ou indicateurs					
#	**Mesure**	**Formule**		**UDM**	**Précision**
6	Ratio du nombre de défauts détectés dans une phase subséquente à la phase d'injection	Nombre de défauts par phase d'injection où la phase de détection est différente de la phase d'injection divisé par le nombre total de défauts pour la phase d'injection pour le projet ou la DDC		Pourcentage	2 décimales
Responsable	**Consommateur**	**Où stocker le résultat**	**Présentation des résultats**	**Fréquence**	
Responsable du projet (Directeur, architecte ou analyste)	Le responsable	Excel ou base de données	Diagramme de secteur pour l'évaluation en fin de projet et diagramme de zone pour l'évolution dans le temps (voir section 0)	À la fin du projet, de la DDC.	
Procédure d'analyse		**Action potentielle selon les résultats**			
Comparer les différents ratios pour identifier la phase où le ratio est le plus élevé.		Revoir les processus de revue et de développement pour la phase où le ratio est le plus élevé pour tenter d'améliorer la qualité à la source et aussi améliorer la revue à cette phase.			

3.2.7 Nombre d'heures passées en revue par phase

Mesure de base					
#	**Mesure**		**Applicabilité**	**UDM**	**Précision**
7	Nombre d'heures passées en revue par phase		Projet ou DDC	Heures	15 minutes
Qui mesure ?	**Source de la mesure**	**Où stocker le résultat**	**Outils**	**Moment**	
Responsable du projet (Directeur, architecte ou analyste)	Kronos (outil de feuille de temps)	Excel ou base de données	Excel	À la fin du projet, de la DDC.	
Procédure de collecte		**Assurance qualité**			
Pour une DDC ou un projet, extraire le nombre d'heures passées en revue par les pairs		L'assurance qualité devra se faire à deux niveaux : par l'éducation du personnel et la surveillance des feuilles de temps au début et par la suite, en appliquant des ratios de pourcentage de temps passé en revue sur la durée totale du projet pour estimer si les chiffres obtenus ont du sens.			
		Schèmes d'interprétation			
		On veut un chiffre plus grand que zéro et qui semble respecter la condition de représentation (plus d'heures pour un gros projet qu'un petit).			

3.2.8 Heures / PFC

Mesure dérivée ou indicateurs						
#	**Mesure**		**Formule**		**UDM**	**Précision**
8	Heures / PFC		Nombre d'heures passées sur le projet ou la DDC divisé par la taille fonctionnelle de l'application ou de la DDC.		Heures	1 décimale
Responsable		**Consommateur**	**Où stocker le résultat**	**Présentation des résultats**	**Fréquence**	
Responsable du projet (Directeur, architecte ou analyste)		Le responsable	Excel ou base de données	Chiffre dans un chiffrier	À la fin du projet, de la DDC.	
Procédure d'analyse			**Action potentielle selon les résultats**			
Vérifier si la mise en place des revues augmente la productivité de l'équipe de développement en réduisant le nombre d'heures par PFC.			C'est la mesure ultime de la productivité. C'est grâce à ce chiffre que l'équipe va savoir si le nombre d'heures investies dans les revues augmente ou non la productivité globale de l'équipe et de combien.			
Notes ou commentaire						
Cette mesure est déjà prise par les équipes de développement de CIRX						

3.2.9 Évolution de l'effort de révision par PFC

Mesure dérivée ou indicateurs						
#	**Mesure**		**Formule**		**UDM**	**Précision**
9	Effort total de révision par PFC		Nombre de minutes passées à faire de la révision. Le temps de révision inclut : • le temps de revue • les tests • le rework dû aux défauts non détectés lors des revues. On prend le nombre d'heures consacrées à chacune des activités divisé par la taille fonctionnelle de l'application ou de la DDC.		Heure par PFC	2 décimales
Responsable		**Consommateur**	**Où stocker le résultat**	**Présentation des résultats**	**Fréquence**	
Responsable du projet (Directeur, architecte ou analyste)		Le responsable	Excel ou base de données	Histogramme en fin de projet et nuage de points pour l'évolution dans le temps. Voir section 3.2.13	À la fin du projet, de la DDC.	
Procédure d'analyse			**Action potentielle selon les résultats**			
Vérifier si la mise en place des revues diminue d'effort de révision par PFC.			C'est par cette mesure qu'on pourra déterminer si l'objectif de réduction de l'effort de révision est atteint. Il faudra s'assurer que la réduction de l'effort de test et de rework n'est pas entièrement remplacée par de l'effort de revue sans quoi, le résultat net de la mise en place des revues dans un objectif de réduction de l'effort global de révision sera nul.			

3.2.10 Pourcentage d'effort de révision sur le total du projet

Mesure dérivée ou indicateurs						
#	**Mesure**		**Formule**		**UDM**	**Précision**
10	Pourcentage de l'effort de révision sur le total du projet		Effort de révision / effort total du projet		Pourcentage	2 décimales
Responsable		**Consommateur**	**Où stocker le résultat**	**Présentation des résultats**	**Fréquence**	
Responsable du projet (Directeur, architecte ou analyste)		Le responsable	Excel ou base de données	Diagrammes de zone (général et détaillé). Voir section 3.2.14	À la fin de chaque projet ou DDC, une fois par mois maximum.	
Procédure d'analyse			**Action potentielle selon les résultats**			
Vérifier si la mise en place des revues diminue d'effort relatif de révision dans l'effort total des projets			Au fur et à mesure que des projets ou des DDC sont livrés, il faut vérifier si l'équipe a tendance à s'améliorer et à réduire le temps de révision ou s'il reste stable (ou pire, si le temps de révision augmente). Si le temps de révision augmente, investiguer les raisons pour vérifier s'il s'agit d'un événement ponctuel ou si les processus de développement doivent être révisés.			

3.2.11 Nombre d'heures passées en correction (rework)

Mesure de base						
#	**Mesure**			**Applicabilité**	**UDM**	**Précision**
11	Nombre d'heures passées en correction			Projet ou DDC	Heures	15 minutes
Qui mesure ?		**Source de la mesure**	**Où stocker le résultat**	**Outils**	**Moment**	
Responsable du projet (Directeur, architecte ou analyste)		Kronos (outil de feuille de temps) ou TFS	Excel ou base de données	Kronos ou TFS	À la fin du projet, de la DDC.	
Procédure de collecte			**Assurance qualité**			
Pour un projet ou une DDC, extraire les bogues de l'outil de gestion de bogues, aller chercher la durée de la correction dans l'attribut « Actual Effort » et en faire la somme.			Une extraction manuelle des informations pourra être faite pour confirmer si les chiffres obtenus sont réalistes.			
			Schèmes d'interprétation			
			On veut un chiffre le plus petit possible car cette valeur représente le rework c'est-à-dire le travail qui doit être refait car un défaut a été introduit et il n'a pas été détecté par l'auteur ni le ou les réviseurs. Il a donc été attrapé plus tard dans le processus de développement et un défaut doit être ouvert pour que l'auteur fasse les corrections nécessaires.			
Notes ou commentaire						
Confirmer avec le client l'endroit où il préfère collecter l'information du temps de correction (outil de feuille de temps ou dans les attributs du défaut)						

ANNEXE IV

LISTE DE VÉRIFICATION

Checklist code review

Web-Micro, version 1, octobre 2016

Sécurité	
☐ Injection	☐ XSS (Cross-Site Scripting)
☐ Authentification / autorisation	☐ Antiforgery token
☐ Protection des sessions (multitabs)	☐ Appliction Https et cacheRolesInCookie
☐ Validation des paramètres	☐ Divulgation d'informations dans le HTML
Structure des projets et des fichiers	
☐ Structure d'une solution	☐ Structure des namespaces
☐ Contenu des fichiers de code	☐ Ne pas réutiliser directement des projets existants
Normes de programmation générale	
☐ Convention de nom (C#, js, css)	☐ Visibilité des classes / membres
☐ Formatage du code	☐ Commentaires
☐ Langue du code	☐ Éviter les variables static / shared
☐ Utilisation de Nuget	☐ Convertir avec les Try... si possible (ex. : int.TryParse)
☐ Bloc de code sur une ligne ou sans bracket {}	☐ Évitez les « Magic string » / « Magic number »
☐ Bonnes pratiques Entity Framework	☐ Utiliser des enum avec extension pour les codes
☐ Bonne utilisation des extension methods	☐ Avertissements et code issues Resharper
Normes de programmation Web	
☐ Validation des paramètres	☐ Lean page / lean controller
☐ Style inline	☐ Modification des CSS avec LESS ou SASS
Architecture, patterns et performance	
☐ Respect des couches et de leurs responsabilités	☐ Injection de dépendance et bonnes pratiques
☐ Complexité des méthodes (métriques)	☐ Utiliser le type le plus abstrait
☐ Gestion des ressources (Dispose)	☐ Don't repeat yourself (DRY)
☐ Filtrer sur SQL et réduire le nombre de requêtes	☐ Retirer les références à des assemblies non utilisées
Tests	
☐ Présence et structure	☐ Format des tests (AAA) et conventions de nom

Explications

Sécurité

Injection (Sévérité : Critique, Nature : Sécurité)

Est-ce que toutes les requêtes SQL sont paramétrées? Attention : l'injection peut aussi se produire avec d'autres types de requêtes (LDAP, ouverture de fichier sur l'OS, etc.).

XSS (Cross-Site Scripting) (Sévérité : Critique, Nature : Sécurité)

Est-ce que les données de formulaires, cookie, querystring sont validées en entré et que les informations affichées sont proprement encodées (HTMLEncode).

Authentification / autorisation (Sévérité : Critique, Nature : Sécurité)

Est-ce que toutes les opérations qui nécessitent une autorisation vérifient que l'utilisateur courant est authentifié et valide qu'il peut effectivement accéder aux données qu'il tente d'accéder. Attention : même si une action sécurisée n'est pas disponible à l'écran, elle peut quand même être appelée via des outils comme Postman. Il faut donc valider en tout temps les autorisations sur les actions sur le serveur.

Antiforgery token (Sévérité : Critique, Nature : Sécurité)

Avec ASP.Net MVC, vérifier l'utilisation de l'antiforgery token dans les posts pour se protéger des attaques de type CSRF (Cross-Site Request Forgery).

Protection des sessions (multitabs) (Sévérité : Critique, Nature : Sécurité)

Si des informations sont mises en session vérifier la possibilité de collision avec le multitab. Si c'est le cas, implémentez une logique de protection du multi-tab ou revoyez l'utilisation faite de la session pour éliminer les conflits possibles. Dans la mesure du possible, limiter au maximum l'utilisation des sessions.

Application Https et cacheRolesInCookie (Sévérité : Critique, Nature : Sécurité)

Dans l'intranet, il faut s'assurer que dans le web.config des applications que le paramètre cacheRolesInCookie soit à false, particulièrement important pour les applications https. Ex. : **`[retiré pour des raisons de confidentialité]`**

Validation des paramètres (Sévérité : Critique, Nature : Sécurité)

Les paramètres d'une méthode doivent être validés, particulièrement pour les méthodes publiques et les méthodes exposées comme « endpoint » (Ex. : action en MVC, opérations en WCF, toutes les méthodes publiques d'un service ASMX).

Divulgation d'informations dans le HTML (Sévérité : Critique, Nature : Sécurité)

Toutes les informations retournées au navigateur peuvent être « lues » par l'utilisateur. Si une information confidentielle ne doit pas être visible à l'utilisateur, la mettre dans un champ caché ou une section non affichée ne fonctionne pas. Si l'information ne doit pas être visible, ne l'envoyez pas au client. Gardez-la dans la session ou la base de données. Dans certains cas (très rares), la solution pourrait être le chiffrement de l'information.

Structure des projets et des fichiers

Structure d'une solution (Sévérité : Medium, Nature : Architecture)

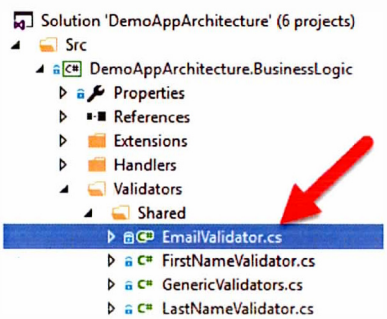
Pour les applications (autre que le site internet qui utilise une structure), tous les projets devraient se retrouver dans une solution. La structure de la solution devrait ressembler à la prise d'écran ci-dessous. Voici quelques explications sur les dossiers principaux :

- **Racine** : Ne devrait contenir que le fichier de solution (.sln) et des fichiers de configuration de VS. Si le projet utilise Git comme gestionnaire de source, les fichiers .gitattributes et .gitignore doivent également se trouver dans la racine du projet.
- **Dependencies** : Toutes les librairies (internes ou tiers) qui ne sont pas installées via Nuget.
- **Packages** : Dossier d'installation pour les librairies installées par Nuget (créer automatique lors de l'installation du premier package Nuget).
- **Src** : Code source de l'application. À l'intérieur de ce dossier, on retrouve un sous-dossier par projet.
- **Tests** : Code des tests (unitaires ou intégrés). À l'intérieur de ce dossier, on retrouve un sous-dossier par projet. Par convention, le nom du projet de test unitaire pour un projet sera [nom_du_projet].UnitTests (ex. : ProjectA.UnitTests) et [nom_du_projet].IntegrationTests (NB : cela ne correspond pas exactement à la prise d'écran puisque ces standards ont évolué avec le temps. On ne suggère pas de modifier les projets existants pour respecter cette convention, mais d'en tenir compte pour les nouveaux projets).
- **Tools** : Tous les outils externes qui pourraient être nécessaires à la compilation ou au déploiement de l'application.

[Prise d'écran retirée pour des raisons de confidentialité]

Structure des namespaces (Sévérité : Medium, Nature : Architecture)

Par convention et par souci de simplicité, les namespaces de vos classes devraient reproduire la structure des dossiers dans un projet. Par exemple, dans le projet ci-dessous, le namespace de la classe `EmailValidator` devrait être DemoAppArchitecture.BusinessLogic.Validators.Shared. Si vous déplacez du code dans votre projet ou si vous réalisez qu'une classe n'est pas dans le bon namespace, utiliser le refactoring `Ajust Namespace` de Resharper pour vous aider.



Contenu des fichiers de code (Sévérité : Medium, Nature : Architecture)

Les fichiers de code ne devraient contenir qu'un seul type public et le nom du fichier devrait refléter le nom de ce type. Exception : pour les enums, il est permis de les mettre dans le même fichier dans la classe qui utilise cet enum mais rien de vous empêche d'en faire un fichier séparé si nécessaire. Comme pour les namespaces, Resharper fournit des refactoring pour extraire un type vers un nouveau fichier ou pour renommer le fichier pour qu'il corresponde au nom de la classe. Utilisez-les, c'est très utile.

Ne pas réutiliser directement des projets existants (Sévérité : Medium, Nature : Architecture)

À la création d'une nouvelle application, il est tentant de prendre une application existante, de la copier dans un nouveau répertoire et de renommer les éléments. Cette façon de faire est à proscrire, car il est très difficile d'éliminer toutes les traces de l'ancienne application de votre projet. Il est plutôt recommandé de vous servir de la solution existante comme d'un modèle et d'en reproduire la structure dans votre nouvelle application. Cela ne prend que quelques minutes et vous évitera bien des soucis, particulièrement avec les problèmes de dépendances des packages Nuget. Vous perdrez plus de temps à régler cela que le temps économisé en copiant la solution. S'il vous faut certains des fichiers dans l'ancienne solution, ne copier que ceux dont vous être certains d'avoir besoin (renommer les namespaces si nécessaire) et en cas de doute, attendez que le besoin se présente. Vous évitez ainsi de trainer des fichiers inutiles et la prochaine personne au projet ne se posera pas la question : « Pourquoi ce fichier de la solution n'est pas utilisé nulle part »?

Normes de programmation générale

Convention de nom (Sévérité : Mineur, Nature : Lisibilité)

Consulter le tableau ci-dessous pour un aperçu des différentes conventions de nom.

Règle	Lang	S'applique à	Exemple
Les types ou membres public, internal, protected ou private doivent utiliser le Pascal Case	C#	Classes, Enum, Struct, Interfaces, méthodes, propriété	`public class StreamReader, internal Employee GetEmployee(), protected enum StatusCode`
Les champs privés d'une classe ou d'une structure débutent par un underscore (_)	C#	Champs privés	`private string _connString`
Les constantes (peu importe la visibilité) utilisent le PascalCase	C#	Constantes	`public const int MaxNbOfItems = 100`
Le nom des interfaces doit commencer par un I majuscule	C#	Interfaces	`public interface IEmployeeService`
Les classes de méthodes d'extension doivent porter le nom de la classe étendue suivi du suffixe Extensions	C#	Classes de méthodes d'extensions	`public static class StringExtensions`

Autres considérations pour les noms :

- Favorisez le nom le plus lisible plutôt que le plus court à écrire. En général, une ligne est lue 10 fois pour une écriture. De plus, les IDE modernes accélèrent grandement les choses. Par exemple, une méthode nommée GetCustomerPurchaseOrders sera beaucoup plus parlante qu'une méthode nommée GetCustPO.
- Évitez les abréviations dans vos noms à moins que celles-ci soient connues de tous dans l'entreprise ou dans le domaine d'affaires (ex : DB, CRX, ID, QC) etc.

Ces conventions sont grandement inspirées des recommandations de Microsoft. Vous pouvez les consulter ici : https://msdn.microsoft.com/en-us/library/ms229002(v=vs.110).aspx

Visibilité des classes / membres Convention de nom (Sévérité : Medium, Nature : Encapsulation)

Pour les classes et les membres, on veut utiliser le niveau de visibilité le plus restrictif possible pour répondre au besoin. Une bonne façon de résonner sur cette règle est de se dire « Tout est privé jusqu'à preuve du contraire ». On utilisera private ou public la majorité du temps et protected dans certains cas d'héritage. On utilise peu la visibilité internal dans nos projets, car on n'a pas souvent le cas où une classe / méthode ne doit être visible qu'à l'intérieur de l'assembly. Si c'est toutefois un besoin que vous avez, considérez la testabilité et assurez-vous de rendre visibles vos membres infernaux aux tests unitaires. Voir le lien suivant pour plus d'informations : http://stackoverflow.com/a/1809482/112259

Formatage du code Convention de nom (Sévérité : Mineur, Nature : Lisibilité)

Utiliser le formatage standard de Visual Studio (ou de Resharper) pour le formatage de votre code. On ne décrira pas toutes les règles ici, mais une fois vos règles correctement configurées, utilisez l'option format document (`edit > advanced > Format Document`) pour réaligner le contenu du fichier. Attention : ne le faire que pour les fichiers que vous modifiez pour le pas générer un paquet de modifications dans le commit qui rendra la vie difficile à la personne qui va revoir le code.

Commentaires (Sévérité : Mineur, Nature : Lisibilité)

Les commentaires dans le code doivent être pertinents et à jour. On privilégiera du code clair qui « s'autodocumente » que des commentaires qui expliquent comment le code fonctionne. Si le réviseur a besoin d'un commentaire pour comprendre le code, c'est probablement que celui-ci n'est pas assez clair. Par contre, des commentaires qui expliquent l'intention du code (le pourquoi) sont très importants, particulièrement si la technique ou l'algorithme utilisé est inhabituel. Comme cette documentation doit être concise et claire, il est recommandé d'écrire ces commentaires en français pour éviter d'ajouter un niveau de confusion supplémentaire suite à la traduction.

Les méthodes et membres publiques doivent également être documentées (en français) avec les commentaires xml (le ///) et les descriptions des paramètres devraient décrire et expliquer les valeurs attendues. Cela est particulièrement important pour les paramètres de type booléen (que représente la valeur vrai ou fait dans ce cas) ou les paramètres qui représentent un code (ex : 0 = Employé régulier, 1 = Employé temporaire). Dans ce cas, expliquer chacune des valeurs possibles et le comportement de la méthode si une valeur inconnue est passée.

Finalement, le code en commentaire devrait être retiré à moins qu'un autre commentaire n'explique la pertinence de garder ce commentaire dans le code source.

Langue du code (Sévérité : Mineur, Nature : Lisibilité)

Après discussion avec les membres de l'équipe, il a été décidé les règles suivantes pour la langue dans le code :

- Le code comme tel doit être en anglais. Cela comprend le nom des variables, des classes, des méthodes ou de tout autre élément du code. Toutefois, il est important d'être constant et d'utiliser toujours les mêmes termes pour représenter le même concept. De plus, si la traduction du concept n'est pas évidente et qu'elle peut porter à confusion, n'hésitez pas à ajouter un commentaire sur la définition de la classe ou de la méthode pour décrire le nom ou le concept français équivalent.
- Les commentaires et descriptions des éléments publics (summary) devraient être en français (voir section sur les commentaires pour plus de détails).

Éviter les variables static / shared (Sévérité : Majeur, Nature : Concurrence)

Les variables statiques peuvent créer des problèmes, car elles sont partagées par tous les utilisateurs de l'application et les modifications à ces valeurs peuvent affecter tout le monde avec des effets de barrer imprévus dépendant du moment ou de l'endroit où se produit la modification

de valeur. De façon générale, on tentera de les éviter. Si vous désirez les utiliser, assurez-vous minimalement que la mise à jour des valeurs est contrôlée contre la concurrence (un lock) et que votre code tient compte du fait que la valeur peut changer à tout moment, y compris en plein milieu d'un algorithme. C'est un gros risque à prendre qui ne vaut généralement pas la peine. N. B. Vous pouvez utiliser des méthodes statiques sans crainte. Ce sont les variables statiques qui peuvent être problématiques.

Utilisation de Nuget (Sévérité : Medium, Nature : Architecture)

Pour gérer les dépendances, privilégiez l'utilisation de Nuget. La gestion des versions des dépendances se fera alors plus facilement. Continuez à référencer manuellement les librairies pour lesquelles il n'existe pas de package Nuget. Dans ce cas, n'oubliez pas d'inclure cette dépendance dans le dossier Dependencies de votre solution.

Utilisez les méthodes Try... si disponibles lors de vos conversions (Sévérité : Mineur, Nature : Performance)

Lorsque vous devez faire des conversions, vérifiez si une méthode TryParse est disponible pour le type que vous utilisez et si c'est le cas, privilégiez celle-ci à la méthode Parse qui lancera une exception que vous devrez gérer. Ex. : int.TryParse() au lieu de int.Parse().

Évitez les blocs de code sur une ligne ainsi que les structures de contrôle sans accolade (Sévérité : Mineur, Nature : Lisibilité)

Pour garder une certaine consistance dans le code et éviter des erreurs difficiles à détecter, il est recommandé de toujours formater vos structures de contrôle (if, else, for, while, foreach) de la même façon c'est-à-dire l'instruction de contrôle sur une ligne et les instructions entre accolades, même s'il n'y a qu'une instruction. Cela facilite la lecture et réduit le risque d'erreur si le code évolue et qu'une 2e instruction s'ajoute au bloc. Voir exemple 1 à la fin du document.

Évitez les « magic strings » ou « magic numbers » (Sévérité : Medium, Nature : Maintenance)

L'utilisation de string ou de chiffre en clair est généralement à proscrire, car cela peut créer de la confusion (on peut se demander d'où vient ce chiffre). Aussi, si le chiffre change dans le futur, il faudra parcourir le code pour voir si la valeur est utilisée ailleurs et si oui, faire les corrections. Ce type de changement peut également introduire d'autres types d'erreurs. Si, par exemple, j'ai une instruction du genre `for (int i; i<100 ;i++)` et que je décide de changer la valeur de la borne supérieure (100) pour autre chose, est-ce que je peux automatiquement conclure que toutes les utilisations du chiffre 100 dans mon code doivent être remplacées? Peut-être pas, ça dépend de l'interprétation du chiffre 100 dans le contexte. Pour toutes ces raisons, il est préférable d'utiliser des variables de configuration ou des constantes et de créer une constante par concept, même si le chiffre que ces concepts représentent est le même (ex : MAX_NB_OF_STUDENTS = 30, MAX_NB_OF_DESKS = 30), car rien de garantie que ce sera toujours le cas. *Exception* : il peut être acceptable dans certains scénarios d'utiliser directement des chiffres pour des concepts qui ne changeront jamais et pour lesquelles la valeur est fixe et connue. Ex. : `items.Count > 0, NbSemaine = NbJour / 7, NbMinutes = NbHeures * 60`.

Bonnes pratiques Entity Framework (Sévérité : Medium, Nature : DB)

Entity Framework est un outil très intéressant, mais qui peut comporter certains pièges à éviter. Voici une liste de quelques bonnes pratiques avec EF :

- **Éviter de faire générer le modèle de BD par EF. En général, le modèle est déjà connu et l'on veut l'utiliser tel quel, à moins d'être en mode exploratoire et même dans ce cas, on voudra probablement « geler » le modèle après quelque temps. Pour désactiver cette fonctionnalité, dans le constructeur de vos DbContext, utilisez `Database.SetInitializer<[Type de votre DbContext]>(null);`.**
- **Contrôlez le cycle de vie de vos contextes, idéalement en les instanciant dans un using pour vous assurer de gérer correctement la connexion avec la base de données et la transaction dans le cas de mise à jour.**
- **Dans la mesure du possible, utilisez le nom de la table pour le nom de la classe d'entité associée à cette table. Toutefois, si l'utilisation du nom de la table fait que le nom de la classe ne respecte pas les conventions de nommage (ex. : TABLE_DES_ELEMENTS), privilégiez un nom de classe qui respecte la convention de nommage (ex. : TableDesElements) et utilisez les fonctions de mapping d'Entity Framework pour associer le nom de la table à la classe. Il est préférable de circonscrire un nom de table non standard ou qui appartient à une autre équipe dans la classe de contexte d'Entity Framework que de « polluer » le code à plusieurs endroits avec des noms qui ne respectent pas la convention.**
- **Dans un contexte web où les requêtes sont courtes, il est peu efficace d'utiliser le lazy loading. Cela peut même être problématique si vous sérialisez par exemple un objet de votre contexte en JSON alors que la connexion avec la BD est fermée. Il est généralement plus simple de désactiver le lazy loading et d'aller chercher toute l'information nécessaire (et juste l'information nécessaire) d'un seul coup.**
- **Dans le cas où votre structure d'objet est complexe, il peut être intéressant de jeter un coup d'œil au query SQL généré par EF. Vous pourriez être surpris par la quantité de traitements nécessaires pour retourner toute l'information demandée et il se pourrait que vous ayez une meilleure performance en découpant votre requête hyper complexe en 2 ou 3 requêtes beaucoup plus simples.**

Utilisez des enums pour représenter un ensemble de valeurs reliées (Sévérité : Medium, Nature : Maintenance)

Si vous avez une liste de constantes à définir (ex. : une liste de statuts de commande), privilégiez l'utilisateur d'un enum. Cela vous permet de vous assurer que vous n'aurez pas de valeur inconnue dans votre champ (à moi de définir un élément de votre enum comme inconnu, ce qui peut être une bonne idée, mais si c'est le cas, il est recommandé de le définir comme valeur par défaut, soit le premier élément de l'enum). Dans les cas où vous devez transformer ce statut en code alpha numérique pour communiquer avec un autre système (ou pour toute autre opération à effectuer ou valider sur la valeur d'un enum), définissez des méthodes d'extensions sur votre enum. *Informations supplémentaires* : vous remarquerez dans certains programmes l'utilisation de classe de type BaseValueType pour offrir l'équivalent d'un enum avec des propriétés supplémentaires. Cette technique ne devrait plus être utilisée, car on a découvert qu'elle supportait mal la sérialisation avec Json.Net qui modifie des valeurs privées dans la classe et peut alors modifier le comportement de l'application. Si vous voulez continuer à utiliser cette

technique, vous devez vous assurer d'exclure de la sérialisation toutes les propriétés dont le type est dérivé de BaseValueType. C'est possible, mais le risque d'erreur est grand. C'est pourquoi la méthode des enums avec des extensions est suggérée. Voir exemple 2.

Bonne utilisation des extension methods (Sévérité : Mineur, Nature : Lisibilité)

Les extention methods (EM) sont un outil très utile pour généraliser des comportements ou pour rendre votre code plus lisible. Toutefois, comme n'importe quel outil puissant, il est possible (et tentant) d'en abuser. Aussi, voici quelques lignes directrices pour une bonne utilisation des extension methods :

- Créer des EM pour les types qui sont en dehors de votre solution (types du framework, de librairies nuget, de librairies internes provenant d'autres projets). Si le type et dans votre solution, privilégiez l'ajout de la fonctionnalité directement dans le type. Petite exception à cette règle : comme les EM sont statiques et que l'instance sur laquelle s'applique la méthode est passée en paramètre, les EM peuvent être pratiques pour des cas où l'instance peut être nulle. Voir exemple 3.
- Application les EM au type de plus précis possible pour les besoins de votre EM. Comme les EM apparaissent dans l'Intellisence, on veut éviter de « polluer » la liste des membres avec des extensions qui ne s'applique pas vraiment au type courant.
- Réunir toutes les EM pour un type dans la même classe. De plus, cette classe ne devrait contenir que des EM et seulement pour un type.
- Respecter les conventions de nom pour les classes contenant des EM : `[nom de la classe étendue]Extensions`. Ex. : `StringExtensions`, `ObjectExtensions`, `IEnumerableExtensions`, `IEnumerableOfBoolExtensions`.

Pour plus d'informations : https://msdn.microsoft.com/en-us//library/bb383977.aspx

Avertissements et code issues Resharper (Non applicable, selon le problème soulevé par Resharper)

Les avertissements à la compilation donnent des informations importantes sur des problèmes potentiels dans votre code. Ils devraient être considérés avec autant d'attention que les erreurs. Essayez de les garder sous contrôle, particulièrement pour les nouveaux projets où il est plus facile de les régler au fur et à mesure. Pour les projets existants qui peuvent comporter beaucoup d'avertissements, utilisez la règle du scout (nettoyez les classes / méthodes / fichiers que vous touchez, mais ne vous lancez pas dans une chasse à tous les avertissements à moins que ce ne soit l'objectif de la demande de modification).

Aussi, lorsque vous ouvrez un fichier de code, Resharper fait une analyse du fichier et donne des indications (code issues) sur les problèmes potentiels. Ils sont affichés dans la barre de défilement de votre fenêtre de code. Vous pouvez également parcourir les code issues en cliquant sur le signe d'avertissement (en haut de la barre de défilement) pour passer au code issue suivant.

Dernière chose concernant Resharper : idéalement, il est plus pratique de laisser Resharper activé en tout temps pour vous les problèmes en temps réel et pour pouvoir les corriger. Toutefois, dans certaines situations (machine lente, grosse solution, etc.), Resharper peut ralentir Visual Studio assez pour causer des désagréments. Dans un premier temps, jetez un coup d'œil à l'article suivant qui donne des indications sur la façon d'accélérer Resharper et Visual studio. Si ça demeure problématique, vous pouvez le désactiver durant votre développement « actif », mais, avant de faire un checkin ou un commit, il serait intéressant de l'activer et de faire la révision des règles pour les fichiers que vous vous apprêtez à envoyer dans le gestionnaire de code source.

Normes de programmation Web

Validation des paramètres (Sévérité : Majeur, Nature : Sécurité)

Une application web est une application de type client-serveur où le client est un navigateur, une application mobile ou une autre application web. Le protocole HTTP, utilisé pour les communications, est très simple et il est facile de modifier le contenu d'une requête envoyé au serveur. Pour cette raison, les informations qui proviennent du client (formulaire, query string, cookie, header http, etc.) doivent **TOUJOURS** être validées sur le serveur et considérées comme invalides jusqu'à preuve du contraire. Les validations effectuées sur le client (par javascript ou une application mobile) permettent d'améliorer l'expérience utilisateur, mais elles ne sont d'aucune valeur pour assurer la validité et la sécurité des informations, car elles peuvent être altérées via les outils de développement des navigateurs (ex. : Chrome developer tools) ou des outils de proxy à l'extérieur des navigateurs (ex. : Fiddler). Les validations doivent inclurent (entre autres) la validation de la présence des informations requises, le format du contenu (text vs int, etc.) ainsi que la validation des droits d'accès à l'information demandée (ex : est-ce que l'utilisateur authentifié peut obtenir l'information du patient demandé).

Pour les validations de sécurité, n'hésitez pas si nécessaire à utiliser plusieurs mécanismes de protection superposés. Par exemple, on pourrait en même temps utiliser un attribut [Authorize] sur une action de contrôleur pour valider en amont que l'utilisateur est authentifié et ensuite valider dans la logique d'affaires l'appartenance du client à un groupe ou si les données demandées lui sont accessibles. Finalement, si plusieurs intervenants sont nécessaires pour obtenir l'information, il peut être à propos, dépendant du niveau de confidentialité des informations, de faire des validations d'autorisation à chaque appel à un service web pour éviter qu'un attaquant potentiel « bypass » la sécurité en appelant directement un service d'accès aux données qui ne serait pas sécurisé. En cas de doute, n'hésitez pas à en discuter avec vos collègues.

Lean page / lean controller (Sévérité : Medium, Nature : Testabilité, maintenance)

Les pages web aspx et les contrôleurs MVC sont difficiles à tester, car elles font souvent appel à des éléments du HttpContext, de la requête, de la session etc. Hors, ces objets n'existent que dans le traitement d'une requête http sur un serveur web. Pour nous permettre de tester le plus de code possible via des tests unitaires automatisés, on veut minimiser la quantité de code contenu dans le code behind des pages aspx et les contrôleurs MVC. En MVC, on utilise pour cela le pattern « lean controller » c'est-à-dire qu'on veut que le contrôleur en fasse le moins possible soit : lire le contenu des requêtes http nécessaire pour le traitement, prendre ce contenu et initialiser une structure d'objet avec, appeler une commande ou un service pour faire le traitement, prendre la réponse de ce traitement, ajouter des informations si c'est nécessaire et formater une réponse ou générer une vue. Toutes les validations, les logiques d'affaires et les traitements devraient se retrouver en dehors du contrôleur.

En Web Form, on utilisera un pattern similaire c'est-à-dire que la responsabilité de la page devrait être de : intercepter les événements de la page, lire les informations requises dans les contrôles de la page, prendre ce contenu et initialiser une structure d'objet avec, appeler une commande ou un service pour faire le traitement, prendre la réponse de ce traitement, mettre à jour les contrôles de la page avec le contenu de la réponse. Toutes les validations, les logiques d'affaires et les traitements devraient se retrouver en dehors de la page.

Style inline (Sévérité : Medium, Nature : Maintenance)

Les styles inline (dans la page ou directement dans la propriété Style d'un élément HTML) sont à proscrire, car on ne peut pas remplacer ces styles globalement en modifiant les fichiers de style du site.

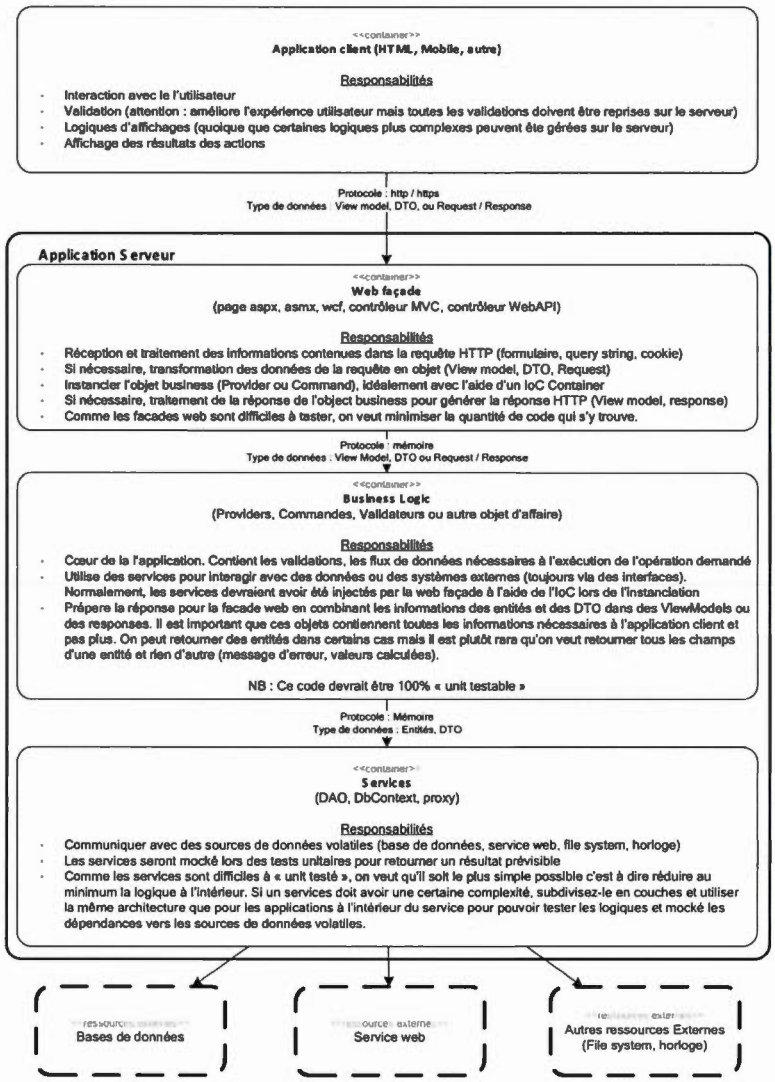
Modification des CSS avec LESS ou SASS (Sévérité : Medium, Nature : Maintenance)

Lorsque des précompilateurs de style comme LESS et SASS sont utilisés dans un projet web, il ne faut pas modifier directement les styles CSS lorsque ceux-ci sont générés par un des outils, car les modifications seront éventuellement perdues si les fichiers CSS sont régénérés (à la compilation ou l'exécution). Faites plutôt vos ajustements dans les fichiers LESS ou SASS correspondants.

Architecture, patterns et performance

Respect des couches et de leurs responsabilités (Sévérité : Medium, Nature : Architecture)

Il y a plusieurs façons de gérer les couches des applications. Toutefois, dans le cas des applications web faites au Centre Rx, les applications devraient avoir une structure assez similaire. La structure générale d'une application avec la responsabilité des couches est décrite dans le modèle ci-dessous :



Il s'agit bien sûr d'un modèle de base et une application plus complexe pourrait comporter plus de couches. Ajustez selon vos besoins, mais ne multipliez pas les couches pour rien.

Injection de dépendance et bonnes pratiques (Sévérité : Medium, Nature : Architecture)

Afin de réduire le couplage et ainsi aider pour les tests unitaires et la maintenance, c'est une bonne pratique d'utiliser l'injection de dépendances pour vos applications. Il y a plusieurs façons de faire de l'injection de dépendances, mais voici quelques-unes des pratiques à privilégier :

- Utilisez Autofac comme outil pour gérer vos dépendances. Installez l'outil via Nuget (`install-package Autofac`). Dépendant de votre type d'application, installez également la librairie d'intégration correspondante
  - Web Form : `Autofac.Web`
  - MVC 4: `Autofac.Mvc4`
  - MVC 5: `Autofac.Mvc5`
  - WebAPI : `Autofac.WebApi`
  - WebAPI 2 : `Autofac.WebApi2`
- Au démarrage de votre application, appelez une classe (nommée par convention BootStrapper) pour configurer les dépendances.
- Dans votre bootstrapper, privilégier la configuration des dépendances par construction automatique (`builder.RegisterType<TConcreteType>().As<IInterfaceType>()`). Avec cette méthode, la construction de vos dépendances s'ajustera automatiquement si de nouveaux paramètres sont ajoutés.
- Même si d'autres alternatives existent, utilisez l'injection par paramètres de constructeurs lorsque vous voulez injecter des dépendances dans vos contrôleurs MVC et toutes les classes que vous créez. Aussi, si possible, n'ayez qu'un seul constructeur avec toutes les dépendances requises et évitez d'avoir un constructeur sans paramètre qui instancie manuellement les dépendances.
- Idéalement, limitez vos dépendances à 5 ou moins. Si vous en avez un plus grand nombre, vérifiez si toutes les dépendances sont nécessaires ou si un redécoupage des dépendances serait requis.
- Si votre constructeur contient des paramètres de type valeur (string, int, etc.), privilégier la construction avec fonction de construction instanciant directement le type concret (`builder.Register<IInterfaceType>(c=> new ConcreteType(param1,param2))`)
- Pour les applications web form, comme l'injection par constructeur est impossible, utilisez plutôt l'injection par propriété. Voir le détail de l'utilisation à l'adresse suivante : http://docs.autofac.org/en/latest/integration/webforms.html
- Attention au cycle de vie de vos dépendances. Pour une application web, les cycles de vie à utiliser devraient être, en ordre de priorité :
  - InstancePerRequest : une instance du type est créée pour chaque requête et est partagée par les autres services durant la durée de vie de la requête. La plupart des dépendances de votre application devraient utiliser ce cycle de vie.
  - InstancePerDependency : une nouvelle instance est créée à chaque demande. Ce type de cycle de vie est à privilégier pour les cas où l'on ne peut pas / ne veut pas partager une instance entre plusieurs services. Un cas possible pourrait être un DbContext pour lequel on veut contrôler et réduire la durée de la connexion.

  - SingleInstance : avec ce cycle de vie, toutes les requêtes web partagent la même instance. À éviter autant que possible, car très dangereux (partage d'information entre les requêtes). Un cas d'utilisation possible : un objet en lecture seule contenant les configurations de l'application (puisque les configurations sont les mêmes pour tout le monde et qu'un changement de configuration amène un redémarrage de l'application).
- Évitez autant que possible d'utiliser un Service Locator c'est-à-dire une classe (ou une référence directe à Autofac) qui sert à créer vos dépendances et qui est utilisée un peu partout dans le code. On veut idéalement configurer les dépendances à un seul endroit (le bootstrapper) et ne plus avoir à faire de référence au container dans le reste de l'application.

Le sujet de l'injection de dépendances est vaste et important, au point où des livres complets y ont été consacrés. Pour plus d'information sur le sujet, vous pouvez consulter l'ouvrage Dependency Injection in .Net de Mark Seemann (disponible dans la bibliothèque d'équipe). Pour des best-practices sur l'injection avec Autofac : http://docs.autofac.org/en/latest/best-practices/. Comme toujours, n'hésitez pas à en discuter avec vos collègues en cas de doute.

Complexité des méthodes (métriques) (Sévérité : Medium, Nature : Maintenance)

La lisibilité et la facilité de comprendre le code sont des éléments importants de la qualité du code. Un des facteurs les plus importants pour faciliter la compréhension et la maintenance du code est la complexité cyclomatique (CC). C'est un grand mot pour décrire un concept simple : la CC représente le nombre de chemins logiques possibles dans le code. Par exemple, un code sans aucune structure de contrôle (if, for, switch, etc.) a une complexité de 1. Si l'on ajoute un if/else, on a une complexité de 2 (on peut passer dans le if ou dans le else). On veut dans la mesure du possible réduire la complexité des méthodes. Une bonne pratique est de regarder attentivement les méthodes dont la complexité dépasse 10 et vérifier si l'on ne pourrait pas la découper la méthode en plusieurs petites méthodes plus simples. Réduire la complexité a deux avantages : en plus de faciliter la compréhension du code, elle en facilite également les tests puisqu'une des règles dans les tests unitaires est de tester au moins tous les chemins possibles donc plus une méthode est complexe, plus on devra écrire de tests pour passer par tous les chemins. Pour obtenir la complexité d'une méthode, 2 options s'offrent à vous : l'outil de Code Metrics intégré à Visual Studio qui vous permet de parcourir un projet ou une solution en entier (plus pratique lors des revues) ou une extension à Resharper (nommé Cyclomatic Complexity) qui créera un code issue dans le fichier si des méthodes ne respectent pas les limites définies dans les configurations (pratique lors du développement).

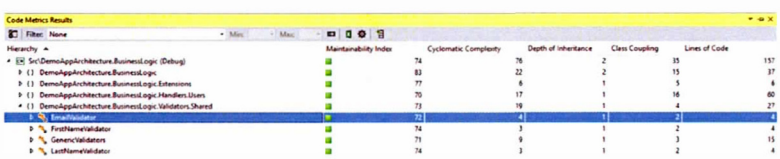


Figure 1 : Outil Code Metrics de Visual Studio

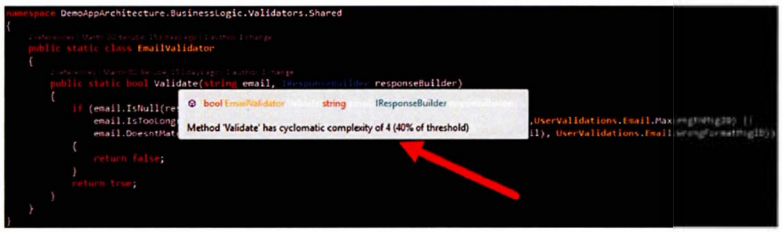


Figure 2 : Affichage de la complexité avec l'extension à Resharper

Utiliser le type le plus abstrait (Sévérité : Medium, Nature : Maintenance)

C'est une des bonnes pratiques de la programmation objet. Lorsque vous définissez une variable ou un paramètre, utilisez le type le plus abstrait qui répond à vos besoins. Par exemple, si vous faites une méthode pour faire une recherche dans une liste, utilisez le type IEnumerable<T> plutôt que List<T>. Ce faisant, vous avez plusieurs effets positifs :

- vous pouvez utiliser votre fonction sur des collections d'autres types que des listes (des arrays par exemple)
- vous informez indirectement les utilisateurs de votre méthode que vous ne ferez que parcourir la collection (car le type IEnumerable ne permet pas l'ajout ou le retrait d'éléments de la liste)
- vous limitez les options disponibles dans l'intellisence aux seules vraiment utiles pour vous.

Évidemment, cela ne veut pas dire d'utiliser le type Object pour tous vos paramètres, mais plutôt de se poser la question lorsqu'on passe un type concret à méthode si c'est le bon type pour répondre au besoin spécifique de la méthode ou si une autre abstraction ne serait pas plus appropriée.

Gestion des ressources (Dispose) (Sévérité : Majeur, Nature : memory leak)

Faites attention aux classes qui implémentent IDisposable, car elle gère des ressources qui doivent être libérées dès que possible (connexion SQL, connexion vers des services web, lock sur des fichiers sur le disque etc.). Lorsque que vous créez une instance d'un objet, vérifiez s'il implémente IDisposable et si c'est le cas, utilisez idéalement un using (ou un try/finally si ce n'est pas possible) pour vous assurer que le dispose sera appelé à la fin de la méthode quoi qu'il arrive.

Si vous utilisez l'injection de dépendance, les outils comme Autofac s'occupe de la gestion des cycles de vie des objets et se chargera d'appeler le Dispose() si nécessaire SI VOUS AVEZ FAIT LES CONFIGURATIONS CORRECTEMENT. Voilà pourquoi il est recommandé d'utiliser un cycle de vie comme InstancePerRequest (tel que décrit plus haut). Dans tous les cas, il est important lorsque vous configurez une dépendance qui implémente IDisposable de faire un test (en débug ou avec des traces) pour vous assurer que le Dispose() est correctement appelé à la fin de la requête et ainsi valider que votre configuration est adéquate. Ce type de problème est difficile à vérifier avec des tests unitaires et ne se manifestera pas nécessairement lors de vos tests inté-

grés. Un problème de gestion de ressources peut prendre des années à se révéler en production.

Finalement, quelques règles et astuces par rapport à la gestion des ressources :

- « *You create it, you dispose it* ». Donc, si vous recevez un objet en paramètre, c'est qu'il a été créé par quelqu'un d'autre, une autre classe ou un outil comme Autofac. Vous ne devez donc pas le disposer, car vous ne savez pas si cet objet sera réutilisé par quelqu'un d'autre par la suite. Par contre, si vous créez l'instance (directement avec new ou une factory), vous devez gérer son cycle de vie et vous assurer que le Dispose sera appelé quoi qu'il arrive avant de quitter la méthode.
- Vous réduisez grandement vos chances de problèmes en réduisant au minimum le partage d'instances qui implémentent IDisposable. Autant que possible, limitez l'utilisation de ces ressources à l'intérieur d'une seule méthode en utilisant un using.
- Pour réduire la durée de vie de vos objets IDisposable (particulièrement pour les DbContext), plutôt que d'injecter directement le contexte dans votre classe, injectez une factory qui vous fournira l'instance au moment où vous en aurez besoin. Vous pourrez ainsi créer cette instance à l'intérieur d'un using et ainsi bien encapsuler son cycle de vie. Si vous regardez l'exemple 4 pour le code et le résultat du test, vous verrez que de cette façon, le DbContext a une durée de vie très courte (le temps de l'appel) et qu'il est ainsi facile de s'assurer de ne laisser aucune connexion trainer.

Don't repeat yourself (DRY) (Sévérité : Medium, Nature : maintenance)

Un concept connu, mais on retrouve encore souvent du code « copy-pasté » d'un autre projet ou d'un autre endroit. Ne faites pas ça à moins que le code n'ait aucunement rapport entre les 2 endroits, mais habituellement, si l'on a pris la peine de copier le code, c'est que ce qu'on allait faire est assez similaire. Utilisez les outils (Resharper et VS) pour généraliser le code (extract method, create class, etc.) pour en favoriser la réutilisation. Visual Studio inclut un outil pour trouver les duplications de code :

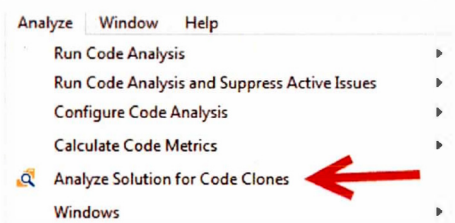


Figure 3: Option de menu pour l'analyse des clones

Figure 4: Résultats de l'analyse des clones

Fitrer sur SQL et réduire le nombre de requêtes (Sévérité : Majeur, Nature : performance)

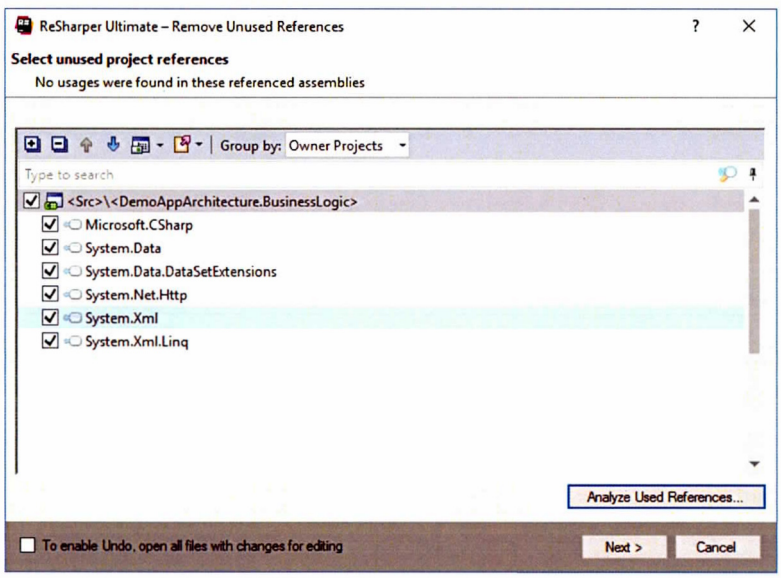
SQL est un outil très puissant pour filtrer et trié des résultats. Malgré le fait qu'on a aujourd'hui des outils comme Entity Framework qui cache les requêtes à la BD en les générant automatiquement, les bonnes pratiques de base pour l'utilisation d'une base de données demeurent :

- **Retourner le moins possible de données : assurez-vous de filtrer correctement vos requêtes SQL pour ne retourner à votre application que les informations nécessaires. Il est beaucoup plus efficace de filtrer les résultats sur SQL que de retourner tous les données et de filtrer dans le code. Cette méthode peut bien fonctionner avec un petit nombre de records, mais devient rapidement un problème de performance important quand la BD grossit.**
- **Réduire le nombre de requêtes au serveur SQL : la majorité du temps, c'est une bonne pratique d'organiser vos requêtes pour retourner toute l'information nécessaire d'un seul coup (ex. : un client avec ces commandes) plutôt que de multiplier les commandes courtes (ex. : aller chercher le client et boucler dans une collection pour aller chercher les commandes une par une sur SQL). Cet anti-pattern est parfois nommé le 1+N (voir http://stackoverflow.com/questions/97197/what-is-the-n1-selects-issue pour plus de détail sur cet anti-pattern). Attention toutefois : il ne faut pas nécessairement réduire à tout prix. Si vous avez une structure d'objet complexe, il se peut que la requête nécessaire pour retourner toute l'information nécessaire soit trop complexe et nuise à la performance. Parfois, 2 requêtes plus petites et bien ciblées sont plus efficaces qu'une grosse requête qui inclue tout. Utilisez les outils à votre disposition (Intellitrace entre autres est très pratique pour inspecter les requêtes générées par Entity Framework) pour vérifier vos requêtes et vous assurez qu'elles sont les plus efficaces possible.**
- **Il faut parfois sortir les requêtes du code pour pouvoir les optimiser. Lorsque les requêtes sont vraiment trop complexes, l'utilisation d'une vue SQL ou d'une procédure stockée peut s'avérer une façon de se simplifier le tout.**

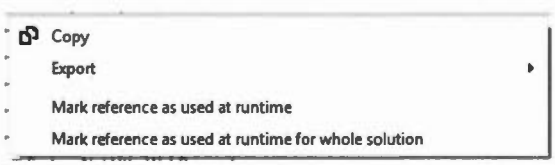
Retirer les références à des assemblies non utilisées (Sévérité : Medium, Nature : maintenance)

Pour simplifier la maintenance pour vos collègues et alléger vos applications, il est intéressant de vérifier que vous utilisez réellement les librairies que votre projet référence. Souvent, les

projets par défauts (file|new|project) inclus des références dont vous n'avez pas nécessairement besoin ou les projets comportent des références vers des librairies qui ne sont plus utilisées suite à un refactoring. Pour identifier ces assemblies, Resharper offre un outil intéressant. Pour l'utiliser, faites un clic droit sur votre solution dans le Solution Explorer et dans le menu Refactor, choisissez **Remove Unused References**. Cela affichera cette fenêtre :



Attention : certaines références sont peut-être utilisées au runtime. Avant de vous lancer dans le retrait massif de références, assurez-vous que votre code est à jour et idéalement, faites ces changements dans une branche séparée par prudence. Aussi, TRÈS IMPORTANT DE TESTER votre application après le ménage (pas juste compiler) pour vous assurer que vous n'avez pas retiré une référence indirecte (utilisée par une autre librairie ou au runtime). Un bon exemple de cela serait une librairie comme Elmah qui est référencée par le web.config sans être nécessairement utilisée dans le code. Si vous découvrez une de ses librairies, vous pouvez l'identifier pour éviter que cette librairie apparaisse de nouveau dans des analyses subséquentes. Pour ce faire, faites un clic droit sur la référence et choisissez Mark reference as used at runtime (ou for whole solution si la librairie est utilisée par plusieurs projets).



Tests

Présence et structure (Sévérité : Medium, Nature : Testabilité)

Durant les revues, on jette un coup d'œil à la présence et la structure des tests. Les bonnes pratiques de tests ne seront pas reproduites ici. Vous pouvez en trouver les détails dans le document de formation sur les tests unitaires disponible à cet emplacement.

Format des tests (AAA) et conventions de nom (Sévérité : Mineur, Nature : Lisibilité)

Ici aussi, référez-vous à la formation sur les tests unitaires pour les détails.

Exemples

Exemple 1

```
int a = 0;
int b = 0;

//Non
if (a == 0) b = 1;

//Non
if (a == 0)
    b = 1;

//Oui
if (a == 0)
{
    b = 1;
}

//Pourquoi? Erreur facile à laisser passer.
//Quelle est la valeur de b après être passé dans le if?
if (a == 0)
    b = 1;
else
    Console.WriteLine("Else");
    b = 2;

//Voyons l'équivalent avec des brackets
if (a == 0)
{
    b = 1;
}
else
{
    Console.WriteLine("Else");
}
b = 2; //Oups, la valeur de b est perdue. Plus évident comme ça
```

Exemple 2

```
public enum OrderStatus
{
    Unknown,
    Created,
    Processed,
    Shipped,
    Delivered
}

public static class OrderStatusExtensions
{
    public static string StringCode(this OrderStatus orderStatus)
    {
        switch (orderStatus)
        {
            case OrderStatus.Unknown:
                return "";
                break;
            case OrderStatus.Created:
                return "CR";
                break;
            case OrderStatus.Processed:
                return "PR";
                break;
            case OrderStatus.Shipped:
                return "SH";
                break;
            case OrderStatus.Delivered:
                return "DV";
                break;
            default:
                throw new ArgumentOutOfRangeException(nameof(orderStatus), orderSta-
tus, null);
        }
    }
}

//Utilisation

OrderStatus.Created.StringCode();
```

Exemple 3

```
public static class IEnumerableOrTExtensions
{
    public static bool IsNullOrEmpty<T>(this IEnumerable<T> collection)
    {
        return collection == null || !collection.Any();
    }
}

//Usage
IEnumerable<string> nullCollection = null;
IEnumerable<int> emptyCollection = new List<int>();

if (nullCollection.IsNullOrEmpty())
{
    //Empty collection business logic
}

if (emptyCollection.IsNullOrEmpty())
{
    //Empty collection business logic
}
```

Exemple 4

```
[TestClass]
public class AutofacFactoryFixtures
{
    [TestMethod]
    public void TestObjectLifecycleWithFactory()
    {
        //Arrange
        var builder = new ContainerBuilder();
        builder.RegisterType<CustomerContext>().As<ICustomerContext>().InstancePerDependency();
        builder.RegisterType<CustomerService>().InstancePerDependency();
        var container = builder.Build();

        //Act
        using (var scope = container.BeginLifetimeScope())
        {
            var customer1 = scope.Resolve<CustomerService>().GetCustomerByID(1);
            var customer2 = scope.Resolve<CustomerService>().GetCustomerByID(2);
        }
    }
}

public class Customer
{
    public int ID { get; set; }
    public string Name { get; set; }
}

public class CustomerService
{
    private readonly Func<ICustomerContext> _customerContextFactory;
    public CustomerService(Func<ICustomerContext> customerContextFactory)
    {
        _customerContextFactory = customerContextFactory;
    }

    public Customer GetCustomerByID(int id)
    {
        using (var context = _customerContextFactory.Invoke())
        {
            return context.GetCustomerByID(id);
        }
    }
}

public interface ICustomerContext:IDisposable
{
    Customer GetCustomerByID(int id);
}

public class CustomerContext : DbContext, ICustomerContext
{
    private readonly Guid _instanceGuid;
    public CustomerContext()
    {
        _instanceGuid = Guid.NewGuid();
        Console.WriteLine("Creation of CustomerContext instance id {0}",_instanceGuid);
    }
    public Customer GetCustomerByID(int id)
    {
        return Customers.SingleOrDefault(x => x.ID == id);
    }

    public IDbSet<Customer> Customers { get; set; }

    protected override void Dispose(bool disposing)
    {
        if (disposing)
        {
            Console.WriteLine("Disposing CustomerContext instance id {0}", _instanceGuid);
        }
        base.Dispose(disposing);
    }
}
```

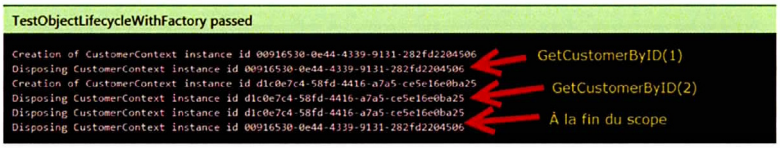


Figure 5 : Résultats obtenus lors de l'exécution de l'exemple 4

ANNEXE V

PROCESSUS DE REVUE DOCUMENTÉ

Processus de revue par les pairs

Documentation permanente

Documentation (D99-1)

Martin Bérubé

Révision du gabarit : 2016-08-26

Destinataire du document

Ce document s'adresse à l'équipe Web-Micro pour définir le processus de revues par les pairs.

Historique de modification

Date	Description de la modification	Par
11 octobre 2016	Création du document	Martin Bérubé
19 mai 2017	Précisions sur la documentation des défauts. Modification dans les sections 3.4.3 et 3.5.2.	Martin Bérubé

Imprimé le 30 juillet 2017 9:07 pm

Table des matières

DOCUMENTATION

1. Introduction

1.1 But

Le but de ce document est de servir de point de référence pour le processus de revue par les pairs. Dans cette phase, on se concentrera sur la revue du code mais la documentation fera éventuellement l'objet de revue également.

1.2 Envergure

Le document se veut une description complète du processus de revue de code. Si des éléments supplémentaires ou de la documentation complémentaire sont requis, ils seront spécifiés dans les références.

1.3 Définitions, acronymes et abréviations

Glossaire

Défaut	Terme générique pour décrire un problème dans un artéfact produit lors du cycle de vie du logiciel. Il peut comprendre (mais pas exclusivement) des omissions, des erreurs, des bogues, des assomptions incorrectes, etc. En résumé, est considéré comme un défaut tout élément qui demande une vérification et éventuellement, une correction.
Auteur	Personne qui a produit l'artéfact (code ou document) à vérifier
Réviseur	Personne assignée à la vérification d'un artéfact
TFS	Team Foundation Server
Couplage	Traduction de Class Coupling. Le couplage représente le nombre de dépendances qu'une classe ou une méthode a sur d'autres classes (en le référençant ou en les utilisant).
Complexité cyclomatique	Traduction de Cyclomatic complexity. Représente le nombre de chemins logiques possible dans une méthode.
Pull Request (PR)	Technique souvent utilisée par les utilisateurs de git pour demander un merge du code personnel vers le code partagé. Le réviseur assigné au PR a la responsabilité de revoir le code avant d'accepter le merge et de transmettre ses observations à l'auteur si des défauts ont été détectés.

1.4 Références

Document	Emplacement	Description

2. Qu'est-ce que la revue par les pairs?

2.1 Définition

La revue par les pairs est une activité durant laquelle un artéfact du cycle de vie du logiciel sera inspecté par une ou plusieurs personnes, autre que son auteur, dans le but d'y déceler des défauts. Les défauts seront alors documentés et l'auteur fera les corrections nécessaires.

2.2 But

Le but premier de la revue par les pairs est de détecter le plus rapidement possible les défauts pour en atténuer l'impact négatif sur la qualité du produit livré. Un défaut trouvé rapidement après son introduction est plus simple et moins coûteux à corriger que s'il est détecté plus tard dans le cycle de vie du logiciel (voir Figure 1.).

2.3 Objectifs spécifiques

Les objectifs spécifiques du Centre Rx pour le processus de revue par les pairs sont :

- Assurer que les bonnes pratiques de sécurité du code soient systématiquement appliquées et vérifiées
- Réduire l'effort de correction de défauts (rework) qui peut représenter, selon les études, une moyenne de 44 % des coûts d'un projet (voir Figure 2).
- Augmenter la qualité interne du code et des documents pour réduire l'effort de maintenance
- Assurer un transfert de connaissance sur les applications
- Partager les connaissances et les compétences techniques
- Uniformiser les pratiques de développement entre les différents projets

2.4 La revue, c'est...

- Une façon d'améliorer la qualité générale des artéfacts produits par l'équipe
- Une occasion de partager de l'information et des connaissances sur les applications, les techniques, les processus d'affaires, le savoir-faire, l'expérience, etc.
- Un outil d'apprentissage, tant pour les auteurs que les réviseurs

2.5 Par contre, la revue n'est pas...

- Une perte de temps. Bien que le réviseur doive utiliser une partie de son temps pour faire des revues, le temps investi est la plupart du temps regagné en double ou en triple en temps de recherche, correction de bogues, tests, problème de production et maintenance. [Gilb, P. 24]
- Un outil d'évaluation (positif ou négatif). Jamais les résultats individuels des revues ne seront utilisés pour blâmer ou encenser un auteur ou un réviseur. On recherche à améliorer la performance globale de l'équipe et les mesures de qualité et de productivités seront faites sur l'ensemble des données pour en dégager les tendances. Les chiffres publiés à la direction seront toujours anonymes.
- Une critique de l'auteur du document. On revoit le code, pas l'individu. L'auteur n'est pas son code.
- Un exercice d'opinion. La revue se voulant un exercice le plus objectif possible, l'idée n'est pas de donner son avis (moi je n'aurais pas fait ça comme ça) mais de démontrer que des règles ou de bonnes pratiques n'ont pas été respectées. Chaque défaut documenté doit pouvoir être justifié par une règle. Il n'est toutefois pas nécessaire de faire référence à la règle à chaque fois, seulement de pouvoir y référer si

nécessaire.

- Une bataille d'égo. Lorsqu'un défaut est découvert, il est de la responsabilité du réviseur de le faire savoir (poliment) et à l'auteur d'en prendre note (humblement). Bien sûr, on peut discuter et argumenter sur le défaut mais toujours dans le respect et dans l'objectif d'améliorer les choses, pas de déterminer « qui a raison ». N'oubliez pas que vous jouez le rôle d'auteur et de réviseur tout à la fois donc traitez vos collègues avec la même courtoisie que vous attendez d'eux. Tout le monde a quelque chose à nous enseigner, du plus junior au sénior.

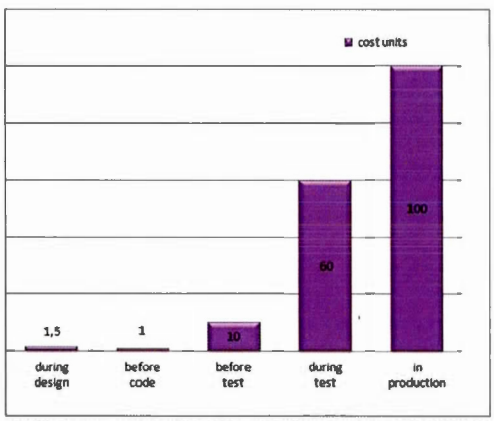


Figure 1 - Coût relatif de correction des défauts selon la phase de détection [SWEC]

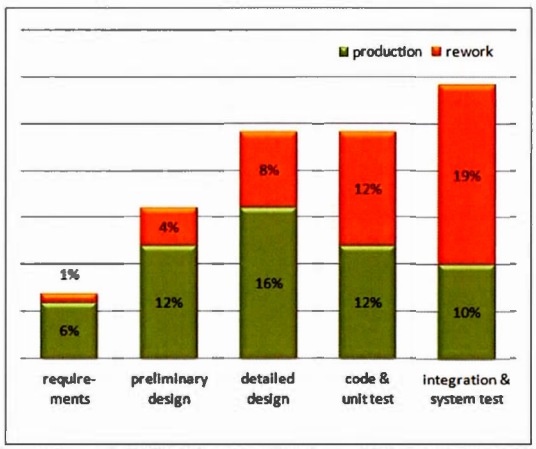


Figure 2 – Proportions de l'effort de correction (rework) dans un projet [SWEC]

3. Processus de revue de code

La section suivante détaille la façon de faire pour les revues du code. Cette procédure est autant importante pour l'auteur du code que le réviseur. Comme les listes de vérification sont connues, on s'attend à ce que l'auteur du code fasse lui-même une première revue de son code pour éliminer le plus possible de défauts à la source.

3.1 Quoi revoir?

La revue de code a plusieurs utilités. Durant l'inspection, on cherche à identifier des défauts réels ou potentiels du code inspecté dans le but d'en améliorer la qualité pour le bien de l'équipe. On veut entre autres évaluer les éléments suivants :

- La sécurité du code
- La structure des projets
- Le respect des normes de codage (clarté, lisibilité et maintenabilité)
- L'utilisation de pratiques reconnues et acceptées par l'équipe
- La présence et la qualité des tests (unitaire et intégration)

Lors de la revue, on va inspecter le code (client et serveur) qui a été produit par les développeurs. Nous ne faisons pas l'inspection du code généré par les outils ou des librairies qui ne nous appartiennent pas ou qui ne font pas partie du projet (ex. : librairies partagées). Par contre, une mauvaise utilisation de ces librairies peut constituer un défaut à reporter.

3.1.1 Listes de vérification

Dans le but d'avoir un accent sur les règles importantes que l'on souhaite voir appliquer et vérifier, vous trouverez en annexe du document une « checklist » d'inspection qui peut vous aider lors de vos revues. Veuillez vous assurer que vous comprenez bien les éléments de cette liste et que vous savez comment identifier les violations à ces règles dans le code en consultant la documentation qui accompagne la checklist. Il s'agit d'un document de départ qui évoluera avec le temps : si des défauts de même nature sont découverts souvent, ils pourront être ajoutés à la liste de vérification. Au contraire, si certains défauts ne sont jamais décelés parce que la technologie ou les techniques utilisées rendent cette règle caduque, elle pourra être retirée de la liste. Attention : cette liste est un minimum. Elle contient les éléments à valider de façon plus précise mais si vous détectez des défauts qui ne font pas partie de la liste, n'hésitez pas à les noter également.

3.1.2 Outil d'analyse de métrique de code

Une partie de l'analyse du code peut être automatisé à l'aide d'outils. Par exemple, l'utilisation des Codes Metrics dans VS permet d'obtenir en quelques secondes la liste des classes et des méthodes les plus problématiques. Voici quelques « guidelines » sur les métriques et les valeurs limites. Attention : il s'agit de suggestions et non pas valeurs absolues. Les seuils aident surtout à identifier les problèmes potentiels et à savoir où porter une attention particulière. Il faut en vérifier le contenu, faire preuve de jugement et ne pas faire (ou suggérer) un refactoring important du code seulement pour se conformer aux seuils suggérés de la métrique. Il faut que le changement ait un impact positif sur la qualité du code (lisibilité, testabilité, maintenabilité, sécurité, etc.).

Seuils suggérés :

Classes

- Nb de méthodes par classe : entre 1 et 10
- Niveau d'héritage (depth of inheritance) : moins de 5
- Couplage : le plus bas possible, à partir de 10, ça peut devenir problématique
- Lignes de code : moins de 500 lignes

Méthodes

- Couplage : le plus bas possible, à partir de 4, ça peut devenir problématique
- Lignes de code : idéalement, moins de 50 lignes, 200 maximum (truc souvent utilisé : si la méthode n'entre pas dans un écran, elle est probablement trop longue).
- Complexité cyclomatique :
  - Moins de 5 : ok
  - Entre 5 et 15 : porter une attention particulière, potentiel de problèmes à la maintenance
  - Plus de 15 : penser sérieusement à un refactoring (extraire une partie de la logique dans d'autres méthodes ou d'autres classes). Exception : une méthode avec un gros « Switch/case » pourrait avoir une complexité élevée (1 par case) mais demeurer facile à comprendre. N'hésitez pas à demander l'opinion de vos collègues si vous avez des doutes.

3.1.3 Analyse statique du code avec Resharper

Resharper fait tout un travail pour vous aider à identifier des problèmes potentiels avec le code. Lorsque vous êtes dans un fichier, regardez la liste des suggestions faites par Resharper et appliquez-les si elles ont du sens dans votre cas. Attention : n'appliquez pas une suggestion à la solution en entier à moins d'être convaincu que c'est nécessaire, car cela va générer une grande quantité de changement dans le prochain commit, ce qui compliquera la tâche du réviseur et les merges. Ne vous lancez pas dans ce genre de changement massif sans en parler à vos collègues d'abord et assurez-vous de faire une branche spécifiquement pour ce refactoring (n'incluez pas d'autres changements en même temps, car il sera difficile de déterminer ce que vous avez fait vs ce que Resharper a fait).

Truc : appliquez la règle du scout (boyscout rule [97Things]) : lorsque vous faites des modifications dans un fichier, laissez-le dans un meilleur état que quand vous l'avez trouvé en vous assurant que le code que vous ajoutez ne comporte pas de code issue et en réglant certains de ceux qui s'appliquent au fichier que vous touchez. Priorisez les avertissements avant de vous attaquer aux suggestions. Comme toujours, faites preuve de jugement : certaines des suggestions peuvent ne pas s'appliquer dans votre cas ou n'apportent rien en terme d'amélioration de code. Si c'est le cas, vous pouvez utiliser la fonction `Disable Once with comment` de Resharper et ajoutez au commentaire la raison pour laquelle cette règle ne s'applique pas dans votre cas précis. Ainsi, la prochaine personne à ouvrir le code saura que ce code issue a été vérifié et qu'aucune autre intervention n'est requise. Attention : on ne fait pas ça juste pour faire « taire » Resharper : la plupart du temps, les suggestions sont pertinentes et devraient être appliquées. L'inactivation d'une règle devrait être une exception et si ce n'est pas le cas, il faudrait peut-être réviser la règle et voir si l'on devrait la désactiver de façon permanente.

Un fichier de règles communes a été créé. Il se trouve dans TFS2015 à l'adresse suivante : `[retirée pour des raisons de confidentialité]`. Pour l'utiliser, faites un clone du dépôt `[retirée pour des raisons de confidentialité]`. Ensuite, allez configurer Resharper pour utiliser ce fichier comme fichier de règle local :

- Cliquez sur le menu Resharper et choisir Manage Options ...

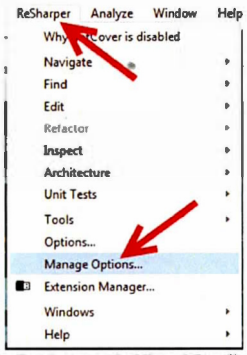


Figure 3 - Menu Resharper de Visual Studio

- Dans la fenêtre, cliquez sur le + à côté de This Computer, cliquez sur Open Settings File ... et aller chercher le fichier GlobalResharperRules.DotSettings à l'endroit où vous l'avez cloné :

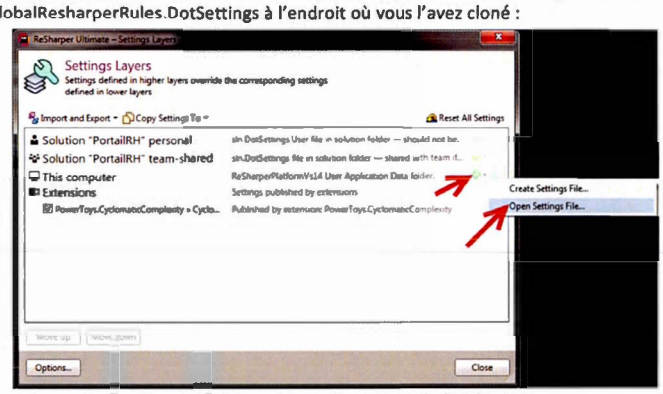


Figure 4 - Gestion de fichiers de configuration de Resharper

- En pointant Resharper directement sur le fichier de règle dans le dépôt Git, il sera facile de faire une mise à jour de votre fichier de règle (git pull) à la dernière version lorsque des règles seront ajustées.
- Finalement, nous devrons assurément désactiver certaines règles avec lesquelles nous ne sommes pas d'accord ou qui ne sont pas importantes pour nous. C'est un « work in progress ». Faites part à vos collègues des règles qui ne sont pas prioritaires ou au contraire, les règles qui ont impact important et qui pourraient voir leur priorité augmentée.

3.2 Qui revoit?

La revue de code doit se faire par un développeur qui n'est pas celui qui a produit le code. Le directeur affectera

un ou des réviseurs au projet au départ et idéalement, elles resteront les mêmes pour la durée du projet pour améliorer l'efficacité et pour simplifier le suivi des corrections. Ce peut être quelqu'un qui travaille sur le même projet ou non, pourvu qu'il travaille sur une autre section du code que celle à inspecter. Les 2 situations comportent des avantages et des inconvénients :

- Un développeur au projet prendra moins de temps à comprendre le contexte du projet. Par contre, il pourrait ne pas remarquer certains problèmes de clarté du code, car il est au courant du « langage » du projet (abréviations, structure, concept d'affaires).
- Inversement, un développeur externe au projet prendra plus de temps pour comprendre le contexte au début mais ce faisant, pourra identifier les parties du code difficiles à comprendre ou qui manque de documentation.

Dans un cas comme dans l'autre, les problèmes tendent à s'atténuer avec le temps.

3.3 Quand et durant combien de temps revoir?

La revue de code est une activité qui demande du temps et de la concentration. Pour une question d'efficacité, elle devrait se faire à intervalle régulier et pas seulement à la fin du projet (à moins bien sûr qu'il s'agisse d'une petite requête de quelques jours). Rendue à la fin, la tâche est beaucoup trop lourde et il est souvent trop tard pour corriger des problèmes de design importants dans l'application qui demanderait un lourd refactoring.

Diverses études sur le sujet en sont venues à la conclusion que le rythme d'inspection idéal pour le code se situe entre 250 et 1000 lignes de code par heure (on ne compte pas ici le code généré automatiquement par les outils), dépendant de la complexité du code inspecté. Un rythme plus rapide ne permet généralement pas de bien comprendre le code et d'y déceler un maximum de défaut. Il faut inspecter à un rythme assez lent pour avoir le temps de réfléchir et se demander non seulement ce qu'il y a d'incorrect dans le code inspecté mais aussi ce qui manque (validations, gestion d'erreurs, log, etc). Ce type de défaut peut être aussi important qu'un bogue dans le code existant.

Il a été également démontré qu'à partir d'un certain point, les inspections trop longues ne permettent pas de trouver plus de défauts. Comme la fatigue mentale se fait sentir après un certain temps et que cela amène une baisse normale du niveau de vigilance de l'inspecteur, il est recommandé de limiter les séances de revue à 1 heure puisque le nombre défauts trouvés tant à plafonner après 60 minutes (voir Figure 5).

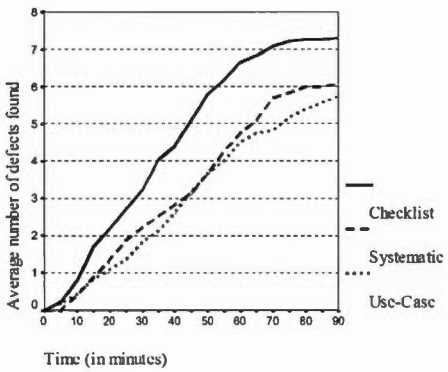


Figure 5 - Évolution de l'efficacité de la revue dans le temps [Smartbear]

C'est une bonne idée de planifier à l'avance la plage horaire que vous consacrerez à la revue pour vous assurer d'être le moins sollicité possible. Choisissez également une période de la journée où vous êtes le plus alerte. Certaines personnes préfèrent le matin tôt, d'autre plus tard dans la journée. Découvrez ce qui est le plus efficace pour vous.

L'intervalle idéal entre les revues dépend donc de la vélocité du projet. Posez-vous la question : après combien de jours le code aura changé suffisamment (entre 250 et 1000 lignes de code ajoutées / modifiées, selon les standards d'équipe ou la complexité du code) pour justifier une revue. Dans la plupart des cas, une revue hebdomadaire sera suffisante mais dans certains cas où le code change rapidement (gros projet ou en début de projet), il se pourrait qu'il faille faire des revues plus régulièrement. Il est également important, peu importe l'intervalle, de faire une revue de l'architecture en début de projet (car ce sont des changements difficiles à faire par la suite) et avant chaque livraison (à la fin du projet, d'un module ou d'un sprint, dépendant du processus de développement utilisé dans le projet).

Bien sûr, il s'agit d'une recommandation et il n'est pas nécessaire de s'en tenir strictement à cela. L'auteur pourrait demander une revue « ad hoc » s'il a des doutes sur une certaine partie de son code. L'inspecteur pourrait également prendre l'initiative de faire des revues un peu plus fréquentes en début de projet pour s'assurer que l'application part sur de bonnes bases. Servez-vous de votre jugement et en cas de doute, n'hésitez pas à demander l'opinion de vos collègues ou d'en discuter avec le directeur assigné au projet.

3.4 Comment revoir?

Le processus de revue du code se fait en 7 étapes. Celles-ci seront décrites dans le détail pour vous aider à bien le comprendre. Les 7 étapes de la revue sont :

- Préparer le code pour la revue
- Faire la demande de revue
- Faire la revue

- Revoir les commentaires
- Faire les corrections
- Réviser les corrections
- Ménage et documentation

Peu importe l'outillage utilisé, les étapes demeurent les mêmes. Dans le cas de notre équipe, nous allons utiliser les outils de révisions fournis par TFS2015. Voici une description sommaire du processus de revue avec cet outil :

> Les projets configurés avec TFS201S utilisent Git comme gestionnaire de code source. La méthode privilégiée par les utilisateurs de Git pour les revues est le principe de **Pull Request (PR)**. Un PR est un ensemble de commit dans une branche locale qu'on pousse dans une branche sur le serveur dans le but de la faire inspecter et qu'elle soit « mergée » dans une branche commune (master ou next la plupart du temps). Une requête de revue est alors envoyée à un réviseur ou des réviseurs qui feront la vérification du code et si tout est correct ou si les défauts trouvés sont mineurs, ils approuveront le PR et le code sera mergé dans la branche de destination. Si toutefois, il y a des corrections plus importantes à faire, le réviseur documente les changements à faire et refuse le PR. L'auteur doit alors prendre en compte les commentaires et pour chacun, corriger le problème ou expliquer pourquoi il pense que le commentaire du réviseur ne nécessite pas de changement.

Petit avertissement avant de commencer : Git est TRÈS permissif et vous permet de faire un peu n'importe quoi (détruire des branches non mergées sans avertissement sur le serveur si vous avez les droits par exemple). Soyez attentif à ce que vous faites et suivez les instructions, surtout au début, pour bien vous familiariser avec le processus.

Voici le détail de chacune des étapes

3.4.1 Préparer son code pour la revue

La première étape pour la revue commence avec la création d'une branche avant même d'avoir commencé votre modification. Pour faciliter la vie du réviseur, vous créez (ou utilisez) une branche locale à partir de la branche de destination (next ou master la plupart du temps). Cette branche peut porter le nom de la fonctionnalité ou du bogue sur lequel vous travaillez. Voir la documentation sur l'utilisation de git pour le détail des conventions de nom de branche. Cette branche ne sera utilisée idéalement que par vous, ça simplifie la vie au réviseur et ça facilite le ménage par la suite. Si vous n'avez pas publié votre branche sur le serveur encore et que vous vous rendez compte que vous avez oublié de créer une nouvelle branche pour votre modification, vous pouvez consulter l'article suivant pour vous expliquer comment déplacer des commits vers une autre branche. Avant de faire la demande de revue, vous devez avoir fait les actions et vérifications suivantes :

- Votre code est « committé » dans votre branche locale
- Vous avez fait un pull des branches serveur, fait les merges si nécessaire (pour intégrer le travail des autres développeurs) et tester le code résultant vous assurer que votre code fonctionne bien une fois intégré. Truc : Intégrez tôt et souvent pour éviter les mauvaises surprises au moment de la demande de PR
- Vous avez fait les vérifications des erreurs / avertissements de Resharper sur les fichiers modifiés
- Vous avez passé au travers de la liste de vérification pour les fichiers que vous avez touchés
- Les tests unitaires sur les classes modifiées ont été créées / mis à jour et exécuté avec succès

3.4.2 *Faire la demande revue*

En utilisant Visual Studio, faites une demande de PR. Dans Team Explorer, utilisez l'option ***Pull Requests*** et cliquez sur ***New Pull Request***.

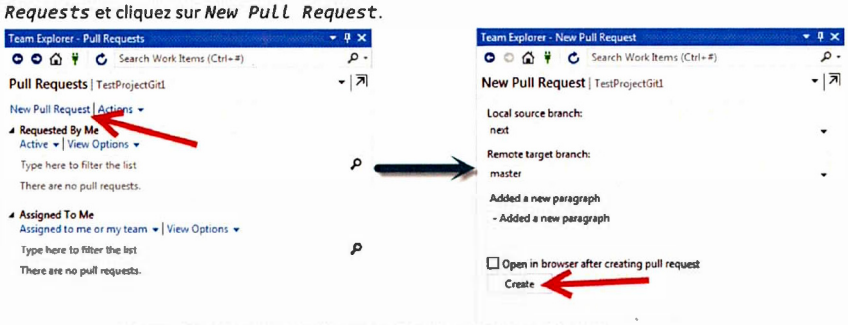


Figure 6 - Créer un Pull Request à partir de Visual Studio

Sélectionnez ensuite la branche source et la branche de destination. Attention : Visual Studio proposera par défaut ce qui semble lui avoir le plus de sens mais ce n'est pas nécessairement ce que vous voulez. Vérifiez.

Entrez un titre et une description significative expliquant le contenu du PR. La demande est maintenant créée dans TFS. Vous pouvez également, après un commit dans la branche à réviser, faire une demande directement via l'interface web de TFS :

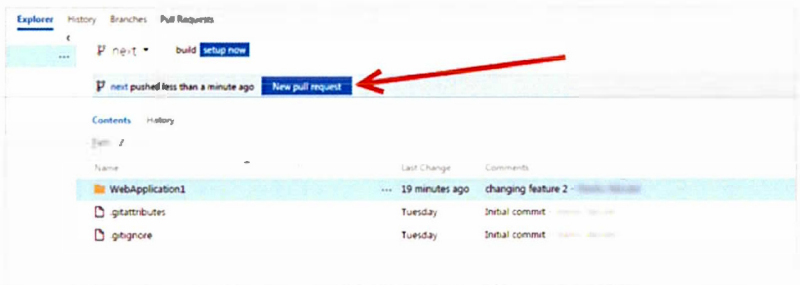
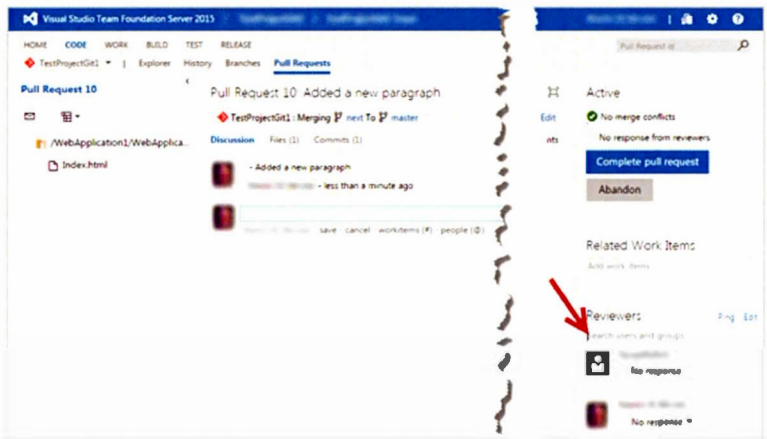


Figure 7 - Créer un pull request à partir de l'interface web de TFS

Vous devez maintenant assigner la revue à quelqu'un. Dans le cadre de votre projet, un ou des réviseurs vous seront attitrés pour faire la revue. Pour les informer qu'ils ont une revue à faire, aller dans le PR sur le site du projet (onglet ***Pull Requests***) et allez inscrire les réviseurs dans la boîte à cet effet en bas à droite :



Les réviseurs verront alors apparaître la demande de PR dans leur liste de PR à faire dans Visual Studio et TFS. **Attention** : la liste des PR ne présente que celles pour le team project en cours. Si la personne travaille dans un autre team project, elle ne la verra pas apparaître. Dépendant de leurs paramètres de notification, les réviseurs recevront également un courriel pour les aviser, ce qui est probablement la meilleure façon de savoir qu'une revue a été demandée. Si vous n'avez pas de nouvelle des réviseurs ou si vous voulez donner des instructions spécifiques sur le PR, utilisez le bouton *Ping* à côté de la boîte de recherche des réviseurs pour envoyer un courriel aux réviseurs. Vous pourrez sélectionner lesquels aviser et vous aurez la possibilité d'inscrire une note dans le message.

3.4.3 *Faire la revue*

En tant que réviseur, vous serez informé qu'une demande de PR vous a été assignée. Le courriel, l'item dans votre Visual Studio ou dans le site du projet vous mèneront directement à la demande (voir image ci-dessous). L'écran du PR est composé de plusieurs éléments :

1. La liste des fichiers inclus dans le PR
2. L'onglet discussions permet de garder un historique de ce qui s'est passé avec le PR (les commentaires entrés, les changements de statut, etc.). Comme à d'autres endroits dans TFS, vous pouvez utiliser les # (hastag) pour référencer un work item (la référence sera transformée en lien cliquable) et le @ (arobas) pour notifier une personne.
3. La liste des fichiers inclus dans le PR (avec les sections de fichier modifiées en évidence)
4. La liste des commits inclus dans le PR
5. Les branches source et destination pour le PR
6. Le statut de la revue par les réviseurs (votre liste de statut est modifiable)
7. Les boutons pour compléter le PR ou l'abandonner.

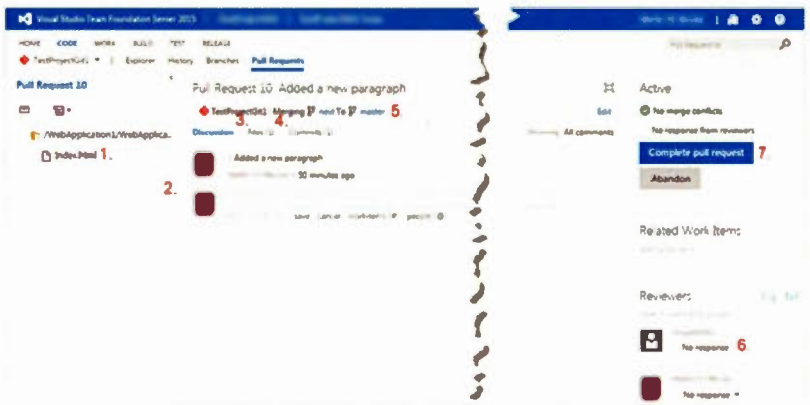


Figure 8 - Fenêtre principale d'un pull request

Pour faire la révision, cliquez sur un fichier (1.) ou la liste des fichiers (3.) pour voir les changements effectués dans les fichiers. La liste 3. ne montre que les sections modifiées tandis qu'en cliquant sur un fichier dans la liste 1, on peut voir le fichier en entier. Pratique pour avoir un peu plus de contexte.

Voici l'écran de révision pour un fichier :

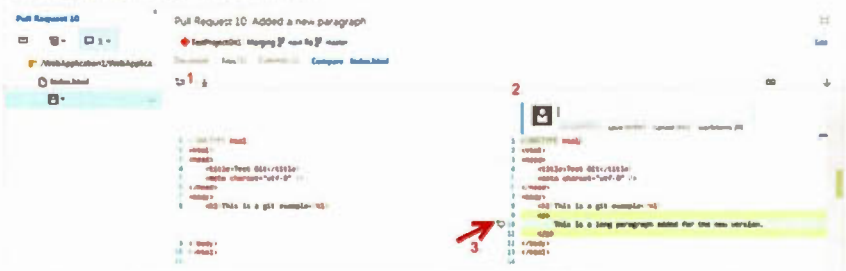


Figure 9 - Comparaison et commentaires sur un fichier

Vous pouvez remarquer quelques éléments importants :

1. Le bouton 1 Permet de faire un commentaire général sur le fichier. Cela ouvre la boîte en point 2.
2. La boîte permet d'entrer un commentaire général sur le fichier
3. Il est également possible d'inscrire un commentaire directement sur une ligne de code en cliquant sur la petite icône de commentaire qui apparaît lorsqu'on passe la souris sur les lignes du code.

Les commentaires doivent être clairs, concis mais contenir toute l'information nécessaire pour permettre à l'auteur de comprendre la nature du défaut détecté et de pouvoir prendre action pour le corriger. Un commentaire devrait avoir un format standard pour simplifier la compréhension et l'analyse des défauts. Le format doit être le suivant :

[(*Sévérité*) : (*nature*)] *Texte du commentaire.*

Par exemple : *[Critique : sécurité] La fonction GetEmployee n'est pas sécurisée correctement et permet d'afficher de l'information sur les employés pour des utilisoteurs non authentifiés.*

Les sévérités à utiliser sont les suivantes :

- **Critique (Critical)** : problème grave pouvant occasionner un crash de l'application, un bris de sécurité ou une corruption des données.
- **Majeur (High)** : problèmes dans le code qui pourraient empêcher l'utilisateur de compléter une action selon les spécifications ou qui peut occasionner des problèmes visibles à l'utilisateur.
    - Ex. : problème d'algorithme, erreurs dans les validations (incorrectes ou manquantes), code incomplet, problème de performance, memory leak, etc.
- **Medium (Medium)** : ici, 2 catégories de problème :
    - problèmes visibles à l'utilisateur mais qui ne l'empêche pas de compléter l'action. Ex. : message d'erreur manquant ou incorrect, erreur visuelle, option de menu dans le mauvais ordre, etc.
    - Cette catégorie doit également inclure les problèmes de qualité du code qui pourrait occasionner des problèmes importants de maintenance à moyen long terme (ex. : architecture incorrecte, mauvaise encapsulation, mauvaise gestion des dépendances, tests unitaires incorrects ou manquants).
- **Mineur (Low)** :
    - Problèmes non visibles à l'utilisateur mais qui pourraient occasionner de la confusion pour les autres développeurs ou la maintenance (ex. : conventions de nom, commentaires manquants ou pas clairs, formatage du code, etc.)

Pour ce qui est de la nature, utilisez celle disponible dans la liste de vérification (check-list) si le défaut noté y est présent. Sinon, utilisez votre bon jugement et utilisez un terme court (1 à 3 mots) pour catégoriser le défaut. Soyez consistants et utilisez le même terme pour les défauts de même nature.

Une fois tous les fichiers revus et les commentaires inscrits, vous devez déterminer si le code est assez de qualité pour être mergé dans la branche de destination. Pour cela, utilisez l'indicateur de réponse en bas à droite à côté de votre nom :

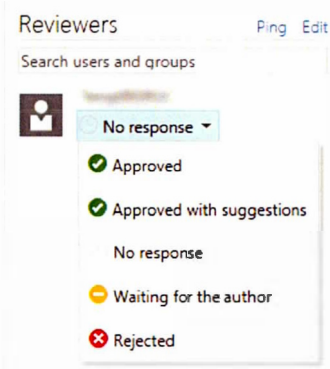


Figure 10 - Liste des statuts de revue

3.4.3.1 *Pull request approuvé*

Si le code est « parfait » ou que les commentaires sont sur les éléments mineurs, utilisez les 2 premières options. Dans ce cas, utilisez le bouton `Complete pull request` pour terminer le merge. L'écran suivant vous sera présenté :

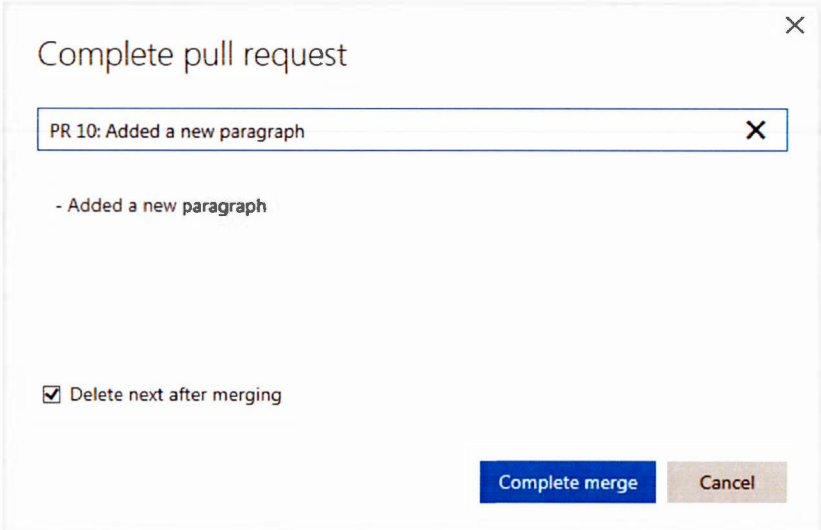


Figure 11 - Fenêtre de confirmation pour terminer le pull request

Petite précision ici : faites attention à la case à cocher ***Delete [branchname] after merging***. Si le PR a été fait d'une branche temporaire vers une branche permanente (ex. : de NewFeatureA vers Next) et que le travail est complété, vous pouvez détruire la branche. Par contre, si le travail est partiel ou qu'il s'agit d'un merge entre 2 branches permanentes (ex : next vers master), ne détruisez pas la branche source. Si jamais vous le faite par erreur, ce n'est pas la fin du monde puisque Git les modifications seront dans la branche de destination et que vous pouvez toujours recréer la branche. Par contre, vous pourriez causer quelques soucis à vos collègues qui pointent sur cette branche. Soyez vigilant. Notez également que l'interface ne vous empêchera pas de compléter le PR même si le statut des réviseurs est ***Rejected***. Vérifiez que les 2 « voyants » sont au vert avant de complété le PR

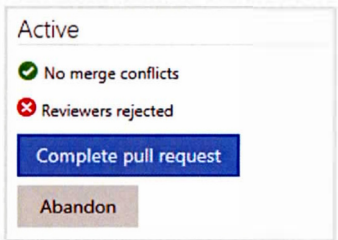


Figure 12 - Indicateurs de statut du pull request

3.4.3.2 *Pull request en attente (Waiting for the author)*

Si quelque chose n'est pas clair, vous pouvez utiliser le statut ***Waiting for the author***. Dans ce cas, assurez-vous de décrire dans la discussion ce que vous attendez de l'auteur (une clarification, un bout de code manquant, etc.). Dans ce scénario, ne cliquez sur aucun des boutons (***Complete pull request*** ou ***Abandon***), car le processus de vérification se poursuivra.

3.4.3.3 *Pull request rejected*

Dans le cas où il y a des erreurs qui doivent être corrigées avant le merge, utilisez ***Rejected***. Ici, 2 scénarios sont possibles : s'il y a des corrections à faire qui ne sont pas trop importantes, on peut laisser le PR ouvert et attendre les corrections de l'auteur. Si toutefois les changements sont trop importants ou s'il y a un problème avec les branches (pas la bonne source et/ou pas la bonne destination), il est parfois plus simple de rejeter carrément le PR (en utilisant le bouton Abandon). L'auteur devra alors faire les corrections nécessaires et soumettre un nouveau PR. Faites attention avec le bouton ***Complete pull request***. Tel que spécifié plus haut, TFS ne vous empêchera pas de merger le PR même si la revue est rejetée. Ça peut être pratique dans certains cas (MEP rush) mais ça devrait être l'exception et non la règle.

3.4.4 *Revoir les commentaires*

SI la revue n'est pas acceptée telle quelle, vous recevrez un courriel vous avisant du statut du PR. En cliquant sur le lien dans le courriel, vous serez redirigé vers l'écran de discussion où vous pourrez consulter les commentaires.

Figure 13 - Discussion sur les commentaires suite à une revue

Les commentaires des réviseurs sont alors bien visibles. En tant qu'auteur, vous pouvez commenter certains des points de la revue (s'ils ne sont pas clairs ou si vous n'êtes pas d'accord). Après discussion avec le réviseur, peut-être que certains éléments seront éliminés des corrections à faire. Dans ce cas, utilisez la boîte déroulante de statut (en bas à droite des commentaires) et mettez-le à *Won't fix*. Toutefois, pour tous les cas de *Won't fix*, un commentaire devrait être inscrit pour expliquer aux réviseurs pourquoi la suggestion ne sera pas faite. Attention : si vous ne prévoyez pas faire de changements dans le code et que vous ne faites que changer le statut des commentaires, il n'y aura pas de notification envoyée aux réviseurs. Dans ce cas, utilisez un courriel ou la fonction *ping* pour les informer que des changements dans le PR ont été faits.

Généralement, la révision des commentaires se fera individuellement par l'auteur. Toutefois, si plusieurs défauts similaires ont été trouvés ou si des explications supplémentaires semblent nécessaires, n'hésitez pas à organiser une courte rencontre pour discuter des points problématiques, surtout si vous devez explique une notion qui n'est pas bien comprise. Il est important que l'auteur comprenne la nature et la raison des changements demandés, non seulement pour corriger le code problématique dans la revue en cours mais aussi pour pouvoir l'appliquer par la suite. Il se peut aussi que l'auteur ait une raison légitime d'avoir fait le code de cette façon dans quel cas, le réviseur pourra en tenir compte et par le fait même, découvrira peut-être une façon de faire qui lui était inconnue. La revue est aussi un outil d'apprentissage, tant pour l'auteur que le réviseur. Faites de ces discussions quelque chose de constructif et non une bataille entre deux façons de faire qui sont peut-être aussi bonne une que l'autre. En cas de doute, n'hésitez pas à consulter vos collègues pour avoir leur opinion sur le sujet.

3.4.5 *Faire les corrections nécessaires*

L'auteur retourne alors dans son code pour faire les corrections nécessaires. Il n'a pas besoin de faire un nouveau PR (si ce dernier n'a pas été complété ou abandonné), il doit tout simplement faire des commits

et pousser son code dans la branche source du PR. Ce faisant, TFS détecte automatiquement que de nouveaux changements ont été inclus dans le PR :

Figure 14 - Indication de changements dans un pull request actif

Les réviseurs sont alors informés que des changements ont été faits et que la révision peut reprendre. Avant de terminer, prenez quelques minutes pour aller dans les commentaires mettre à `Resolved` tous ceux que vous avez adressés dans le commit.

3.4.6 *Réviser les corrections*

Le réviseur peut retourner dans le PR et voir la dernière version du code avec les corrections et les commentaires entrés précédemment. Dépendant du statut du commentaire, l'action à porter sera différente :

- `Active` : il s'agit probablement de commentaires non résolus ou l'auteur a oublié de les mettre à `Resolved`. Vérifiez avec l'auteur si c'est volontaire ou un oubli. Si c'est un oubli, vous pouvez revoir la correction et le mettre directement au statut `Closed`. Sinon, l'auteur n'a pas terminé son travail.
- `Resolved` : vérifier si la correction est satisfaisante. Si c'est le cas, le réviseur peut changer le statut du commentaire à `Closed`. Si ce n'est pas le cas, il doit être remis à `Active` et un commentaire doit être ajouté pour indiquer ce qui cloche dans la correction.
- `Won't fix` : le réviseur peut tout simplement les mettre au statut `Closed` ou les remettre à `Active` s'il n'est pas d'accord.

En temps normal, le réviseur ne devrait pas changer le statut de révision (en bas à droite) et compléter le PR tant que tous les commentaires ne sont pas au statut `Closed`. Toutefois, comme décrit plus haut, TFS ne vous empêchera pas de compléter le PR même si des commentaires restent ouverts ou que le statut de révision est à `Rejected`.

Ici aussi, si vous ne faites que modifier le statut des commentaires sans changer le statut du PR ou exécuter une action (Compléter ou abandonner), l'auteur ne sera pas notifié qu'un changement a été effectué. Informez-le via courriel ou la fonction `Ping`.

Si tout est conforme, utilisez l'option `Complete Pull request` et la requête est terminée. Sinon, le processus reprend à l'étape 4 (revoir les commentaires)

3.4.7 *Ménage et documentation*

La dernière étape du processus de revue est le ménage et la documentation. À cette étape, l'auteur et les réviseurs ont quelques tâches à accomplir :

3.4.7.1 *L'auteur*

Une fois le PR complété, l'auteur doit mettre à jour son code local et faire le ménage dans ces branches. Voici les 3 scénarios les plus fréquents :

1. PR d'une branche permanente vers une autre branche permanente (ex : next vers master). Dans ce scénario, la seule chose à faire est de faire un pull de la branche de destination (ex. : master) pour vous assurer d'avoir les derniers changements, incluant le contenu du PR. Ce n'est pas obligatoire mais ça peut vous faciliter la vie plus tard, car votre next et votre master étant maintenant identiques, les différences entre les 2 branches seront limitées aux nouveaux changements que vous ferez.
2. PR d'une branche temporaire (non complétée) vers une branche permanente (ex. : featureA vers next). Ce scénario est identique au précédent. Lors du PR, la branche restera sur le serveur et vous l'aurez encore localement. Ici aussi, assurez-vous de faire un pull sur la branche de destination pour être à jour.
3. PR d'une branche temporaire (complétée) vers une branche permanente (ex. : FeatureA vers next). Si le travail est complété, on veut faire le ménage dans les branches et retirer celle qui n'est plus utile (la branche source). Dans un premier temps, faire un pull de la branche de destination (next) comme dans les scénarios précédents. Ensuite, vous pouvez détruire la branche FeatureA localement. Si vous des commit non mergé localement ou qui n'ont pas été poussé vers un remote, un message d'avertissement sera affiché comme quoi vous risquez de perdre des commit. Si vous avez ce message, soyez certain que les changements non mergé ne sont pas importants, car ils seront perdus.

   Pour détruire la branche sur le serveur, la façon la plus simple est au moment de complétér le PR. En tant que réviseur, vous pouvez cocher la case *Delete Branch after merge* (voir Figure 11) mais si vous n'avez pas créé la branche ou si vous n'êtes pas un admin, vous ne pourrez pas détruire la branche. Toutefois, lorsque le PR est complété, l'auteur peut aller détruire la branche lui-même à l'aide du bouton *Delete Source Branch* dans le PR :

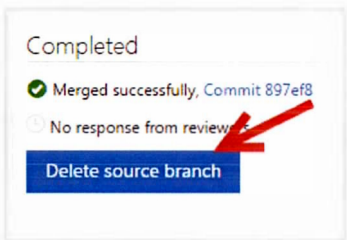


Figure 15 - Détruire la branche source du serveur après un pull request complété

Vous pouvez également détruire des branches sur le serveur via Visual Studio :

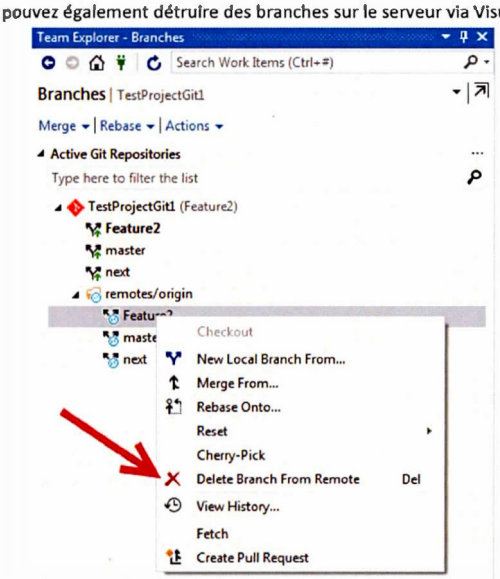


Figure 16 - Détruire une branche sur le serveur à partir de Visual Studio

Attention ! VS ne vous donnera pas de warning si des commits sur le serveur ne sont pas mergés dans une autre branche. Vous perdrez alors du code. Vérifiez l'historique de la branche avant de procéder.

3.4.7.2 *Les réviseurs*

Dans le cas des réviseurs, ceux-ci ont la responsabilité de maintenir à jour la base de connaissances. Selon les cas trouvés, ils peuvent demander des ajustements à la checklist de vérification et/ou dans les règles de Resharper (activer / désactiver ou changer la sévérité d'une règle). Ces suggestions devront être consignées dans la liste de suggestion des revues par les pairs et feront l'objet de discussions lors des rencontres de développeurs ou de rencontres spécifiques au besoin.

3.5 Mesures

Avec les revues de code, nous voulons mettre en place certains métriques pour évaluer la productivité des revues de code. Il ne s'agit de déterminer qui fait les revues les plus rapides (ça n'a aucune importance, car les données générées seront agrégées) mais plutôt d'évaluer si (et combien) le temps passé à faire des revues réduit l'effort de correction dans l'équipe, ce qui est un des objectifs premiers du Centre Rx pour mettre en place des revues. La plupart des informations nécessaires à ces mesures sont obtenues automatiquement par TFS ou d'autres outils. Par contre, 2 informations doivent être entrées manuellement par les membres de l'équipe :

3.5.1 Nouvelle activité dans la feuille de temps

Une nouvelle activité *Revue par les pairs* a été ajoutée à la feuille de temps (Revue par les pairs). Veuillez consigner le temps que vous passez à faire de la revue dans cette activité, pour le projet qui a nécessité le changement de code. Cette activité s'applique aux réviseurs. L'auteur continue de mettre son temps dans l'activité de développement, même pour les corrections faites suite à la revue.

3.5.2 Nouveaux champs dans les bogues dans TFS2015

Par la revue, on veut réduire le temps de correction (c'est-à-dire, les bogues trouvés lors des tests par les analystes, les clients ou les utilisateurs). On doit donc trouver une façon de le mesurer. Pour cela, on continue d'utiliser les bogues de TFS pour documenter les défauts trouvés (qui ont passés au travers du processus de revue sans être détecté) mais 5 informations supplémentaires devront être fournies (voir Figure 17) :

1. La sévérité du défaut. La sévérité doit respecter les mêmes critères que ceux utilisés lors de la revue. Voir la section 3.4.3 pour les détails.
2. À quel moment le bogue a été trouvé (*Found during*). Il y a 5 valeurs possibles à ce champ (*Requirements, Specifications, Development, QA, Production*). Dans cette phase du processus de revue, nous nous concentrons sur le code donc les bogues devraient avoir été détectés durant les 3 dernières étapes. Pour vous aider, un petit guide pour vous aider à déterminer quelle valeur utiliser :
    a. *Development* : en tant que développeur, vous avez trouvé un bogue dans le code DE QUELQU'UN D'AUTRE durant votre développement ou vos tests locaux. On ne documente par les bogues trouvés dans son propre code lors du développement, des tests unitaires ni des tests locaux (ça fait partie intégrante des activités liées au développement). De plus, on ne documente pas comme un bogue des éléments trouvés lors des revues. On considère comme un bogue un défaut dans le code qui n'a été détecté ni par le développeur qui l'a fait ni par le réviseur.
    b. *QA testing* : en tant qu'analyse, client, testeur (ou développeur participant à l'effort de QA), vous avez trouvé un bogue durant vos vérifications. C'est le cas le plus fréquent aujourd'hui.
    c. *Production* : peu importe le rôle, un bogue a été détecté en production et est documenté pour correction future. On ne documentera pas les bogues critiques (à régler sur le champ) durant la crise mais si le problème est relié à un problème dans le logiciel, il serait intéressant de le faire une fois le problème résolu pour qu'on puisse éventuellement en tenir compte dans des revues futures. Cela pourrait également faire l'objet d'une suggestion dans la base de connaissances.
3. La source du bogue (*Bug source*) : lorsque le développeur investigue le bogue, il doit tenter d'en déterminer la source. La source peut être déterminée à la création du bogue par l'analyste ou le testeur si elle est évidente mais c'est généralement lors de la recherche par le développeur que l'on pourra confirmer cette information. Il y a 3 sources possibles (*Requirements, Specifications,*

Development). Voici un guide pour choisir la bonne valeur

a. *Requirements* : on assignera cette source si le défaut est dû au fait que le client a changé d'idée ou qu'un des besoins du client n'a pas été documenté (plus rare au niveau des bogues, cela devrait normalement générer de nouvelles tâches).
b. *Specifications* : on assignera cette source si le défaut a été causé par des instructions manquantes, incomplètes ou incorrectes dans les tâches qui ont servi à produire le code fautif
c. *Development* : défaut dans l'implémentation.

4. La version dans laquelle on a trouvé le problème (*Found in version*) : autant que possible, essayer de regrouper les bogues par version. Avec TFS2015, les assemblies sont versionnées automatiquement durant les builds et le numéro de ce build est disponible dans les attributs de la dll (voir Figure 18). Si vous n'avez pas de numéro de version spécifique, utilisez un ID qui aura du sens pour les gens du projet. Ça peut-être le numéro de requête, le nom du la version, le numéro du build, etc. L'important, c'est de tenter de regrouper ensemble les bogues d'une même livraison.
5. Temps total de correction (*Completed works*) : nombre d'heures qui ont été requises pour corriger le problème. Ce qu'on veut recueillir est le temps TOTAL pour la correction c'est-à-dire : documenter le bogue, chercher la cause, corriger, retester. Donc, comme le bogue peut changer de main quelques fois, vous devez **additionner** votre temps à celui déjà présent dans le champ, arrondi aux 15 minutes. Ex. :

Activité	Temps requis	Completed works
Documenter le bogue	10 minutes (arrondi 15 minutes)	0.25
Rechercher la cause	30 minutes	0.75
Développement (correction, tests unitaires, tests locaux)	1 heure	1.75
Test de la correction par le QA	30 minutes	2.25
Total		2h15

Si le problème n'est pas résolu, on continue d'accumuler du temps :

Activité	Temps requis	Completed works
Documenter le bogue	15 minutes	0.25
Rechercher la cause	30 minutes	0.75
Développement (correction, tests unitaires, tests locaux)	1 heure	1.75
Test de la correction par le QA	30 minutes	2.25
Documenter le résultat du test. Non corrigé	15 minutes	2.50
Développement	30 minutes	3.00
Re-test de la correction	30 minutes	3.50
Total		3h30

Figure 17 - Écran de saisie d'un bogue

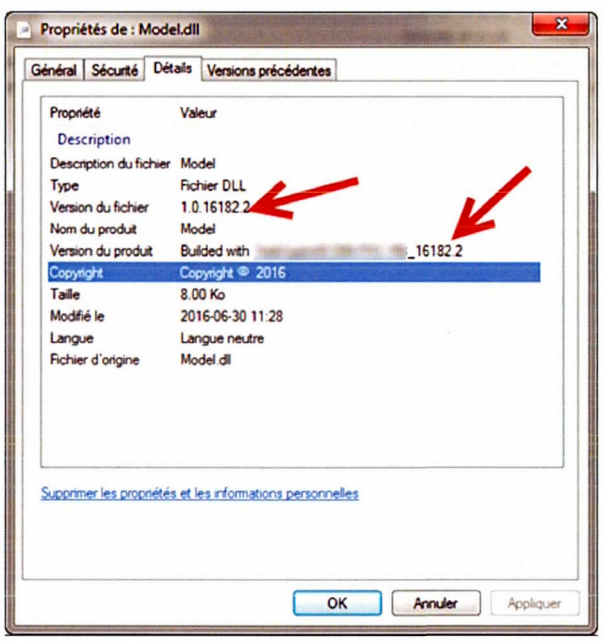


Figure 18 - Version de l'assembly dans les attributs de la *dll*

3.6 Rétroaction

Le processus de revue de code est évolutif comme tout ce qu'on fait en logiciel. Avec le temps, les outils s'améliorent ou l'on trouve des façons plus efficaces de faire les choses. C'est pourquoi il est important de communiquer avec vos collègues si vous avez des améliorations au processus à suggérer ou si vous éprouvez des difficultés. On veut tirer quelque chose de positif des revues, en faire un outil d'apprentissage mais aussi d'amélioration continue.

Toutes les idées sont les bienvenues et elles seront discutées en équipe lorsque des rencontres de développeurs ou de rencontres spécifiques si nécessaire. C'est un processus qui vous appartient, tout ce qui peut le rendre plus efficace tout en conservant (ou en améliorant) les objectifs de qualité ne peut qu'être bénéfique pour tout le monde.

3.7 Trucs et astuces

- Même si beaucoup d'opérations sur git sont possibles dans Visual Studio et TFS, l'implémentation n'est pas complète et plusieurs commandes ne sont pas disponibles. La ligne de commande de git est riche en fonctionnalité et si vous ne trouvez pas la fonctionnalité voulue dans VS / TFS, vérifiez la documentation de git, elle vous sera utile.
- Les outils évoluant sans cesse, ne vous arrêtez pas aux petits irritants mais essayez de travailler de la façon la plus efficace possible. Comme spécifié plus haut, faites-nous part de ces irritants, nous travaillerons à trouver une solution ou une alternative si possible.

4. Annexe

- Liste de vérification (check-list)

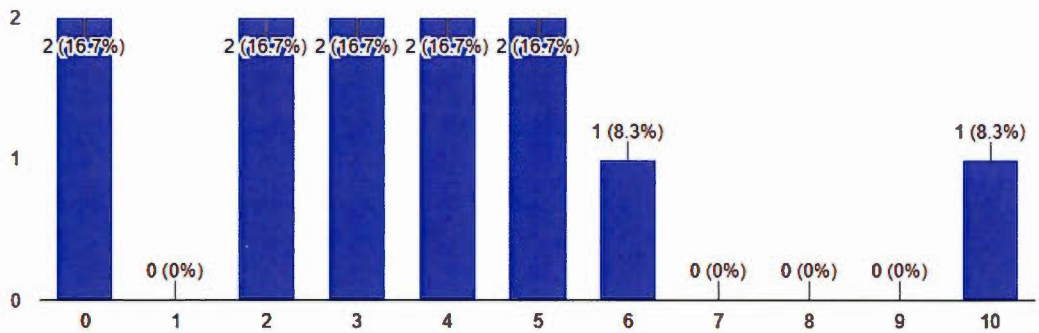
Références

- Trudel, S. (2005). Méthode d'inspection, CRIM.
- Gilb, T. and D. Graham (1993). Software Inspection, Addison-Wesley.
- Wiegers, K. E. (2002). Peer Reviews in Software, Addison-Wesley.
- Llie, M (2013, 8 janvier). "Code review guidelines", Récupéré le 11 octobre 2016, de http://www.codeproject.com/Articles/524235/Codeplusreviewplusguidelines
- McConnell, S. (2004). Code Complete (2nd edition), Microsoft Press.
- Sprunck, M. (2012, 16 octobre). "Selected Rules of Thumb in Software Engineering." Récupéré le 30 mars, 2017, de http://www.sw-engineering-candies.com/blog-1/rules-of-thumb-in-software-engineering.
- Martin, R. C. (2009). "The Boy Scout Rule." Récupéré le 1er avril, 2017, de http://programmer.97things.oreilly.com/wiki/index.php/The_Boy_Scout_Rule.

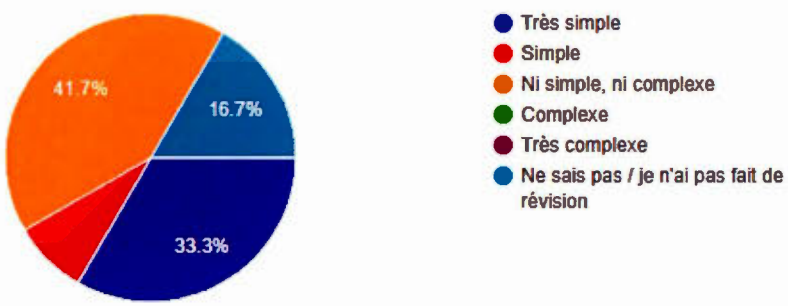
ANNEXE VI

SONDAGE DE SATISFACTION ET RÉSULTATS

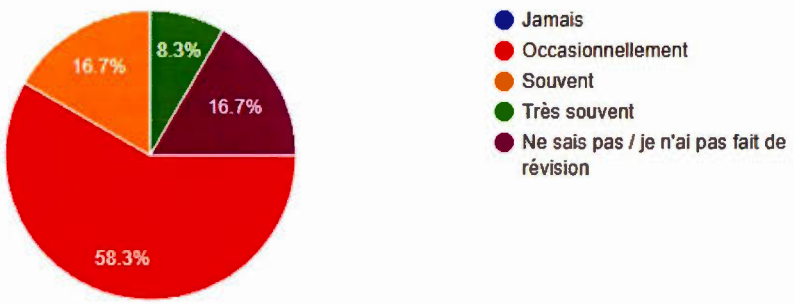
1. En tant que réviseur, à combien de revues avez-vous participé ?



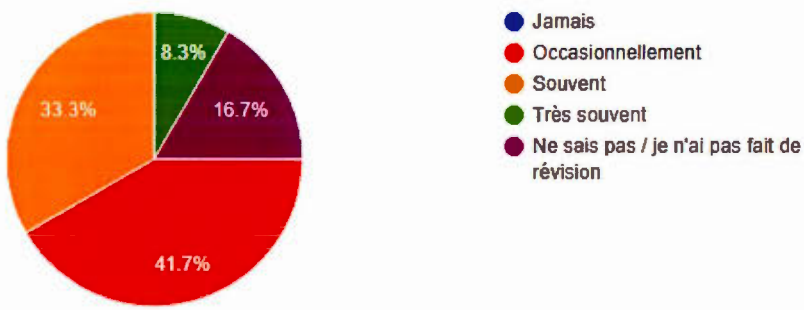
2. En tant que réviseur, comment évaluez-vous la complexité du processus de revue actuel ?



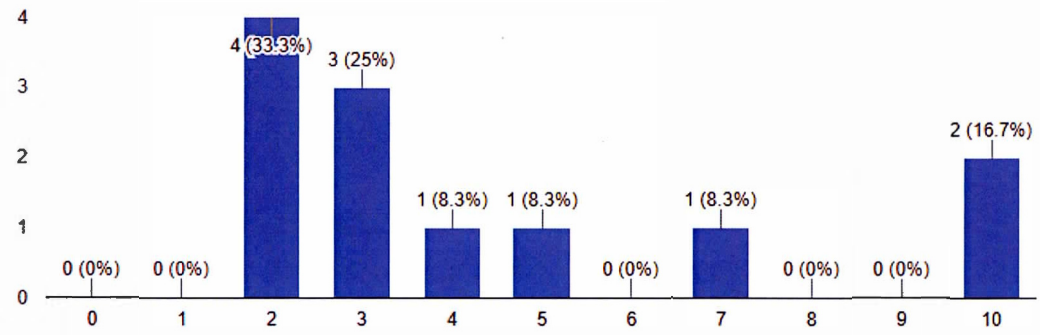
3. En tant que réviseur, à quelle fréquence le processus de revue vous a-t-il permis de partager des connaissances ou compétences techniques avec vos collègues ?



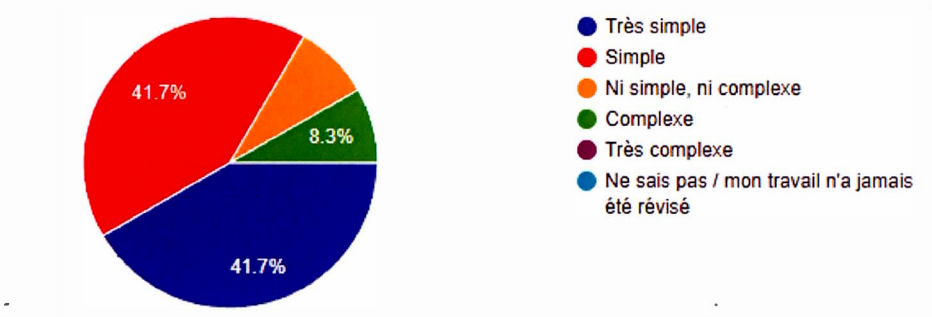
4. En tant que réviseur, à quelle fréquence le processus de revue vous a-t-il permis d'acquérir des connaissances sur le fonctionnement d' applications qui vous étaient alors inconnues ou que vous connaissiez très peu ?



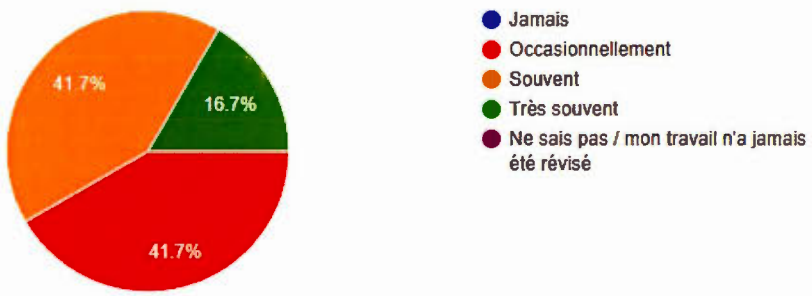
5. En tant qu'auteur, à combien de revues avez-vous participé ?



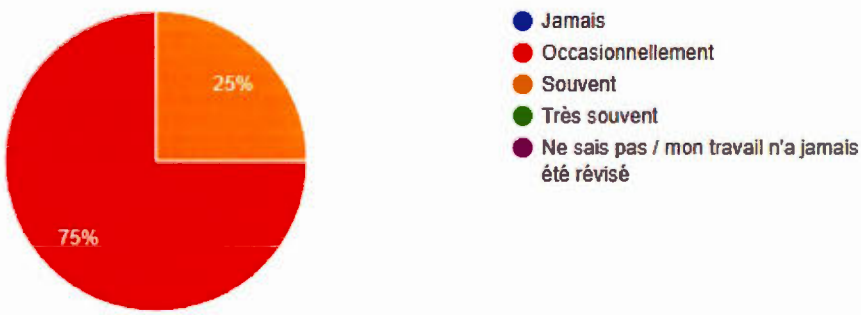
6. En tant qu'auteur, comment évaluez-vous la complexité du processus de revue actuel, particulièrement pour obtenir et gérer le feedback reçu lors des revues ?



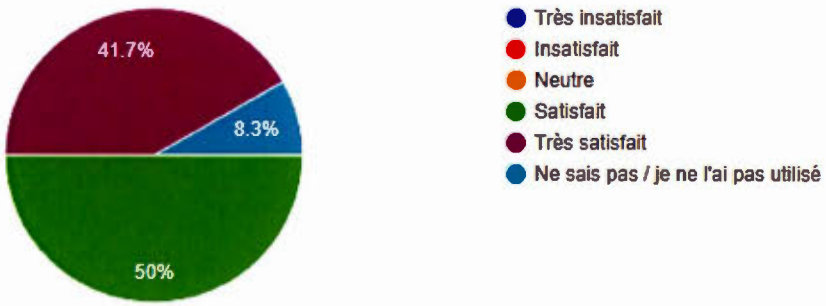
7. En tant qu'auteur, à quelle fréquence le processus de revue vous a-t-il permis d'acquérir des connaissances ou compétences techniques grâce aux suggestions et commentaires de vos collègues ?



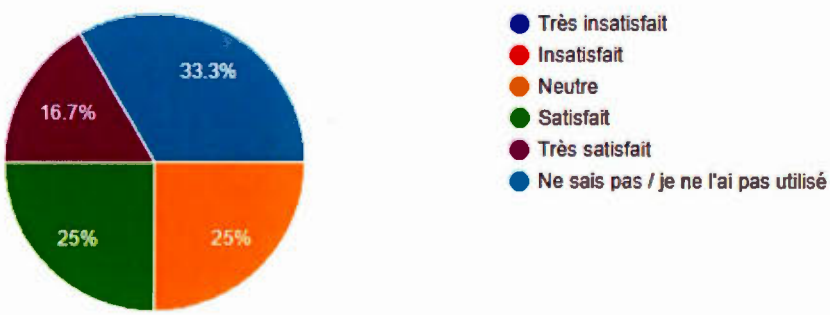
8. En tant qu'auteur, à quelle fréquence le processus de revue vous a-t-il permis de partager avec vos collègues des connaissances sur les applications sur lesquelles vous avez travaillé ?



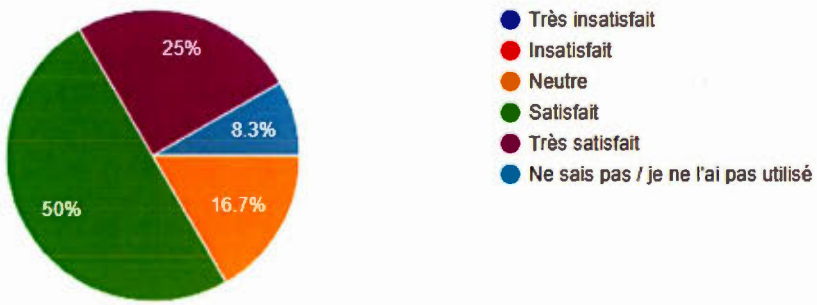
9. Quel est votre degré de satisfaction avec l'outil de révision (TFS2015)



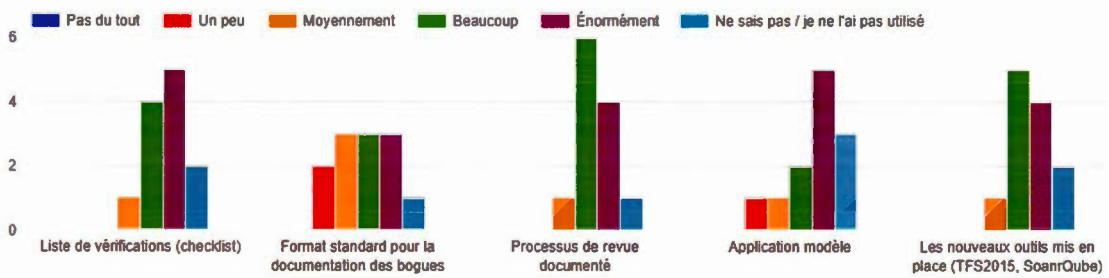
10. Quel est votre degré de satisfaction avec l'outil de suivi de la qualité (SonarQube)



11. Quel est votre degré de satisfaction avec la liste de vérification utilisée lors des revues ?



12 .Selon vous, indiquez en quoi les éléments suivants contribuent (ou non) à rendre le processus de développement plus uniforme d'un projet à l'autre :



13. Qu'avez-vous appris de votre expérience avec le processus de revue ?

Le partage de connaissances est intéressant. À la fois pour le développeur que pour le reviewer.

Intéressant de voir une autre perspective sur notre code ou sur le code d'un collègue.

Le processus m'a permis d'accélérer ma démarche pour rendre mon travail moins "brouillon" et dispersé.

Une fois que l'égo est mis de côté il n'y a que du positif.

Qu'il y avait plusieurs choses que je ne faisait pas de la bonne façon. Ça été très bénéfique pour moi.

Plus efficace si fait sur une base régulière en cours de développement et l'utilisation de tfs2015 rend l'expérience plus dynamique

14. Avez-vous des suggestion à faire pour améliorer le processus de revue (à tous points de vue) ?

1. Le processus doit toujours être présenté comme étant bidirectionnel.

2. SonarQube doit être mis de l'avant plus en détails. L'outil présente beaucoup d'informations intéressantes à différents degrés, mais il semble négligé. Les code smells moins triviaux que les références non-utilisées sont généralement significatifs.

ANNEXE VII

PRÉSENTATION DES RÉSULTATS À LA DIRECTION

17-08-19

Processus de revues par les pairs

Présentation des résultats - direction

Martin Bérubé

Date de la présentation : 2017-07-31

Révision : 1

Centre d'information Ri ître

Contenu

- Rappel du projet et des objectifs
- Données sur les revues
- Résultats du sondage
- Atteinte des objectifs
- Retombées pour l'équipe
- Travaux futurs
- Confidentialité des résultats pour remise à l'UQAM
- Questions, commentaires, suggestion

2

Quelques définitions...

- Problème potentiel : Élément potentiellement incorrect découvert dans un artéfact (document ou code) durant le processus de revue.
- Défaut : Élément incorrect dans un artéfact (document ou code) découvert après la revue (ce qui a passé au travers des mailles du filet)
- Phase d'injection : Phase (définition des besoins (I01), analyse fonctionnelle (D16) ou développement) où le défaut a été introduit.
- Phase de détection : Phase où le défaut a été détecté. Plus un défaut est détecté « loin » de sa phase d'injection, plus il est coûteux de le corriger.
- Temps de rework : Temps nécessaire pour documenter, corriger et retester un défaut.
- PFC : Point de fonction COSMIC

3

Rappel des objectifs du projet

- Assurer que les bonnes pratiques de sécurité du code soient systématiquement appliquées et vérifiées
- Réduire l'effort de corrections de défauts trouvés par les analystes lors des tests intégrés et les clients (business et grand public)
- Augmenter la qualité interne du code et des documents pour réduire l'effort de maintenance
- Transfert de connaissances sur les applications
- Partage de compétences techniques
- Uniformiser les pratiques de développements (standards et bonnes pratiques)

4

Rappel des éléments du projet

- Évaluer comment les revues étaient effectuées avant la révision du processus
- À partir des informations recueillies, mettre en place un nouveau processus de revue documenté
- Installer les outils nécessaires et documenter leur fonctionnement
- Former les participants sur le nouveau processus de revue
- Prendre et comptabiliser les mesures sur les revues effectuées
- Documenter et présenter les résultats

5

Données sur les revues (résumé)

De janvier 2017 à juillet 2017, 15 projets ont été analysés (4 retirés du rapport final, car ils n'étaient pas assez avancés). Pour les 11 projets retenus, il y a eu

- 101 heures de revues documentées.
- 331 problèmes potentiels ont été détectés et pour la plupart corrigés.
- Suite aux revues, 529 défauts ont été détectés lors du QA (aucun en production pour les projets analysés durant les 6 mois de l'initiative).
- 28 % des défauts détectés proviennent des spécifications (D16, D21), 72 % proviennent du développement.
- La qualité des données a été un enjeu important durant le projet.

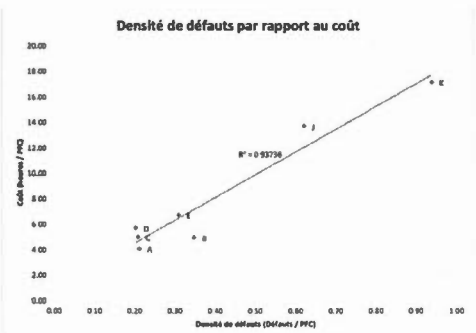
6

Données sur les revues (résumé)

- La densité de défauts détectés (et par conséquent, le temps de rework associé) semble être le meilleur prédicteur du coût d'un projet.
- Le temps de correction moyen par défaut semble influencé par la qualité du code (tendance à confirmer, car données fragmentaires, 7 projets seulement)
- Pour l'instant, aucun impact statistiquement significatif de la revue sur la densité de défauts détectés, la qualité du code ou le temps de rework. Hypothèses :
  - Qualité des données (taille fonctionnelle, nombre de défauts, temps de rework)
  - Manque d'expérience des réviseurs (connaissance du processus, vitesse d'inspection trop élevée, liste de vérification non intégrée)

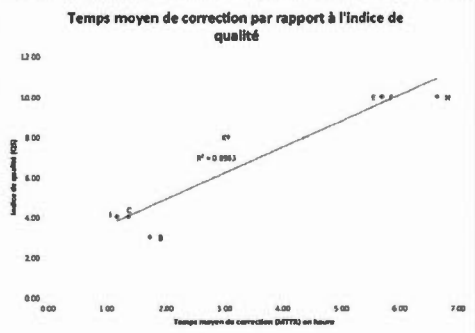
7

Données sur les revues (résumé)



8

Données sur les revues (résumé)



9

Données sur les revues (détails)

Code	Info Projet			Taille et effort			Défauts		Revues							Rework			
	Fia.	Type	Size	PPC	HT	HT / PPC	Nb	Nb / PPC	HDR	PP	Eff.	PP / HDR	PP / PPC	Vit. Revue	% HDR / HT	Hther	%Hther / HT	Hther / PPC	MTTR	QS
A	B	A	G	531	2133	4.01	112	0.21	15	56	33.33%	3.73	0.11	35.40	0.70%	337	15.80%	0.63	3.01	N/A
B	C	B	M	52	253	4.88	18	0.35	2.5	45	71.43%	18.00	0.87	30.80	0.99%	31.00	12.25%	0.60	1.72	3
C	A	A	M	106	521	4.92	22	0.21	14	140	88.42%	10.00	1.32	7.57	2.69%	29.75	5.71%	0.28	1.35	4
D	B	B	P	40	222.5	5.63	8	0.20	0	0	0.00%	0.00	0.00	0.00	0.00%	11.75	5.28%	0.29	1.47	N/A
E	C	B	P	16	86.25	4.95	4	0.31	0.75	6	60.00%	8.00	0.38	21.33	0.87%	32.75	26.34%	1.42	5.69	10
F	D	B	M	85	641.38	7.55	17	0.20	0	0	0.00%	0.00	0.00	0.00	0.00%	96.99	15.12%	1.14	5.71	10
G	C	B	P	42	227.5	7.63	34	1.14	0	0	0.00%	0.00	0.00	0.00	0.00%	20.71	9.10%	0.48	0.61	9
H	D	B	G	498	3854.5	8.47	80	0.18	21	4	4.76%	0.19	0.01	21.67	0.54%	530.66	13.77%	1.17	6.63	10
I	D	B	G	250	2514.3	9.75	140	0.54	31.5	19	11.95%	0.60	0.07	8.19	1.25%	163.45	6.50%	0.63	1.17	4
J	C	B	P	29	386.25	13.66	18	0.62	8.25	0	0.00%	0.00	0.00	4.64	1.58%	51.60	13.02%	1.78	2.87	N/A
K	C	B	M	87	1386.6	17.11	76	0.94	10	61	44.53%	6.10	0.75	8.10	0.72%	232.00	16.74%	2.86	3.05	8

Résultats de sondage

Du 12 au 17 juillet 2017, les 16 développeurs de l'équipe ont été invités à répondre à un sondage de satisfaction sur leur expérience. 12 ont répondu. 10 ont participé comme réviseurs, tous ont vu leur travail être revu. Quelques résultats :

Simplicité du processus de revue

Réviseurs

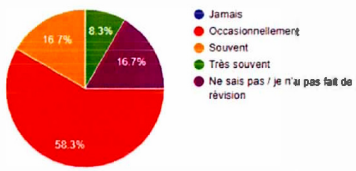
Auteurs

11

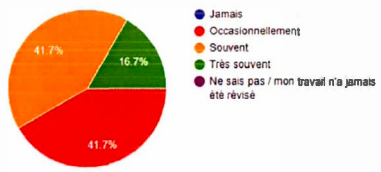
Résultats de sondage

Transfert de compétences techniques

Réviseurs



Auteurs

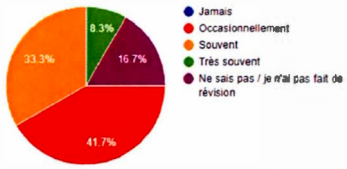


12

Résultats de sondage

Transfert de connaissances sur les applications

Réviseurs



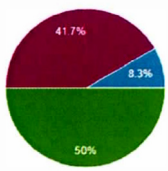
Auteurs

13

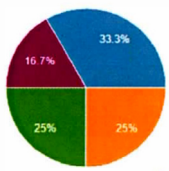
Résultats de sondage

Satisfactions avec les nouveaux outils

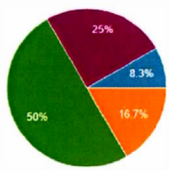
TFS 2015

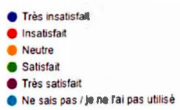


SonarQube



Liste de vérification

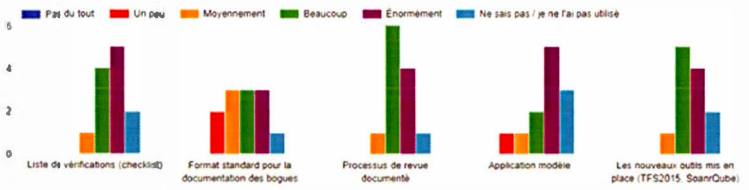




14

Résultats de sondage

Outils qui contribuent le mieux à uniformiser le développement



15

Atteinte des objectifs initiaux

Objectif	Atteint ?
Assurer que les bonnes pratiques de sécurité du code soient systématiquement appliquées et vérifiées	Partiel
Réduire l'effort de corrections de défauts trouvés par les analystes lors des tests intégrés et les clients (business et grand public)	Non prouvé
Augmenter la qualité interne du code et des documents pour réduire l'effort de maintenance	Non prouvé
Transfert de connaissances sur les applications	Oui
Partage de compétences techniques	Oui
Uniformiser les pratiques de développements (standards et bonnes pratiques)	Oui

16

Respect des éléments du projet

- Tous les éléments prévus ont été réalisés
- Le budget a été respecté

Prévu	Consommé	Restant (estimé)
210h	190h	10h

17

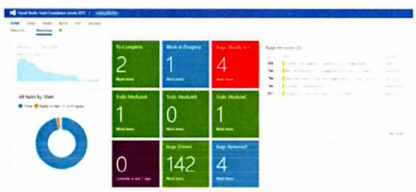
Retombées directes pour l'équipe Web-Micro

- Processus de revue documenté
- Liste de vérification précise
- Mise en place d'outils de suivi des revues (TFS2015)
- Mise en place d'outils de suivi de la qualité (SonarQube)
- Occasions de partager de l'information sur les solutions, les applications et les compétences techniques

18

Autres retombées pour l'équipe Web-Micro

- Meilleure rigueur dans la documentation des défauts
- Culture de la mesure et opérationnalisation (tableaux de bord)
- Revue du processus d'estimation de projet (COSMIC, poker planning)
- Désirs exprimés d'un processus plus rigoureux et plus contrôlé



19

Travaux futurs

- Continuer la sensibilisation sur l'importance de la rigueur dans le processus de revue et de mesure
- Formation SonarQube (demande du sondage)
- Mettre en place des revues documentées des spécifications
- Insister sur la qualité des artéfacts et ses conséquences
- Suivre la progression des mesures sur une base régulière (automatisation et tableau de bord)
- Poursuivre la révision COSMIC
- Réviser le processus annuellement
- Évaluer la possibilité d'assigner 2 développeurs sur tous les projets (rétroactions et challenges plus rapides et fréquents)

20

Confidentialité des résultats pour remise à l'UQAM

- Aucun nom de personne (sauf le mien et les noms des directeurs dans la section remerciement)
- Le Centre Rx, le Groupe Jean Coutu et l'équipe Web-Micro sont cités
- Aucun nom de projet réel (que des projets de tests dans les prises d'écran). Les mesures de projet sont réelles.
- La productivité de l'équipe (en PFC) est citée
- Aucun détail par rapport à l'infrastructure (à part les noms des outils comme TFS2015, Sharepoint, Resharper et SonarQube)

21

Confidentialité des résultats pour remise à l'UQAM

Annexes

- Plan de projet (générique)
- PowerPoint des présentations de démarrage et de résultat.
- Plan de mesure (générique)
- Document du processus de revue
- Liste de vérification :
  - Descriptions des items de revue
  - Description générale de l'architecture des applications
- Résultats du sondage (anonymes)

22

Questions

- Commentaires
- Suggestions

23

BIBLIOGRAPHIE

Bankes, K. and F. Sauer (1995). Ford systems inspection experiences. International Information Technology Quality Conference. Orlando, Florida, Quality Assurance Institute.

Beck, K. and C. Andres (2004). Extreme Programming Explained: Embrace Change (2nd Edition), Addison-Wesley Professional.

Beck, K., et al. (2001). "Manifesto for Agile Software Development." de http://agilemanifesto.org/.

Bergeron, É. (2016). Chromium, l'open-source à la sauce Google. Montréal, UQAM.

CMMI® Product Team (2010). "CMMI® for Development, Version 1.3." de http://cmmiinstitute.com/resources/cmmi-development-version-13.

Cohen, J., et al. (2013). Best Kept Secrets of Peer Code Review, SmartBear Software.

Gilb, T. and D. Graham (1993). Software Inspection, Addison-Wesley.

Google (2017). "Contributing Code." Récupéré le 30 mars, 2017, de https://www.chromium.org/developers/contributing-code.

Letouzey, J.-L. (2016). "The SQALE Method for Managing Technical Debt" de http://www.sqale.org/wp-content/uploads/2016/08/SQALE-Method-EN-V1-1.pdf

Martin, R. C. (2009). "The Boy Scout Rule." Récupéré le 1er avril, 2017, de http://programmer.97things.oreilly.com/wiki/index.php/The_Boy_Scout_Rule.

McConnell, S. (2004). Code Complete (2nd edition), Microsoft Press.

Microsoft (2017). "Team Foundation Server 2015 Update 2." Récupéré le 31 mars, 2017, de https://www.visualstudio.com/en-us/news/releasenotes/tfs2015-update2-vs.

Müller, M. M. (2005). "Two controlled experiments concerning the comparison of pair programming to peer review." The Journal of Systems and Software **78**: 166-179.

Power, K. (2013). Understanding the impact of technical debt on the capacity and velocity of teams and organizations: viewing team and organization capacity as a portfolio of real options. Proceedings of the 4th International Workshop on Managing Technical Debt. San Francisco, California, IEEE Press: 28-31.

Schwaber, K. and J. Sutherland (2016). "The Scrum Guide." Récupéré le 30 mars, 2017, de http://www.scrumguides.org/docs/scrumguide/v2016/2016-Scrum-Guide-US.pdf.

Shatwani, R. (2010). "A Quantitative Investigation of the Acceptable Risk Levels of Object-Oriented Metrics in Open-Source Systems." IEEE TRANSACTIONS ON SOFTWARE ENGINEERING **36**(2): 216-225.

SmartBear (2017). "Code Review Tool and Peer Review For Developers | SmartBear Collaborator." Récupéré le 31 mars, 2017, de https://smartbear.com/product/collaborator/overview/.

Sprunck, M. (2012, 16 octobre). "Selected Rules of Thumb in Software Engineering." Récupéré le 30 mars, 2017, de http://www.sw-engineering-candies.com/blog-1/rules-of-thumb-in-software-engineering.

Trudel, S. (2005). Méthode d'inspection, CRIM.

Votta, L. G. J. (1993). Does every inspection need a meeting? ACM SIGSOFT Symposium on Software Development Engineering. Los Angeles, ACM Press: 107-114.

Wiegers, K. E. (2002). Peer Reviews in Software, Addison-Wesley.

Wiegers, K. E. and J. Beatty (2014). Software Requirements, Microsoft Press.

Williams, L. A. and R. R. Kessler (2000). "All I Really Need to Know about Pair Programming I Learned In Kindergarten." Communications of the ACM **43**(5): 108-114.