Influence d'une phase de préparation par l'auteur sur le nombre d'itérations d'une revue de code

par

Véronique Vit

Essai présenté au CEFTI

en vue de l'obtention du grade de maître en technologies de l'information

(maîtrise en génie logiciel incluant un cheminement de type cours en technologies de l'information)

FACULTÉ DES SCIENCES

UNIVERSITÉ DE SHERBROOKE

Longueuil, Québec, Canada, 2018-12-06

Sommaire

La problématique abordée dans cet essai est liée à l'utilisation des revues de code dans un contexte de développements applicatif. Bien que plusieurs études aient été réalisés pour démontrer leur pertinence, il n'en demeure pas moins que plusieurs personnes doutent de leur efficacité due au temps additionnel requis. Même les entreprises qui utilisent les revues de code depuis plusieurs années, et qui sont convaincues de leur rentabilité, trouvent parfois que ce processus peut être long, causant des délais importants [7].

Compte tenu de cette problématique, l'objectif de cet essai est de vérifier si c'est possible de réduire le temps écoulé pour une revue de code en réduisant à un le nombre d'itérations. Dans ce contexte-ci, une itération équivaut au nombre de fois où le ou les réviseurs doivent valider la revue de code.

Pour ce faire, l'hypothèse qui a été posée est que, en ajoutant une phase de préparation, fait par l'auteur, il sera possible de réduire le nombre d'itérations à un, réduisant par le fait même la durée totale de revue nécessaire. Un projet expérimental a été créé afin de pouvoir valider l'impact de cette nouvelle phase de préparation pour ensuite en analyser les effets.

Les résultats n'ont pas permis de démontrer clairement que l'ajout d'une telle phase permettait de réduire à un le nombre d'itérations. Cela dit, une diminution de 8 % du nombre d'itérations moyennes nécessaires a été observée pour les groupes participants à l'expérimentation, versus 6 % pour les groupes non participants. Cette amélioration s'est aussi fait ressentir au niveau de la durée moyenne d'une revue de code, qui a été diminuée de 38 % (versus 30 % pour les groupes non participants).

Pour conclure, bien que l'hypothèse n'ait pu être validée positivement, l'analyse des résultats permet toutefois d'observer une amélioration. Le lien avec la phase de préparation n'a par contre pu être démontré sans equivoque. De plus amples études sont nécessaires pour, entre autres, confirmer ces résultats ainsi que déterminer son impact en matière de coûts.

Remerciements

Je tiens tout d'abord à remercier mon directeur de recherche, le professeur Pierre Martin Tardif, pour son expertise, ses conseils et son soutien. Je remercie aussi Mme Lynn Legault et M. Vincent Echelard de m'avoir guidée dans les cours INF787 et INF788, pour leurs conseils académiques. Je tiens également à remercier M. Claude Cardinal, directeur au CeFTI, sans qui cette aventure ne me serait même pas passée par la tête. Merci pour ta présence et ton encadrement tout au long de mon cheminement académique.

Un grand merci aux participants et tous ceux qui ont été impactés, de près ou de loin par la mise en place de l'expérimentation pour cet essai. Sans vous, cet essai n'aura pas été possible.

Merci à Christian pour son soutien et son accompagnement durant les nombreuses heures de cours et de travaux. T'avoir à mes côtés a rendu les travaux beaucoup plus agréables.

Finalement, je remercie aussi ma famille, la vieille et la nouvelle, pour leur soutien et leur compréhension tout au long de ce périple. Merci surtout à mon conjoint, Alexis qui a su me motiver à poursuivre malgré les rebondissements de la vie et qui a pris les règnes de la famille pendant que je me consacrais à mes études.

Table des matières

Liste des tableaux

Liste des figures

Glossaire

Auteur : développeur à l'origine de la modification pour laquelle une revue de code est

Itération : une itération représente un cycle de revue de code où l'auteur est amené à faire valider son code. Chaque revue de code a au minimum une itération. Chaque rejet de la part d'un réviseur ajoute une itération.

CSharp : un language de programmation orienté, commercialisé par Microsoft.

Moyenne : mesure statistique caractérisant les éléments d'un ensemble de quantités. Elle exprime la grandeur qu'aurait chacun des membres de l'ensemble s'ils étaient tous identiques sans changer la dimension globale de l'ensemble.

Écart type : mesure de la dispersion de données. Ce nombre indique dans quelle mesure la plus la dispersion des notes de chaque participant est

Liste des sigles, des symboles et des acronymes

TFS : Team Foundation Server. Il s'agit d'un logiciel permettant, entre autres, la gestion des fichiers sources d'un logiciel ainsi que le suivi d'éléments de travail, telles les revues de codes.

DLL : Dynamic Link Library. Il s'agit d'un bibliothèque logicielle d'un programme. Elles sont utilisées pour permettre l'interfaçage entre deux programmes.

Introduction

Depuis l'arrivée des ordinateurs personnels, au milieu des années 70, l'industrie de la conception logicielle est en pleine expansion. De nos jours, l'informatique est omniprésente dans le quotidien. Les applications logicielles ne sont plus restreintes aux ordinateurs; elles se retrouvent maintenant dans les automobiles, dans les balayeuses, les montres, les téléphones, etc. Les entreprises ont de plus en plus besoin de concepteurs de logiciels afin de répondre à la demande grandissante d'informatisation.

La conception de logiciel consiste à traduire les besoins d'un client en application informatique. La maintenance est une étape importante dans le cycle de vie d'une application; c'est souvent la plus coûteuse [1]. Celle-ci consiste à modifier le code existant soit pour y apporter une correction, une adaptation ou un perfectionnement. Il faut donc être en mesure de modifier le code existant efficacement, sans prendre trop de temps, et ainsi permettre une bonne évolution des applications. C'est pourquoi de nombreuses pratiques ont été introduites afin d'aider les concepteurs à améliorer leur code. Une de ces pratiques est la revue de code. La revue de code consiste à faire valider le code de chaque modification, fait dans la phase d'implémentation ou de maintenance, par au moins une autre personne impliquée dans le développement. Les réviseurs jugent si des ajustements sont nécessaires avant la mise en production. Cette révision de code se joint aux tests afin d'accroître la qualité du logiciel. En plus d'inspecter pour des erreurs qui auraient pu s'y glisser, on y certifie la conformité avec les demandes du client ainsi que la maintenabilité. Pour faciliter cette dernière, les concepteurs doivent bien organiser et commenter leur code. Ainsi, la compréhension du code est améliorée. Ceci facilite les modifications futures pour ce même concepteur, mais surtout pour toute autre personne qui doit y apporter des altérations.

Le standard IEEE [2] utilise le terme inspection en référence à cette pratique. Cependant, pour des raisons de clarté et de compréhension, le terme « revue de code » est utilisé tout au long de cet essai.

L’implantation d’un processus de revue de code nécessite d’ajouter des heures au projet pour la validation du code. La rentabilité des revues de code a déjà fait l’objet de plusieurs études. Les heures sauvées en correction d’anomalies et en maintenance permettent de rapidement récupérer les heures additionnelles acquises. Shell Research estime sauver en moyenne trente heures de maintenance pour chaque heure investie dans les revues [3].

Toutefois, l’efficience des revues de code doit être évaluée par chaque entreprise. Plus encore, elle devrait être jaugée ponctuellement afin de s’assurer que sa mise en place est toujours efficace. Plusieurs facteurs, tels le nombre de réviseurs ou l’ampleur des modifications, peuvent venir contribuer à l’inefficacité d’une revue de code. Les critères de rendement varient par entreprise, parfois même par groupe de travail. Certains se concentrent sur la réduction d’anomalies, peu importe le coût. D’autres sont prêts à accepter un risque plus élevé en matière d’anomalies afin de limiter la durée consacrée aux revues de code.

Cet essai vise à vérifier s’il y a un moyen qui permet de réduire le temps de révision du code à une itération.

L’essai se divise en quatre sections. La première section présente le contexte de la recherche, accompagné d’un résumé de différents écrits sur le sujet, suivi de la présentation de la problématique concernée dans cette étude. La seconde section expose la méthodologie utilisée ainsi que les résultats obtenus. Puis, la troisième section décortique les résultats. Finalement, la quatrième section présente la conclusion à l’essai.

Chapitre 1

Mise en contexte

La section précédente décrit sommairement ce qu'est une revue de code. Dans celle-ci, différents concepts sont élaborés afin de faciliter la compréhension de la problématique présentée dans cet essai.

1.1 Phases de conception

Bien qu'il existe plusieurs méthodes de conception de logiciel, elles contiennent typiquement les phases suivantes, même si l'ordre varie d'une méthode à l'autre :

- Collection et analyse des besoins : il s'agit de bien répertorier les besoins du client.
- Conception : à cette étape, les solutions qui permettront d'implémenter les différentes fonctionnalités sont déterminées.
- Implémentation et Codage : il s'agit de l'activité principale. À cette étape, le développement c'est-à-dire la programmation des différentes fonctionnalités est effectuée.
- Contrôle : il s'agit de vérifier les différentes fonctionnalités pour s'assurer qu'elles fonctionnent bien, qu'elles donnent de bons résultats et répondent aux besoins.
- Déploiement : il s'agit de l'installation chez le client.
- Maintenance : cette étape commence une fois le logiciel installé et accepté par le client. Il s'agit de corriger les erreurs qui auraient pu se glisser en production, après le déploiement. Cette étape vient généralement avec un contrat d'une durée prédéterminée.

Les revues de code sont utilisées dans la phase d'implémentation et codage. On la retrouve parfois aussi dans la phase de maintenance afin de valider les corrections apportées.

1.2 Introduction des revues de code

Michael Fagan présente, en 1976, la première version des revues de code, aussi nommée « inspection » [4]. Il y présente une approche itérative divisée en cinq étapes, chacune effectuée pour les phases de conception et d'implémentation :

1. Exposé général (en équipe) : l'auteur décrit les modifications, le contexte général ainsi que le détail de changements spécifiques (logique, dépendances, etc.).
2. Préparation (individuelle) : les participants étudient la documentation fournie à l'étape précédente. Le but est de comprendre les modifications, autant du point de vue de la conception que de la logique.
3. Inspection (en équipe) : une fois la conception et la logique comprises par tous les membres de l'équipe, chaque modification est revue afin de trouver des erreurs. Environ 24 heures après l'inspection, le modérateur, nommé lors de la rencontre, produit un rapport afin de s'assurer que tous les problèmes seront corrigés.
4. Réfection (*rework*) : l'auteur corrige les problèmes notés à l'étape précédente.
5. Suivi (*Follow-up*) : le modérateur valide que les problèmes ont été corrigés. Si plus de cinq problèmes ont été recensés, une nouvelle inspection est suggérée.

L'élaboration de ce processus nécessite trois types de participants, soit :

- Auteur : développeur qui écrit le code à valider;
- Modérateur : gestionnaire du processus de revue de code. Il est chargé de guider les réunions et faire le suivi;
- Lecteurs : membres de l'équipent qui sont amenés à réviser le code. Aussi nommés réviseurs.

Dans une de ses études [22], M. Fagan propose de limiter la durée de chaque réunion à deux heures afin d'optimiser l'efficacité. De plus, des équipes de quatre participants sont suggérées; bien qu'il puisse être nécessaire d'augmenter dans certains cas.

Cette inspection vise à réduire les coûts liés aux réfections dues aux erreurs découvertes après la phase de conception et après celle d'implémentation. Rappelons que plus une erreur est découverte tard dans le processus, plus elle est coûteuse à rectifier [5].

1.3 Processus des revues de code

La Figure 1-1 présente les principales étapes d'une revue de code. Une fois le code prêt à être révisé, l'auteur de la modification, c'est-à-dire le développeur, sélectionne un ou plusieurs réviseurs à qui il envoie une demande de revue de code. Chaque réviseur examine le code et y appose des commentaires. Les réviseurs décident ensuite s'ils acceptent, ou non, la mise en production du code.

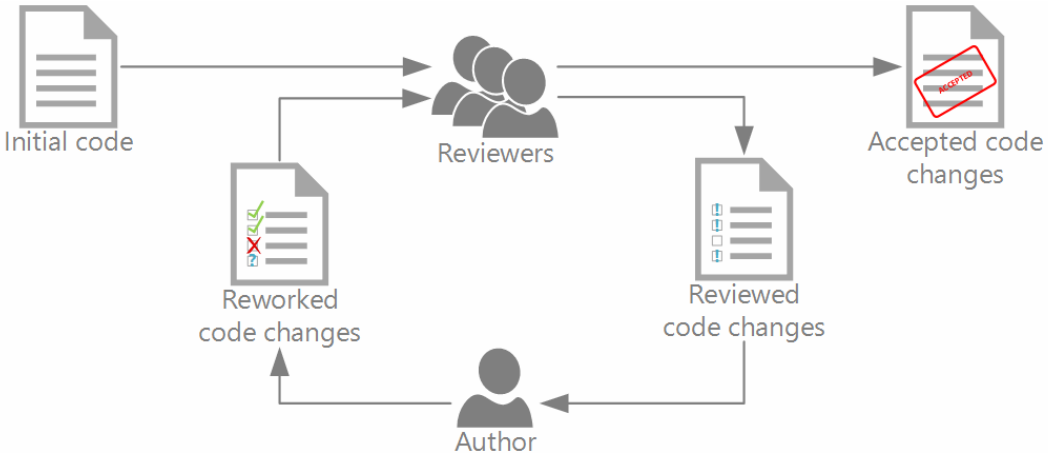


Figure 1-1 Principales étapes d'une revue de code

Source : Manekovskiy, A. (2015), Why Your Team Should Do a Code Review on a Regular Basis, [En ligne], http ://amanek.com/

Il existe un grand éventail de types d'erreurs et de gravité. Selon les erreurs, les réviseurs apposent un status à la revue de code. Les statuts possibles sont « approuvée » (sans commentaires), « approuvée avec commentaires » ou « refusée ».

Le statut « approuvée avec commentaires » permet au réviseur d'indiquer au développeur qu'il a apposé des commentaires mais qu'il juge ceux-ci mineurs. Dans un tel cas, le développeur est responsable de faire les adaptations mais n'aura pas à resoumettre la modification pour une révision additionnelle.

Lorsqu'une revue de code est refusée, l'auteur doit appliquer les modifications demandées par les réviseurs et resoumettre les nouvelles modifications pour une deuxième itération de revue de code. Il n'y a pas de règles dictant quand une revue de code doit être approuvée avec commentaire versus refusée. Chaque réviseur fait le choix en fonction de ses propres critères d'évaluation.

Dans un monde idéal, le processus de revue de code ne consisterait qu'en une seule itération. Il est toutefois utopique de penser que les commentaires des réviseurs soient toujours mineurs; surtout dans le cas d'un développeur débutant.

1.4 Évolutions des revues de code

Dès le milieu des années 1980, de nombreuses variations du processus de Fagan voient le jour. Certaines cherchent à réduire les coûts alors que d'autres veulent augmenter la quantité de défauts trouvés. On retrouve cinq grandes catégories de revues de code [6].

La première, la revue formelle, est celle telle que présentée par Fagan. Elle se distingue par sa rigueur; c'est un processus qui doit être suivi à la lettre, où chaque étape est bien définie.

La deuxième demande à l'auteur de présenter sa modification à un autre développeur, ce dernier étant à ses côtés. Les deux participants étant ainsi réunis peuvent facilement débattre et même corriger le ou les problèmes ensemble. Des outils permettent d'implanter cette catégorie de revues de code malgré la distance qui sépare l'auteur du réviseur.

La troisième consiste à utiliser les courriels comme élément de communication. L'auteur envoie un courriel avec les fichiers affectés par ses modifications aux différents réviseurs. Après les avoir examinés, les questions et discussions des réviseurs se font par l'entremise de courriels.

La quatrième catégorie de revue de code est celle où des outils sont utilisés. Ces outils servent à aider les participants lors du processus de la revue de code. Il peut s'agir, par exemple, d'assembler automatiquement les fichiers modifiés ou afficher les modifications en tant que différences (avant et après).

Finalement, la cinquième catégorie de revue de code est une méthode de développement. Il s'agit de la programmation par paires. Cette méthode demande que deux développeurs travaillent ensemble, à un seul poste de travail. Pendant qu'un écrit le code, l'autre le valide au fur et à mesure. On retrouve donc, un auteur et un réviseur à chaque instant du développement.

Malgré le grand nombre d'évolutions, certains concepts des revues de code demeurent inchangés. On y retrouve toujours les mêmes éléments, le code source et des commentaires. Ces commentaires peuvent être autant des questionnements, des informations additionnelles à communiquer à l'autre ou des erreurs identifiées.

1.5 Motivations des revues de code

Lors de la présentation du processus, Fagan met l'emphase sur la détection d'erreur. C'est la mesure la plus facile à évaluer. Mais, il stipule tout de même qu'un des bénéfices les plus importants d'une telle revue de code est l'amélioration des développeurs suite à une rétroaction. En effet, chaque commentaire soulevé sur son code permet au développeur de s'améliorer. Chaque revue de code lui permet de mieux cerner les erreurs types qu'il a l'habitude de commettre.

Bacchelli et Bird [7] se sont penchés, en 2013, sur les motivations derrière l'utilisation des revues de code. Leurs résultats confirment que, bien que les revues de code aient évolué, la motivation principale reste inchangée. En effet, 44 % des développeurs justifient l'utilisation de revues de code pour trouver des défauts. Il dénote tout de même plusieurs autres facteurs d'importance, autant chez les développeurs, les gestionnaires que les testeurs. La Figure 1-2 présente le résultat de leur sondage auprès de 873 développeurs. L'amélioration du code, c'est-à-dire la lisibilité, les commentaires, la cohérence, émerge comme deuxième motivation la plus importante avec 39 % des votes.

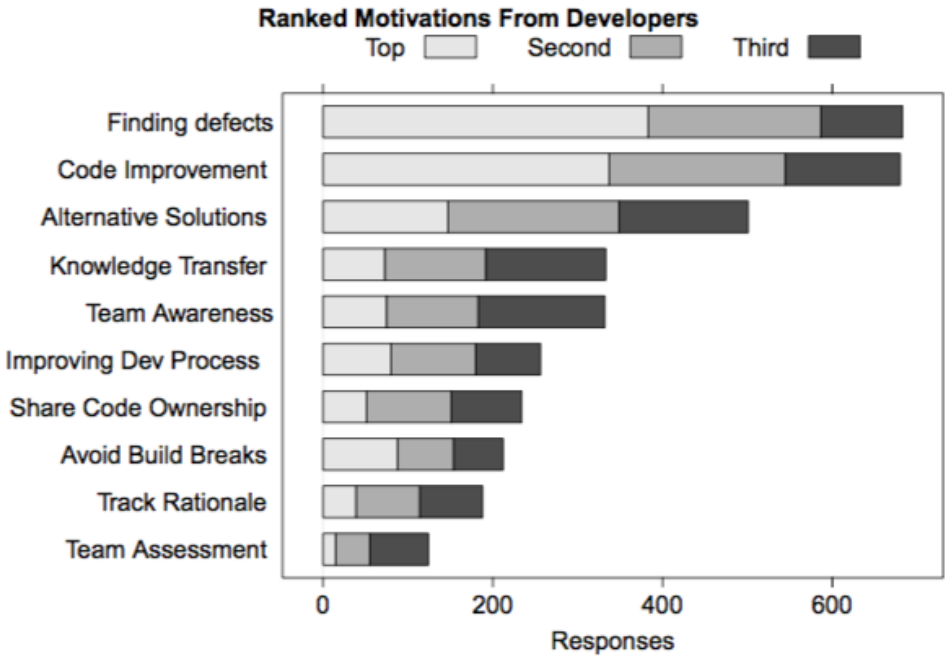


Figure 1-2 Motivations des développeurs pour l'utilisation de revues de code

Source : Bacchelli, A. et Bird, C. (2013), p. 715

1.6 Mesures d'efficacité des revues de code

Suite à la présentation du processus de Fagan, de multiples études [3], [7], [8] ont analysé son efficacité. Pour ce faire, plusieurs mesures ont été recueillies et observées. Les plus populaires sont :

- le nombre de lignes de code source modifié;
- la durée du processus de revue de code;
- le nombre de défauts trouvés.

Ces mesures permettent d'obtenir des mesures d'analyses plus complètes telles :

- la vitesse d'inspection : nombre de lignes de code modifiées par nombre d'heures de revue de code;

- le ratio de défauts : nombre de défauts trouvés par nombre d'heures de revue;
- la densité des défauts : nombre de défauts trouvés pour 1000 lignes de code modifiées.

Ces mesures ont permis, à de nombreuses reprises, de démontrer l'efficacité des revues de code. Malgré tout, plusieurs critiquent encore leur pertinence. Certains estiment qu'elles coûtent plus qu'elles ne valent [9].

1.7 Constats

Lors des entrevues menées par Bacchelli et Bird [7], les réviseurs mentionnent que le plus gros défi auquel ils font face est la compréhension. Leur analyse des commentaires laissés dans 570 revues de code confirme leurs propos. Après l'amélioration du code, qui représente 29 % des commentaires, la compréhension est la deuxième catégorie de commentaire la plus recensée. Les développeurs sondés dans cette étude mentionnent qu'ils sont en mesure de répondre mieux et plus rapidement lorsque l'auteur fournit le contexte et des indications dans la revue de code. Dans une autre étude, Bird et ses collègues [10] viennent réitérer le besoin de compréhension. Ils proposent des améliorations, présentées à la Figure 1-3, à apporter au processus de revues de code, autant pour les auteurs que pour les réviseurs. Plusieurs de ces recommandations visent la préparation de la revue de code par l'auteur. Une des suggestions faites aux auteurs est de viser des changements incrémentaux, le plus petit possible. De plus, ils conseillent de documenter les changements en décrivant l'approche utilisée ainsi que la raison de la modification. L'auteur peut même aider le réviseur en lui indiquant une approche à prendre pour la revue de code. Ceci est particulièrement utile dans du code orienté-objet. Finalement, ils rappellent que les auteurs devraient toujours s'assurer de tester leurs modifications.

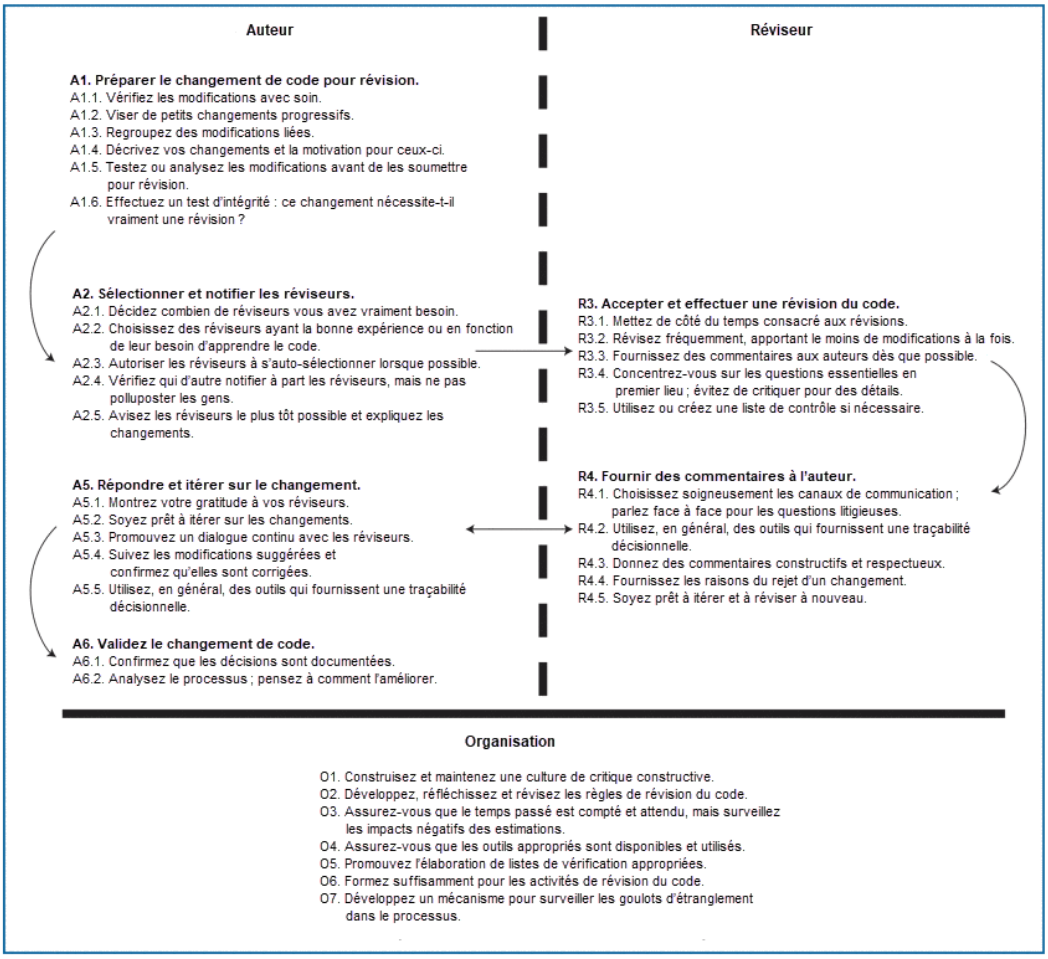


Figure 1-3 Meilleures pratiques suggérées par l'étude chez Microsoft

Traduction libre

Source : MacLeod, L., Greiler, M., Storey, M.A., Bird, C. et Czerwonka, J. (2018), p. 38

Les recommandations suggérées dans ces deux études sont basées sur des analyses qualitatives. Des études quantitatives sont nécessaires pour confirmer les bienfaits de ces recommandations.

Le prochain chapitre présente plusieurs ouvrages scientifiques utiles à la compréhension de la problématique et de l'hypothèse posées.

Chapitre 2

Revue de la littérature

La section précédente présente les différents éléments nécessaires à la compréhension des écrits scientifiques. Celle-ci résume des recherches passées et identifie les trous que tente de remplir cette essai.

2.1 Méthodologie de recherche

Des articles, livres et revues de colloques ont été trouvés, pour la majorité, à l'aide de l'outil de recherche bibliothécaire de l'Université de Sherbrooke. Les mots-clés « *code review* » et « *inspection* » ont permis de trouver la plupart des articles sur les revues de code. Une recherche avec le mot « *checklist* » a ensuite permis de trouver des articles traitant de l'utilisation et la création d'une liste de contrôle dans le contexte des revues de code. Dans tous les cas, les résultats ont été filtrés pour ne contenir que ceux ayant « *computer* programming » ou « computer software » dans le sujet. Les banques de données « Engineeringvillage », « IEEE » et « ACM Digital Library » ont aussi été utilisées avec les mêmes mot-clés.

2.2 Taille des modifications

Une etude faite à l'université de Trier, en Allemagne, s'est penché sur le ratio de soumissions de modifications par rapport au nombre accepté dans un logiciel libre de licence (*open source*) [11]. Les chercheurs y étudient le pourcentage de modifications acceptées ainsi que le temps nécessaire à l'acceptation de celles-ci. De plus, ils s'interrogent à savoir si les petites modifications (moins de 4 lignes de code modifiées) sont acceptées plus facilement que les grosses (plus de 15 lignes de code modifiées). Pour les deux applications étudiées, le taux d'acceptation d'une modification est d'environ 40 %. Ils évaluent ensuite le taux

d’acceptation en relation avec la taille des modifications. Ils en concluent que, en général, les petites modifications ont plus de chances d’être acceptées.

En ce qui a trait au temps nécessaire pour la mise en production d’une modification, environ un quart des modifications ont pris entre 24 et 48 heures. La moitié moins d’une semaine et environ un tiers a pris plus de deux semaines. Ils n’ont toutefois pas été en mesure de déterminer si l’ampleur d’une modification a un impact sur le temps nécessaire pour l’approbation.

Cette étude est à prendre avec un bémol, car il s’agit d’une étude de cas que sur deux applications. Il faudrait que l’expérience soit reconduite sur un plus gros échantillon. De plus, l’analyse des données fournies ne permet pas clairement de comprendre leurs conclusions. Cela dit, les résultats sont intéressants et laissent croire qu’une plus petite modification a plus de chance d’être acceptée. Selon ces résultats, la réduction de la taille des modifications devrait augmenter le nombre d’acceptations de modifications, réduisant ainsi le nombre d’itérations.

Bien que les recherches précédentes aient conclu que les petites modifications sont plus enclines à être approuvés rapidement, l’étude [12] faite par des chercheurs de l’université de Waterloo, au Canada, n’arrive pas tout à fait à la même conclusion. En effet, l’examen statistique de la grosseur des revues de code en rapport avec le temps nécessaire pour les traiter ne donne qu’une très faible relation. Ce n’est qu’en subdivisant les revues de code en groupes qu’ils arrivent à voir un impact, sur le temps et le nombre d’itérations, plus tangible.

Ils ont séparé plus de 10 000 revues de code en quatre groupes égaux, selon la taille des modifications. Le groupe A représente les plus petites demandes de revue, avec en moyenne quatre lignes de code modifiées. Pour celles-ci, une durée moyenne de 440 minutes, avec une médiane de 39 minutes, a été observée pour traitement des revues de code. Le groupe B, des modifications d’environ 17 lignes de code, a nécessité en moyenne 531 minutes avec une médiane de 46 minutes. Le groupe C, avec une moyenne de 54 lignes modifiées, a pris 542 minutes en moyenne avec une médiane de 48 minutes. Et finalement, le groupe D, les

modifications d'environ 432 lignes de code, ont pris 545 minutes, avec une médiane de 64 minutes.

En plus d'impacter la durée de révision, ils ont été en mesure de démontrer que la taille d'une modification a un impact significatif sur le nombre d'itérations de la revue de code. Pour les groupes A et B, la médiane est d'une seule itération alors qu'elle est de deux pour le groupe C et de trois pour le D. Cette dernière statistique est en lien direct avec la question de cette essai. Elle renforce l'hypothèse que la décomposition d'une modification, comme proposée dans la phase de préparation, a un impact sur le nombre d'itérations de la revue de code.

Finalement, bien que les études quantitatives n'ont pu démontrer hors de tout doute l'impact de la taille des modifications sur le processus de revue de code, une étude qualitative récente [10] met en évidence cet aspect comme étant un des défis les plus importants auquel font face les réviseurs. Ceci confirme, à nouveau, l'importance de considérer cet aspect lors de la soumission de revues de code.

2.3 Compréhension de la modification

Une des principales problématiques lors des revues de code est liée à la compréhension. Il est bien difficile, pour les réviseurs, de donner des commentaires pertinents si le but de la modification, ou la modification elle-même, n'est pas compris. Qui plus est, il est souvent nécessaire d'en comprendre le contexte. Une correction peut être bonne pour un problème particulier, mais causer à son tour plusieurs problèmes. C'est pourquoi une bonne compréhension générale est nécessaire pour être un bon réviseur. Une étude publiée en 2017 examine les différents commentaires laissés par les réviseurs afin d'identifier ceux qui indiquent une confusion [13]. Leur but était de comprendre comment les réviseurs expriment leur confusion. Une analyse manuelle de 800 commentaires a permis d'établir qu'entre 50 % et 60 % des commentaires indiquent une confusion de la part du réviseur. Plus précisément, 50 % pour les commentaires d'ordre général et 60 % pour les commentaires spécifiques à un fichier ou une ligne de code. Les résultats de cette étude mettent en évidence le manque de

compréhension des réviseurs. L'ajout de commentaires explicatifs, par l'auteur, ne peut qu'aider à diminuer ce pourcentage.

En 2016, Kononenko et al [21] se sont penché sur un tout autre aspect des revues de code. Ils ont tenté de comprendre la perception qu'ont les développeurs des revues de code. 88 développeurs ont répondu à leur sondage de 19 questions. L'analyse de ces réponses a, entre autres permis de, reconfirmer l'impact de la taille des modifications sur la durée nécessaire pour la revue de code. La clarté des explications fournies par l'auteur a été mentionnée par plusieurs répondants comme étant un élément important autant sur la durée nécessaire pour la revue de code que sur la décision d'accepter ou non les modifications. Pour être efficace, les réviseurs doivent comprendre le problème, le raisonnement derrière la modification, comment cela corrige le problème ainsi que, pourquoi l'auteur a choisi une façon de faire plutôt qu'une autre.

Plus l'auteur fournit des explications aux réviseurs, moins ce dernier a d'effort, et donc de temps, à consacrer pour trouver les réponses à ses questions. De plus, la présence de ces clarifications confirme, ou infirme, rapidement au réviseur que l'auteur a bel et bien compris le problème. Cette étude vient renforcer l'hypothèse que l'ajout de commentaires, par l'auteur, viendra améliorer le processus des revues de code.

2.4 Types de défauts

La plupart des études sur les revues de code sont axées autour de la diminution des défauts. Très peu ont pris le temps d'évaluer les types de défauts trouvés par celles-ci. C'est ce qu'ont fait deux membres de l'IEEE [14]. Leur hypothèse est qu'il est inutile d'observer la quantité de défauts si ceux-ci ne sont pas classés. En effet, une revue de code avec 150 défauts de standard et d'orthographe n'est pas comparable avec une dans laquelle on retrouve deux défauts fonctionnels. Cette analyse leur a permis d'identifier que plus de 70 % des défauts sont d'évolutivité (documentation, lisibilité, structure) alors que seulement 20 % sont des défauts fonctionnels, laissant 10 % de faux négatifs. Bien que les défauts d'évolutivité sont importants pour la maintenance et l'évolution de l'application, ceux-ci n'ont pas d'impacts fonctionnels pour les clients. La Figure 2-1 présente les différentes catégories de défauts tels

que proposés dans l'étude. Notons que cette classification n'est pas universelle, chaque étude choisit une classification qui lui convient.

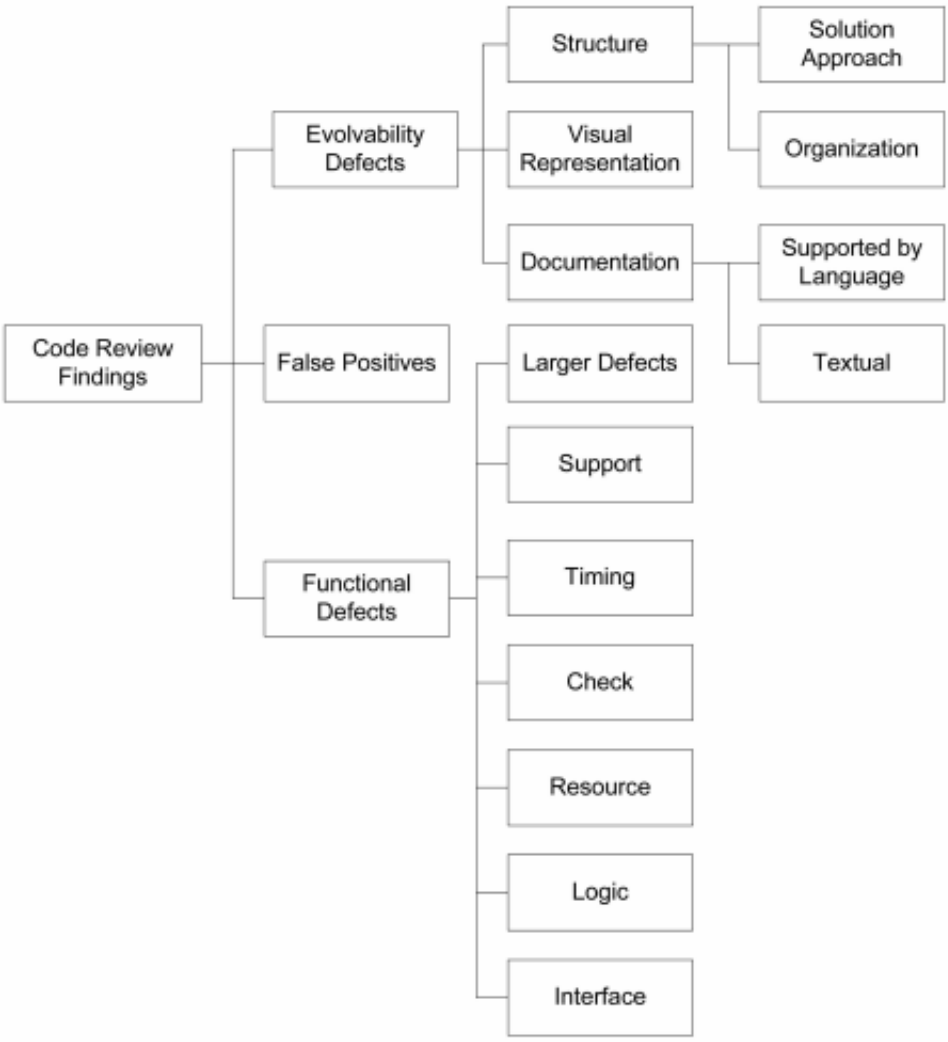


Figure 2-1 Classification des défauts de revues de code

Source : Mäntylä, M. et Lassenius, C. (2009), p. 436

Le fait de connaître les types de défauts généralement trouvés dans les revues de code est essentiel à l'élaboration d'une bonne liste de contrôle.

Dans le même ordre d'idées, l'étude de Bosu A. et al [15] tente de déterminer les caractéristiques indiquant que le code est de faible qualité. Le résultat de leur étude qualitative fait ressortir trois grandes causes de problèmes dans les revues de code, c'est à dire :

1. mauvaise lisibilité;
2. complexité inutile;
3. non-maintien de l'intégrité de l'application.

L'absence de commentaire s'est retrouvée très bas dans la liste soumise par les sondés. Ceci laisse croire que les réviseurs s'attendent à du code lisible et facile à comprendre. Du code complexe, même si bien commenté, demandera plus d'effort de compréhension et donc, prendra plus de temps à réviser. Encore une fois, ces caractéristiques serviront lors de la définition des éléments de la liste de contrôle.

2.5 Annotations de l'auteur

En 2006, une étude a été faite chez Cisco Systems inc. [6]. Au total, 2500 revues de code faites par 50 développeurs ont été analysées. Plusieurs mesures ont été analysées et comparées. Un des aspects examinés est l'impact qu'a la phase de préparation de l'auteur sur la densité des défauts trouvés par la revue de code. Dans ce contexte, la phase de préparation consiste à demander à l'auteur d'annoter son code source avant d'envoyer la demande de revue de code.

L'hypothèse était que le fait de demander à l'auteur de se contre-vérifier permettrait de diminuer le temps passé à faire la revue de code sans impacter la qualité du code. Ils ont donc demandé aux auteurs d'annoter leurs modifications afin d'expliquer certains choix et guider le réviseur. Le fait de commenter ses adaptations force l'auteur à repenser à ses changements. La théorie est qu'en faisant ceci, l'auteur découvrira, de lui-même, quelques défauts. Ceux-ci n'auront donc pas à être traités par les réviseurs.

Au final, ils ont été en mesure de confirmer que la préparation de l'auteur a bel et bien un effet sur la densité des défauts. Contrairement aux cas sans préparation, la densité des défauts était plus stable. Plus encore, la densité n'était jamais de plus de trente défauts par mille lignes de code modifiées, avec une grande majorité sans aucun défaut. Bien que ces résultats soient concluants, les auteurs y apportent tout de même un bémol. En effet, ce changement de densité peut aussi être expliqué par le fait que les commentaires de l'auteur ont miné la capacité des réviseurs à critiquer le code; occultant ainsi certains défauts.

Les auteurs de cette recherche suggèrent de pousser les recherches plus loin quant à l'impact de la préparation de l'auteur sur la quantité de défauts et le temps nécessaire pour les trouver. Ils proposent aussi des recherches futures pour analyser l'effet d'une liste de contrôle, par auteur, sur la densité des défauts. Bien que ce ne soit pas exactement ce qui est étudié dans cet essai, ça s'en rapproche beaucoup. Leurs résultats sont utiles pour la validation de l'hypothèse.

2.6 Liste de contrôle

Dans leur étude, Jenkins et Ademoye [16] étudient l'ajout des revues de code, à l'aide d'une liste de contrôle, dans un cours d'introduction à la programmation au baccalauréat. Étalés sur deux ans, ils ont comparé deux groupes, un avec les revues de code et l'autre sans. Ayant fait leurs recherches avant le début de l'expérimentation, Jenkins et Ademoye étaient déjà au fait avec les difficultés d'utiliser des revues de code dans un contexte académique. En effet, plusieurs facteurs environnementaux, tels la personnalité ou la force de chaque étudiant, viennent fausser leur efficacité. Un étudiant plus faible est difficilement en position de commenter le code d'un étudiant plus fort. C'est pourquoi ils ont opté pour une approche non traditionnelle; c'est-à-dire une revue code individuelle. Chaque étudiant devait, à l'aide d'une liste de contrôle préétablie, valider son propre code. Les résultats démontrent que cet ajout n'a pas eu d'impact significatif sur les résultats moyens des étudiants. Toutefois, une grande majorité des participants (80 %) ont mentionné, à l'aide de sondages, qu'ils croient que cette pratique a amélioré leur code. Cette étude est très pertinente pour l'essai en cours puisqu'il s'agit d'une des rares publications qui étudie l'impact d'une liste de contrôle pour l'auteur, soit

avant l'envoi à des réviseurs. Il faut toutefois nuancer ces résultats dans le contexte courant car, dans leur cas, un délai fixe de livraison leur est donné. Il est probable que certains participants aient omis la phase de la revue de code pour plutôt se consacrer à terminer le travail dans les délais. En entreprise, les développeurs sont bel et bien soumis à des délais de livraison, toutefois, le processus de revue de code n'est pas escamoté pour permettre de compléter le développement.

En 2008, Hatton [17] se penche sur l'impact de l'utilisation d'une liste de contrôle par le réviseur. Il tente de démontrer qu'une liste aide le réviseur à trouver plus de défauts. Pour ce faire, Hatton fournit aux participants un programme dans lequel il connaît déjà le nombre de défauts. Il bâtit ensuite une liste de contrôle en s'assurant que chaque défaut du programme soit couvert par au moins un élément de la liste. Une fois la liste prête, elle est distribuée et expliquée aux participants.

Dans la première phase de son étude, Hatton présente et explique sa liste de contrôle à 22 participants. Il leur laisse le choix d'utiliser, ou non, la liste de contrôle lors de la révision des modifications. L'auteur prétend que cette façon de faire représente le mieux la réalité de l'industrie; chaque employé choisit sa façon de travailler.

Dans une deuxième phase, il sélectionne 238 nouveaux participants, qu'il divise en deux groupes. Seul un des deux groupes reçoit la liste de contrôle. L'expérimentation est contrôlée de façon à s'assurer que les deux groupes n'ont pas d'interactions en eux; évitant ainsi qu'un participant du groupe sans liste de contrôle mette la main sur une liste de l'autre groupe. Dans les deux cas, l'auteur a été dans l'incapacité de démontrer une relation ou absence de relation entre l'utilisation d'une liste de contrôle et le nombre de défauts trouvés.

Ces résultats ne permettent toutefois pas non plus de conclure que l'utilisation des listes de contrôle n'est pas pertinente dans un processus de revue de code. Une autre étude de 2008 confirme que son utilisation améliore la compréhension [18] [19]. Elle est particulièrement utile pour l'intégration de développeurs novice à un projet existant.

Dans cette expérimentation, les participants, 18 étudiants au baccalauréat en technologie de l'information, avaient 30 minutes pour faire une revue de code sur une classe de 169 lignes

de code, contenant 13 erreurs. Une fois l'inspection terminée, on leur a demandé d'ajouter une nouvelle modification. La classe sans erreur leur a été fournie. Encore une fois, un délai de 30 minutes leur a été imposé. Dans cette étude, la taxonomie de Bloom [20] est utilisée pour démontrer l'amélioration des connaissances et la compréhension des étudiants.

Cette étude est intéressante, car elle démontre qu'une technique, ici la revue par liste de contrôle, peut avoir des bénéfices bien au-delà des mesures habituellement analysées. Il est par contre important de tenir compte de la taille restreinte de l'étude. Des études plus poussées sont nécessaires pour bien confirmer ces résultats.

Cette section résume différentes études concernant les défis auxquels font face les réviseurs ainsi que des pistes de solutions pour leur faciliter la tâche. Elle présente aussi des études qui évaluent l'utilisation de listes de contrôle pour les réviseurs. La présente étude propose d'appliquer certaines recommandations trouvées dans la littérature à l'aide d'une liste de contrôle à utiliser par l'auteur d'une modification.

La prochaine section présente le détail de la problématique ainsi que l'hypothèse posée.

Chapitre 3

Problématique

Depuis l'introduction des revues de code, en 1976, plusieurs variantes ont vu le jour. Malgré les différentes adaptations, le noyau reste le même, il s'agit d'un processus permettant à l'auteur d'une modification de faire valider celle-ci par un ou plusieurs réviseurs. La motivation principale qui stimule l'utilisation de revues de code reste la même; la recherche de défauts. En effet, il a été prouvé à maintes reprises que plus un défaut est trouvé tôt dans le processus de développement, moins il est coûteux de le corriger. La revue de code étant au tout début du processus, tout défaut trouvé dans celle-ci permet de sauver du temps et de l'argent. En plus de trouver des défauts, la revue de code s'est vue bénéfique pour la formation. En plus d'assurer un transfert de connaissance sur cette partie du code, la revue de code permet à divers développeurs d'examiner la solution présentée, favorisant ainsi l'amélioration de la solution et son implémentation.

La théorie qui définit le processus semble simple, l'auteur envoie ses modifications à des réviseurs, ceux-ci les annotent et indiquent s'ils approuvent ou non la modification dans son état actuel. Il n'est toutefois pas aussi simple à mettre en œuvre. Pour qu'une revue de code soit efficace, il faut que les réviseurs prennent le temps de bien comprendre ce qui a été fait, et surtout pourquoi. Sans une bonne compréhension, les commentaires des réviseurs perdent toute leur pertinence. Pour faciliter une bonne compréhension, plusieurs études proposent à l'auteur de la revue de code d'annoter cette dernière. Ces explications fournissent aux réviseurs un point de départ important à leur compréhension de la modification. Il est aussi proposé de réduire la taille des modifications le plus possible. Naturellement, plus une modification est grosse, plus il est difficile pour le réviseur de rester concentrer et à bien tout comprendre. Certaines variations des revues de code, utilisent des listes de contrôle pour aider le réviseur à identifier les défauts potentiels. Malheureusement les études sur l'efficacité de ces listes se sont avérées peu concluantes. Il ne faut toutefois pas prendre ces résultats comme un indicatif que ces listes soient inutiles [23]. La

composition de la liste est un facteur clé de succès, il faut savoir bien construire la liste en fonction des besoins, il n'existe aucune liste qui peut répondre aux exigences de tous les projets, entreprises, réviseurs.

La Figure 1-3 présente une liste de 35 meilleures pratiques pour améliorer le processus de revue de code découlant d'une étude menée chez Microsoft [10]. Ces meilleures pratiques sont proposées à la suite d'une étude qualitative faite à l'aide d'un sondage répondu par 911 développeurs et de 18 entrevues semi-structurées. Bien que cette étude soit très intéressante et d'actualité, rien ne prouve que l'application de ces pratiques ait bel et bien des effets positifs. Une étude quantitative est nécessaire pour le démontrer. Ces meilleurs pratiques sont donc que des suggestions. Une entreprise qui cherche à améliorer un aspect de ses revues de code, que ce soit la quantité de défauts trouvés, le délai accordé par revue de code, a intérêt à s'interroger sur la pertinence de la mise en place d'une ou plusieurs de ces pratiques.

Il a été prouvé que les revues de code permettent de sauver du temps et de l'argent sur la correction de défauts en maintenance. D'une part, en déployant un produit qui contient des défauts, l'image de la compagnie est affectée. D'autre part, à une époque où la technologie change rapidement, le temps supplémentaire que prennent les revues de code n'est pas à négliger. Il faut livrer de la valeur rapidement. Même si les revues de code permettaient de trouver cent pour cent des défauts, si le logiciel est désuet au moment de sa livraison, le gain de qualité est inutile. Il faut donc trouver des améliorations simples permettant au processus de revue de code d'être tout aussi efficace, mais en prenant le moins de temps possible.

Les meilleures pratiques de revue de code suggérées proposent quatre étapes pour l'auteur du code, c'est-à-dire le développeur, et deux étapes pour le réviseur. L'auteur doit préparer la revue de code, choisir et aviser le réviseur, répondre et procéder aux itérations et valider le changement de code. Le réviseur doit accepter et procéder à la revue de code et fournir les éléments à reprendre dans le code.

Puisque les revues de code apportent des avantages certains au niveau de la qualité d'un produit livré mais qu'ils allongent le délai de livraison, les entreprises ont tout intérêt à trouver un moyen de réduire la durée du processus de revue de code. Cet essai vise à vérifier si

l'ajout d'une phase de préparation, fait par l'auteur de la modification, pourrait être bénéfique sur la durée totale nécessaire pour l'acceptation d'une revue de code. Rappelons que chaque fois qu'une revue de code est rejetée, une nouvelle itération est nécessaire, ce qui entraîne une augmentation de la durée de la revue de code.

3.1 Objectifs et hypothèses

L'objectif est de vérifier si l'ajout d'une phase de préparation, fait par l'auteur, permet de réduire le nombre d'itérations nécessaire à l'acceptation de la revue de code afin de réduire la durée totale nécessaire entre la soumission et l'acceptation d'une revue de code.

Plus précisément, la phase de préparation consiste à demander à l'auteur d'une modification de faire valider sa revue de code en fonction d'une liste de contrôle préétablie.

La question que traite l'essai est la suivante : « Est-ce que l'ajout d'une phase de préparation, effectuée par l'auteur, permet de limiter à un, le nombre d'itérations afin de réduire le temps global de développement? ».

La recherche consiste à vérifier l'impact de l'ajout d'une phase de préparation sur le nombre d'itérations nécessaire pour l'approbation d'une revue de code, comme indiqué dans la Figure 3-1. Le nombre de fois que le développeur doit reprendre le travail dans le code est comptabilisé avec et sans phase de préparation. Le temps moyen nécessaire pour l'acceptation d'une revue de code est aussi mesuré avec et sans la phase de préparation.

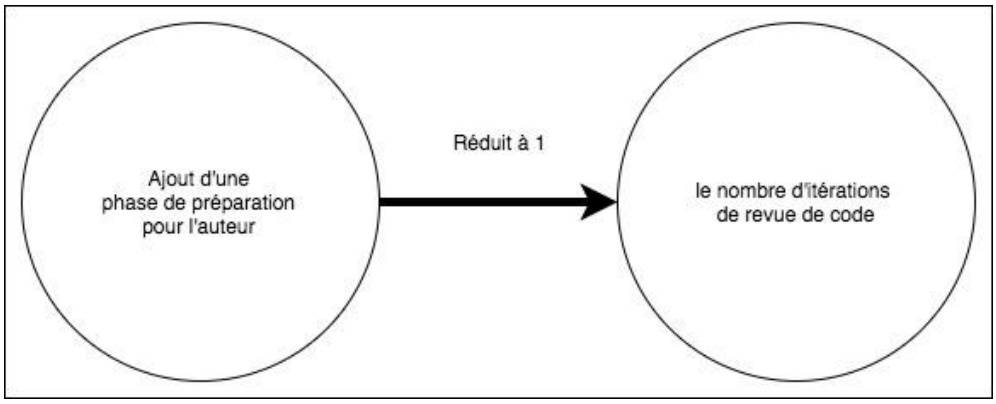


Figure 3-1 Schéma conceptuel de recherche

L'idéale serait de n'avoir aucune itération mais dans les faits, la complexité du travail du développeur peut nécessiter le regard d'une seconde personne. C'est d'ailleurs pour cette raison que les revues de code ont vu le jour. Une itération est acceptable, il n'y a par contre pas de raison que l'ensemble des éléments à corriger nécessite plus d'une itération. En ajoutant une phase de préparation, le développeur a une liste pour le guider. L'itération sert alors à trouver des défauts que l'auteur n'a pu déceler.

L'hypothèse suppose que oui, l'ajout de cette phase de préparation permet de réduire le nombre d'itérations et donc, la durée totale de revue de code.

3.2 Limites

Cet essai se limite à l'évaluation de l'impact dans un contexte de processus de revue de code sans réunions, utilisant un outil de gestion de code source. Cet outil, TFS, permet l'automatisation et le suivi des revues de code.

3.3 Méthodologie proposée

La méthodologie utilisée se base sur un devis de recherche quantitatif de type corrélationnel. Le but étant d'évaluer l'impact sur la durée nécessaire pour l'acceptation d'une revue de code suite à l'ajout d'une phase de préparation pour l'auteur.

3.4 Mise en œuvre

La démarche consiste à introduire une phase de préparation, pour l'auteur, afin d'évaluer l'impact de celle-ci sur le nombre d'itérations nécessaire pour l'approbation d'une revue de code. L'expérimentation a lieu sur cinq équipes, pour un total d'environ 70 participants et s'échelonne sur une période de six mois. Les étapes de la méthodologie sont décrites dans le Chapitre 4.

Chapitre 4

Approche proposée

Ce chapitre traite l'approche utilisée pour valider l'hypothèse de recherche évoquée dans le Chapitre 3. Il présente les différentes étapes à suivre qui constituent la stratégie de recherche.

L'hypothèse stipule qu'il est possible de réduire à un le nombre d'itérations de revues de code suite à l'ajout d'une phase de préparation pour l'auteur. Plus précisément, la phase de préparation consiste à demander à l'auteur d'une modification de faire les actions suivantes avant chaque soumission de revue de code :

- Valider ses modifications selon une liste de contrôle préétablie.
- Valider que la revue de code est la plus découpée/petite possible.
- Fournir une description aux vérificateurs (la raison de la modification, une explication du fonctionnement, et tout autre information jugée nécessaire pour la compréhension).
- Fournir une liste des cas de tests effectués.

Pour valider cette hypothèse, les étapes suivantes sont mises en œuvre :

1. Mise en place des outils de prise de mesure
    a. Prise de mesures — historique
2. Élaboration d'une liste de contrôle
    a. Sondage pour trouver les éléments les plus significatifs pour les réviseurs (voir Annexe I)
    b. Établissement d'une liste à partir des données recueillies
    c. Entrevue (de groupe) pour prioriser les éléments de la liste
    d. Prise de mesures
3. Expérimentation — groupe bêta

    a. Expliquer les consignes d'utilisation de la liste aux auteurs
    b. Utilisation de la liste sur un groupe bêta pour une période de trois mois
    c. Prise de mesures
    d. Analyse des mesures après trois mois
    e. Ajustement de la liste
4. Expérimentation — phase 2
    a. Mise en place sur un plus gros échantillon
    b. Expliquer les consignes d'utilisation de la liste aux auteurs
    c. Prise de mesures
    d. Analyse des mesures après trois mois
5. Cueillette et analyse des résultats

Chacune de ces étapes est expliquée dans le détail dans la section 4.2.

4.1 Échantillon

La population cible de cette étude est l'ensemble des développeurs, en entreprise, qui font réviser leur code à l'aide d'un outil de gestion de code source, tel Team Foundation Server (TFS) de Microsoft. Il est par contre très difficile de faire ce genre d'étude en impliquant tous les développeurs.

Pour les fins de cette essai, une entreprise ayant un peu plus de 200 développeurs a accepté de participer. Un échantillon de 50 développeurs a été sélectionné à l'aide de la méthode d'échantillonnage probabiliste en grappe. Plus précisément, les responsables de groupes de développements ont été sondés quant à leur intérêt à participer à l'étude. Cinq ont accepté. Il y a, en moyenne, une douzaine de développeurs, d'expérience hétéroclite, par équipe. De ces développeurs, il y a typiquement un ou deux réviseurs et un gestionnaire. La majorité des réviseurs sont aussi des auteurs, mais ne font que très peu de développement dans une année.

4.2 Description de l'approche

Cette section détaille les différentes étapes de l'approche.

4.2.1 Mise en place des outils de mesures

Pour être à même de valider l'hypothèse il est important d'avoir le nombre moyen d'itérations avant et après la mise en place de la phase de contrôle. Le logiciel de gestion de sources, Team Foundation Server (TFS), de Microsoft, possède déjà toute l'information nécessaire. Les revues de code soumis à partir du 1er janvier 2017 sont disponible pour les analyses de cette étude.

Les « DLL » Microsoft.TeamFoundation.WorkItemTracking.Client et Microsoft.TeamFoundation-Client, fournis par Microsoft, permettent la création d'un programme, en CSharp, qui interroge TFS afin d'extraire les révisions de chaque revue de code soumis par les participants. Dans TFS, chaque changement à la revue de code est considéré comme une révision. Pour le besoin de l'étude, seuls les changements de statut sont considérés. Les statuts TFS sont :

1. *Proposed* : création de la revue de code; celle-ci n'a pas été envoyée.
2. *Active* : la revue de code a été soumise pour révision. Il est possible qu'une revue de code soit active, mais qu'elle ne contienne aucun réviseur.
3. *Rejected* : la revue de code a été rejetée par au moins un réviseur.
4. *Approved* : la revue de code a été approuvée, avec ou sans commentaire, par tous les réviseurs. Il est possible que la revue soit approuvée sans réviseur.
5. *Cancelled* : l'auteur a annulé la revue de code.
6. *Checked-In* : les modifications de la revue de code ont été mises en production.

Une fois les révisions extraites de TFS, celles-ci sont gardées dans une base de données relationnelle pour des analyses futures. La Figure 4-1 présente le diagramme de celle-ci.

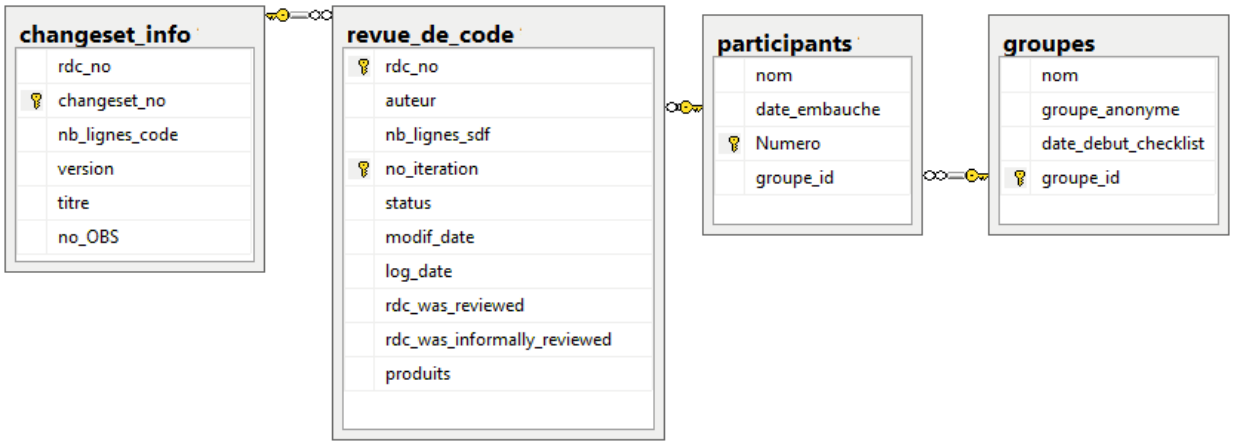


Figure 4-1 Diagramme de la base de données de mesures

Puisque les tables de la base de données contiennent des informations sur chaque étape d'une revue de code, une vue a été créé afin d'obtenir les statistiques groupées par revue de code (par numéro). En plus de regrouper, la vue filtre pour ne considérer que les revues de code qui ont été soumises avec au moins un réviseur. Ceci est nécessaire car l'outil utilisé, TFS, permet de créer des revues de code, et de les approuver, sans réviseur. Ces cas ne sont pas pertinents pour l'essai car aucune révision n'a été faite, il n'y a donc eu aucune possibilité de multiples itérations. Il est donc important de les filtrer dans les statistiques. À noter que TFS permet de spécifier des réviseurs sans que ceux-ci aient l'opportunité de la rejeter. Il s'agit d'une fonctionnalité utilisée par les participants pour aviser un nombre de personnes d'une modification sans leur demander formellement de la réviser. Ces cas sont aussi ignorés.

La vue contient les informations suivantes :

- rdc_no : numéro unique identifiant la revue de code;
- auteur : nom de l'auteur de la revue de code (anonymisé);
- groupe : utile pour les statistiques par groupe de travail;
- date_embauche : utile pour les statistiques par ancienneté;
- nb_soumissions : nombre de fois où le statut de la revue de code a été changé pour « *Active* »;

- nb_approbations : nombre de fois où le statut de la revue de code a été changé pour « *Approved* »;
- nb_rejets : nombre de fois où le statut de la revue de code a été changé pour « *Rejected* »;
- date_soumis : première date à laquelle la revue de code a été activée (statut est « *Active* »);
- date_approbation : date de la dernière approbation (statut est « Approved »);
- date_mep : date à laquelle la modification a été mise en production (statut est « *CheckedIn* »);
- rdc_avec_preparation : indique si la revue de code a été faite avant ou après l'ajout de la phase de préparation;
- nb_lignes_code : nombre de lignes de code modifiées (lors de la mise en production);
- type_version : indique s'il s'agit d'une modification faite dans une version cliente (en production) ou une version de développement;
- type_modif : indique s'il s'agit d'une correction ou d'un développement.

Le nombre de soumissions, de rejet, et d'approbations est conservé afin d'obtenir un meilleur aperçu du cheminement de la revue de code. Ces informations serviront à valider les cas limites. Par exemple, une revue de code peut avoir le cheminement suivant :

Proposed →Active →Cancelled →Active →Rejected →Active →Approved →CheckedIn

Dans ce cas-ci, nb_soumissions = 3, nb_approbations = 1, nb_rejets = 1.

Ici, le nombre de soumissions est plus élevé que le nombre de rejets plus un (on ajoute un pour la soumission initiale). Ceci est dû à l'annulation de la revue suite à une soumission. Aucune information n'est disponible quant à la raison pour laquelle la revue de code a été annulée. Entre le moment où la revue de code a été soumise et le moment où elle a été annulée, des réviseurs ont été contactés. Rien n'indique que ceux-ci n'ont pas déjà commencé leur révision, entraînant ainsi un coût en matière de temps de révision. Il est possible qu'un réviseur soit passé voir le développeur directement pour discuter de la modification, après quoi l'auteur annule la revue pour apporter des modifications. Ainsi, il évite que les autres réviseurs impliqués perdent leur temps à réviser une revue qui va

changer. Ou peut-être, est-ce l'auteur qui a remarqué une erreur et a voulu la corriger avant de la resoumettre. Environ 10 % des revues de code se retrouvent dans cette situation.

Pour les besoins de cette essai, le champ « nb_rejets » (auquel on ajoute 1) représente le nombre d'itérations de revues de code. Cela dit, le nombre de soumissions sera aussi analysé afin de déterminer si l'ajout de la phase de préparation impact le nombre d'annulations de revues de code.

TFS ne contient pas les données liées au nombre de lignes de code modifiées. La modification d'un script existant permet d'extraire ces informations dans un fichier csv, lequel est ensuite importé dans la base de données puis, intégré avec les autres statistiques. L'absence de cette donnée dans TFS ajoute une limitation quant à l'analyse de ces résultats. En effet, contrairement aux données obtenues de TFS, disponible par itération de revue de code, le nombre de lignes de code modifiées obtenu par ce script ne tient compte que de la modification finale (celle mise en production). Cette donnée est donc moins précise qu'initialement espérée. Par exemple, une première itération de la revue de code peut avoir modifié cent lignes de code alors que l'itération finale, celle mise en production, en modifie vingt. Dans les statistiques, la revue de code comptabilisera un total de vingt lignes modifiées. Cela dit, cette information demeure tout de même très pertinente afin d'obtenir un ordre de grandeur de la complexité de la modification.

Un chiffrier Microsoft Excel permet d'analyser les données de la vue via des tableaux et graphiques croisés dynamiquement. Des exemples sont présentés à la section 4.2.1.2.

4.2.1.1 Prise des mesures historiques

Une fois les outils de mesures en place, une première prise de mesure permet d'obtenir l'historique. Ceci sert à obtenir les différentes statistiques concernant le processus de revue de code avant l'ajout de la phase de préparation. Les données de l'année précédant l'expérimentation (2017) sont prises comme base de comparaison.

4.2.1.2 Résultats attendus

Le tableau qui suit présente un exemple des certaines mesures obtenues. Il montre des informations sur l'auteur (anonymisé), un numéro unique pour identifier chaque revue de code, le nombre d'itérations nécessaire pour la revue ainsi que la durée, en jour, entre la demande de revue et son acceptation.

Tableau 4-1 Mesures des revues de code

auteur	groupe	rdc_no	Années d'expérience	Durée revue (jours)	Nombre d'itérations
Participant 48	Groupe E	186152	5	0,108043981	2
Participant 38	Groupe D	198121	22	0,80130787	1
Participant 50	Groupe E	199402	5	0,733032407	1
Participant 49	Groupe E	209821	5	2,952291667	2
Participant 14	Groupe C	222374	0	20,83194444	4
Participant 50	Groupe E	228453	5	49,37969907	1
Participant 53	Groupe E	250460	8	2,868206019	1
Participant 20	Groupe B	181123	1	2,209965278	1

La durée de la revue de code est calculée en soustrayant la date de la dernière approbation (date_approbation) à la date de soumission (date_soumis). À noter qu'il est important de ne pas focaliser sur cette mesure car elle est sujet à interprétation. Plusieurs explications peuvent expliquer une durée élevée :

- Modification complexe à analyser;
- Réviseur surchargé/non-disponible (voir même en vacances);
- L'auteur a priorisé d'autres tâches entre les différentes itérations.

Le nombre d'années d'expérience est calculé en soustrayant la date de la soumission (date_soumis) à la date d'embauche de l'employé (date_embauche). Il est important d'utiliser la date de la soumission dans ce calcul et non la date du jour puisque c'est l'expérience du développeur au moment de la soumission de la revue de code qui nous importe.

L'analyse de ces résultats se fait à l'aide de tableaux et graphiques croisés dynamiquement, selon différents facteurs, par exemple, par années d'expérience et par groupe.

Tableau 4-2 Itérations par groupe

Groupes	Nombre d'itérations	Nombre d'itérations avec annulations	Durée RDC (jours)	Nb RDC
Groupe A	1.35	1.23	5.44	677
Groupe B	1.47	1.35	4.77	1117
Groupe C	1.48	1.38	4.24	631
Groupe D	1.42	1.31	1.97	1133
Groupe E	1.47	1.36	6.75	1414
Total général	**1.44**	**1.33**	**4.72**	**4972**

Ces tableaux nous permettent d'analyser l'impact de différents critères sur le nombre d'itérations moyen ainsi que la durée moyenne. Par exemple, les données historiques nous permettent d'analyser l'impact du nombre d'itérations sur la durée moyenne, en jours, d'une revue de code. C'est ce que présente la Figure 4-2.

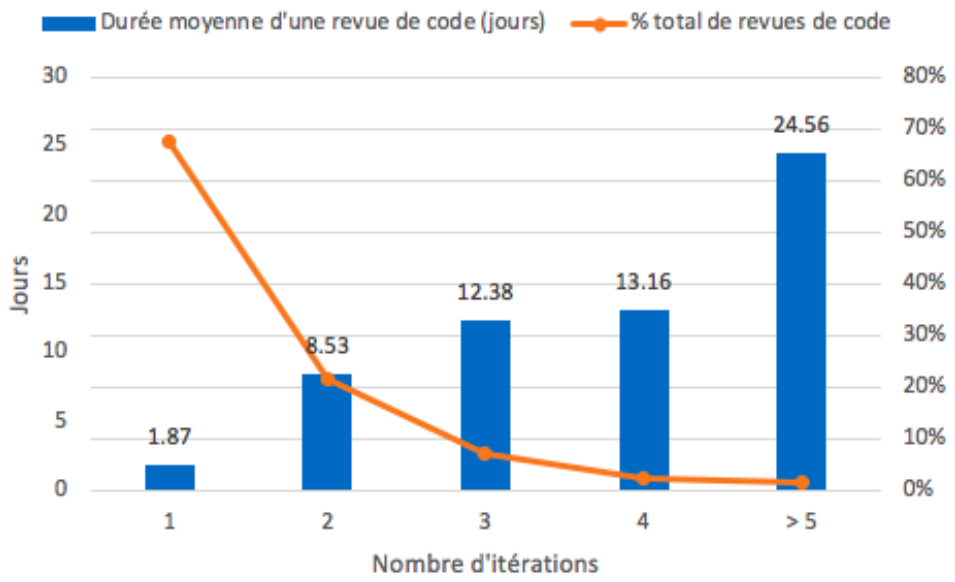


Figure 4-2 Durée, en jour, d'une revue de code en fonction du nombre d'itérations (sans préparation)

Les résultats finaux sont cueillis et analysés au chapitre 5.

4.2.2 Élaboration de la liste de contrôle

Pour définir une liste de contrôle efficace, elle doit répondre aux besoins précis de l'entreprise. Un sondage de dix questions est acheminé aux 16 réviseurs de l'échantillon, avec un délai d'une semaine, pour baliser les items de la liste de contrôle. Le temps moyen nécessaire pour répondre au questionnaire est de quinze minutes. Le sondage permet d'identifier les éléments d'améliorations les plus significatifs pour les réviseurs. Les réviseurs sont questionnés sur :

- les commentaires les plus fréquents qu'ils ont l'habitude de donner;
- leur impression de l'impact de la grosseur d'une modification sur la durée nécessaire pour le traitement d'une revue de code;
- ce qui les aiderait à traiter les revues de code plus rapidement;
- quels conseils ils donneraient aux développeurs pour faire approuver leurs modifications du premier coup.

La liste est ensuite construite à l'aide des résultats du sondage. Les résultats démontrent un besoin d'avoir plus d'informations sur les tests effectués, sur la source du problème et la solution apportée.

La liste contenant une trentaine d'éléments est ensuite validée par un sous-groupe de l'échantillon, c'est-à-dire quelques réviseurs et quelques développeurs. Le choix des participants à ce sous-groupe de contrôle est fait sur une base probabiliste par réseaux. En d'autres mots, les responsables de groupes suggèrent un à deux membres de leur équipe.

La validation se fait lors d'une rencontre dans laquelle chaque élément de la liste est discuté afin de déterminer sa pertinence, sa priorité ainsi que sa formulation. À la fin de la rencontre, les participants sont encouragés à indiquer les éléments qui sont manquants afin de discuter de leur ajout potentiel. À la toute fin, un total de 34 éléments ont été identifiés pour la liste de contrôle.

4.2.3 Expérimentation — groupe bêta

Une fois la liste complète et validée, l'expérimentation peut débuter. Pour s'assurer d'avoir un maximum d'impact, l'expérimentation est faite en deux phases. La première est faite sur un groupe bêta de 14 participants (un groupe de développement).

Une rencontre de groupe est organisée pour la présentation du projet, ses implications dans leur travail, la liste de contrôle ainsi que pour expliquer les consignes d'utilisation. Les participants sont encouragés à fournir leur rétroaction au fur et à mesure du projet. Ainsi, à la fin de la période d'essai, après trois mois, leurs commentaires servent à revalider la liste et y apporter des ajustements.

Après trois mois, de nouvelles mesures sont prises et analysées en comparaison avec celles prises avant l'introduction de la phase de préparation. Puis, une deuxième rencontre est organisée pour discuter des impacts qu'a eus cette nouvelle phase sur le travail des auteurs et des réviseurs. C'est aussi à ce moment que la liste de contrôle est revalidée pour s'ajuster aux différents commentaires soulevés par les participants.

4.2.4 Expérimentation — phase 2

Suite à l'expérimentation du groupe bêta et à l'ajustement de la liste de contrôle, une rencontre est planifiée avec chaque groupe de participants (quatre autres groupes de développement). Tout comme pour le groupe bêta, on leur présente le projet, ses implications et la liste de contrôle.

Trois mois après ces rencontres, les mesures sont reprises et analysées. C'est à ce moment que les résultats de l'étude sont obtenus.

4.2.5 Cueillette et analyse des résultats

À la suite de l'expérimentation, les résultats seront recueillis et analysés selon différents facteurs. Ils seront comparés aux résultats historiques cueillis avant le début de l'expérimentation.

4.3 Facteurs clés de succès

Pour que la démarche soit valide, il est important que la liste de contrôle soit définie avec les participants. Chaque entreprise, chaque groupe de développement, a sa façon de travailler, ses forces et faiblesses. C'est pourquoi l'utilisation d'une liste « générique » est ignorée. Il est plus judicieux de créer la liste en fonction des besoins de la population ciblée.

La définition de cette liste est difficile à valider car, il n'existe pas de barème permettant d'évaluer la pertinence des différents items.

4.4 Approche de validation des résultats

Pour que les résultats soient considérés comme valides, il faut au moins une vingtaine de participants avec des expériences variées, soit avec des auteurs juniors (moins de deux ans) et des séniors (plus de dix ans). Un minimum de trois mois de mesures doit être considéré, avec un minimum de 500 revues de code.

Un employé de l'entreprise dans laquelle a lieu l'expérimentation a le mandat de valider les données de cette étude. Ceci consiste à valider le programme d'extraction de données, la base de données ainsi qu'un chiffrier sous Microsoft Excel.

L'expérimentation a eu lieu durant la période du 10 avril 2018 au 16 novembre 2018. Les données recueillies sont présentées dans le chapitre cinq qui suit.

Chapitre 5

Analyse des résultats

5.1 Contraintes et échantillonnage

Pour des raisons de contraintes de travail, chaque groupe a commencé l'expérimentation à des moments différents, variant du début avril (groupe bêta) à la mi-août.

Puisque la période de données historiques disponible est plus longue (voir section 4.2.1.1) que celle de l'expérimentation, les données ont été filtrées pour ne considérer que les mois où l'expérimentation a eu lieu. Par exemple, pour le groupe C, les données de l'expérimentation s'échelonnent du 15 juillet 2018 au 15 novembre 2018. C'est pourquoi les données historiques considérées, c'est-à-dire sans phase de préparation, sont celles du 15 juillet 2017 jusqu'au 15 novembre 2017. L'application d'un tel filtre est nécessaire afin de s'assurer d'avoir des données comparables. Sans ce filtre, on obtient environ 6000 revues de code qui ont été faits sans la phase de préparation versus 500 avec la phase de préparation. Ceci a un impact sur l'analyse des résultats puisqu'une revue de code ayant considérablement plus d'itérations que les autres aura un impact beaucoup plus important sur la moyenne d'itérations en fonction du nombre total de revues de code analysés. En réduisant les données considérées, le nombre de revues de code avec et sans préparation est mieux distribué.

Le type de version de l'application (clientes versus de développement) a aussi dû être filtré pour ne considérer que les versions de développement car le nombre de revues de code dans les versions clientes est trop faible dans le groupe participant versus les groupes non participants (dix fois plus). La comparaison entre ces deux ensembles (participants vs non-participants) est pertinente afin de s'assurer que les gains réalisés par les participants sont réellement liés à l'implantation de la nouvelle phase de préparation.

Finalement, puisque l'expérimentation porte sur l'effet qu'a un ajout au processus actuel, seuls les auteurs ayant fait des revues de code à la fois pendant la période historique et d'expérimentation sont considérés. Conséquemment, les employés nouvellement embauchés, ainsi que ceux qui ont quitté avant le début de l'expérimentation, ont été filtrés des résultats. Cela dit, il est tout de même pertinent d'observer l'impact de la phase de préparation en fonction de l'expérience de l'auteur, particulièrement sur les nouveaux embauchés. C'est pourquoi ces participants sont considérés, exceptionnellement, dans l'analyse par ancienneté du Tableau 5-2.

5.2 Résultats obtenus

Les résultats sont analysés sous différents angles afin de mieux comprendre leur signification. La section qui suit présente le nombre d'itérations moyen, ainsi que leur durée moyenne avec et sans étape de préparation selon :

- le groupe de travail;
- l'ancienneté de l'auteur;
- le nombre de lignes de code modifiées par la revue de code.

Les statistiques ont aussi été analysées en fonction du réviseur afin de déterminer si un, ou plusieurs réviseurs influencent les résultats. Ces données n'ont pas été concluantes, elles ne sont donc pas présentés ici. Il en va de même pour le type de modification (correction versus développement).

Tel que mentionné plus tôt (section 4.2.1), les statistiques tiennent compte autant du nombre d'itérations (nombre de rejets plus un) que du nombre de soumissions (nombre d'itérations considérant les annulations). Ces données ont été filtrées de la présente section afin d'en faciliter la lecture. Elles sont par contre disponibles dans l'Annexe II. Pour bien comprendre les résultats, les concepts de moyenne et d'écart type seront nécessaires.

5.2.1 Durée par nombre d'itérations

La durée moyenne d'une itération, en jour, sans préparation, a été présentée à la Figure 4-2. Elle présente une augmentation marquée de la durée d'une revue de code en lien avec l'augmentation du nombre d'itérations. Rappelons que la durée est calculée entre le moment où la revue de code est soumise pour la première fois jusqu'au moment où elle est acceptée par tous les réviseurs. La Figure 5-1 présente cette même information mais cette fois-ci avec une phase de préparation. En comparant ces résultats avec ceux présentés précédemment, il est possible de voir une amélioration, peu importe le nombre d'itérations. Le lien entre le nombre d'itérations et la durée moyenne est toutefois moins évident; cela sera discuté plus loin dans la présentation des impacts sur la durée (section 5.3).

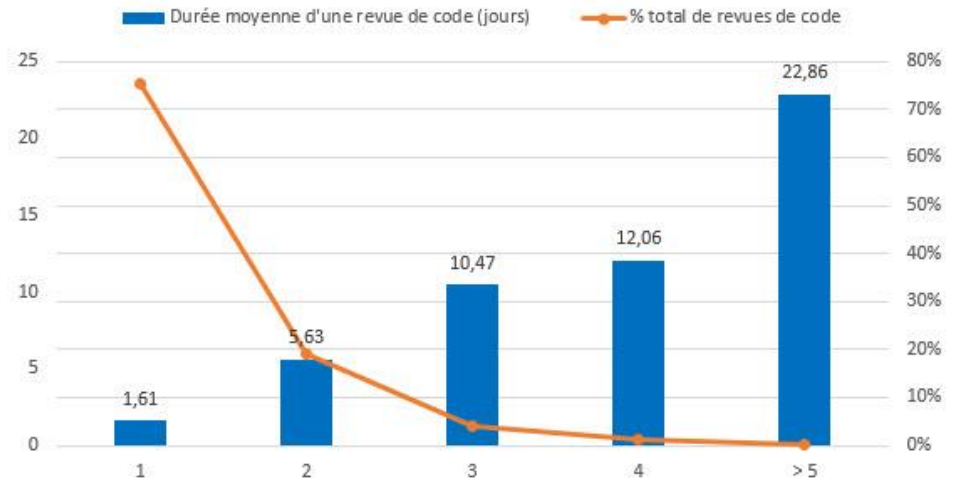


Figure 5-1 Durée moyenne d'une revue de code selon le nombre d'itérations (avec préparation)

5.2.2 Distribution des revues de code

La Figure 5-2 présente la distribution des revues de code en fonction du nombre d'itérations. On y observe une augmentation d'environ 7 % du nombre de revues de code ayant une seule itération. Considérant la durée moyenne des revues de code ayant plusieurs itérations, présentée à la Figure 5-1, ce gain paraît assez significatif.

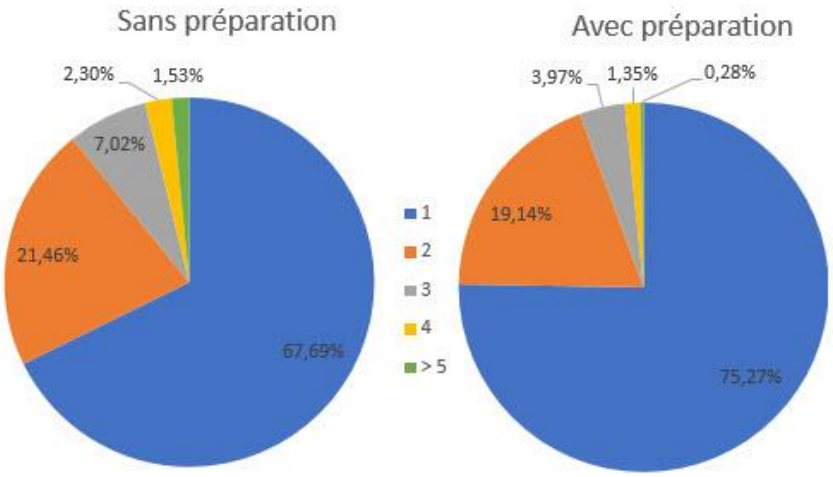


Figure 5-2 Distribution des revues de code par nombre d'itérations

5.2.3 Par groupe de travail

Le Tableau 5-1 présente le nombre d'itérations moyen, par groupe de travail, avec et sans la phase de préparation. Une amélioration globale de 8 % est observée pour l'ensemble des groupes, cela dit, l'analyse par groupe démontre une variation importante. Il est difficile de tirer quelconques conclusions des résultats individuels de chaque groupe puisque le nombre de revues de code soumis par chacune varie. Le groupe A a vu son nombre augmenter de 9 %, alors que le groupe E l'a vu diminuer de 20 %. L'augmentation peut s'expliquer par l'application de la consigne de réduire la taille des modifications. La diminution quant à elle est plus difficile à expliquer. Une hypothèse

est que les développements étaient de plus grosse envergures durant la période d'expérimentation, nécessitant plus de temps de conception que d'implémentation.

Tableau 5-1 Nombre d'itérations par groupe de travail

	Sans préparation		Avec préparation		
Groupe	Nombre moyen d'itérations	Nombre	Nombre moyen d'itérations	Nombre	Différence (%)
Groupe A	1,25	319	1,18	348	-6%
Groupe B	1,43	274	1,24	210	-14%
Groupe C	1,32	219	1,28	254	-3%
Groupe D	1,36	252	1,20	205	-12%
Groupe E	1,41	502	1,32	394	-6%
Total	**1,36**	**1566**	**1,25**	**1411**	**-8%**

5.2.4 Par ancienneté de l'auteur

Le Tableau 5-2 présente les mêmes informations que le Tableau 5-1, mais cette fois en fonction du nombre d'années d'expérience de l'auteur. Ceux-ci sont regroupés pour considérer le fait que les participants ont acquis un an d'expérience entre le moment de la prise de données historique, sans préparation, et le début de l'expérimentation.

Comme indiqué à la section 5.1, les employés nouvellement embauchés sont inclus dans ces résultats alors qu'ils ne le sont pas dans les autres tableaux. Il est intéressant de les analyser afin de voir l'impact que peut avoir une liste de contrôle sur des employés ayant peu ou pas d'expérience dans l'entreprise.

Tableau 5-2 Nombre d'itérations par année d'expérience de l'auteur

	Sans préparation		Avec préparation		
Nombre d'années d'expérience	Nombre moyen d'itérations	Nombre	Nombre moyen d'itérations	Nombre	Différence (%)
< 1	1,54	280	1,54	263	0%
[1- 3[	1,39	709	1,30	818	-6%
[3- 5[	1,20	127	1,22	195	2%
5 et plus	1,26	453	1,14	459	-10%

On remarque une diminution du nombre d'itérations plus marquée pour les auteurs ayant plus de 5 ans d'ancienneté. Cela pourrait s'explique par le fait qu'ils ont pris de mauvaises habitudes; que la liste de contrôle est venue leur remémorer les exigences auxquels ils doivent porter attention pour satisfaire les réviseurs. Il est par contre surprenant de voir qu'il n'y a absolument aucune amélioration pour les juniors (moins d'un an d'expérience). Une hypothèse pour expliquer ce résultat est que les développeurs font des tâches beaucoup plus simples dans leur première année, laissant moins de place à l'erreur et donc, à l'amélioration en termes de nombres d'itérations. Une autre possibilité est que ceux-ci, ayant reçu de nombreuses formations, utilisaient déjà une forme de liste de contrôle (notes prises lors des formations).

Le graphique présenté par la Figure 5-3 permet de voir que les développeurs ayant entre un et trois ans d'expérience sont le plus touchés par l'expérimentation car ils sont ceux qui produisent le plus de revues de code, groupe pour lequel on observe une amélioration de 6 %.

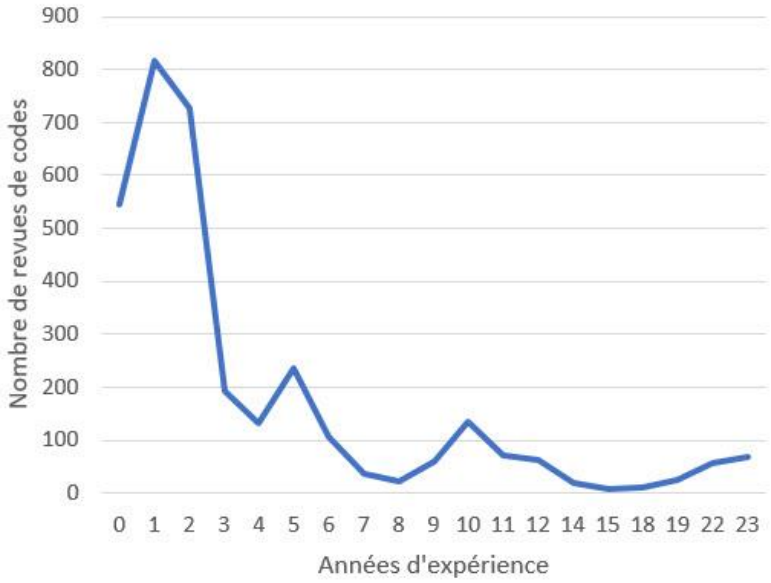


Figure 5-3 Nombre de revues de code par année d'expérience de l'auteur

5.2.5 Par nombre de lignes de code modifiées

Le nombre de lignes de code modifiées est non-négligeable lorsqu'on parle de durée ou de complexité de révision de code. C'est pourquoi cette information est incluse dans l'analyse du nombre d'itérations. Le

Tableau 5-3 présente les statistiques obtenues, par tranche de cent lignes de code modifiées, jusqu'à concurrence de 500 lignes.

Comme indiqué à la section 4.2.1, contrairement aux autres données, qui sont recueillies par itérations, les données concernant le nombre de lignes de code modifiées sont calculées sur la dernière itération seulement, c'est-à-dire au moment de la mise en production.

Tableau 5-3 Nombre d'itérations par nombre de lignes de code modifiées

	Sans préparation		Avec préparation		
Nombre de lignes modifiés	Nombre moyen d'itérations	Nombre	Nombre moyen d'itérations	Nombre	Différence (%)
1-100	1,16	936	1,13	888	-3%
101-200	1,50	187	1,34	185	-11%
201-300	1,61	101	1,36	95	-16%
301-400	1,77	79	1,53	59	-14%
401-500	1,58	43	1,60	43	1%
>501	1,80	220	1,60	141	-12%
Total	**1,36**	**1566**	**1,25**	**1411**	**-8%**

L'augmentation du nombre moyen d'itérations en fonction du nombre de lignes de code modifiées, surtout sans la phase de préparation, vient appuyer les études précédentes sur l'impact de la taille d'une modification sur la revue de code.

Les résultats ne démontrent pas de lien entre le nombre de lignes de code modifiées et l'ajout d'une phase de préparation; le gain est similaire pour la majorité des groupes. On remarque que les modifications de moins de 100 lignes de code, représentant environ 60 % des revues de code, ont un gain assez petit comparé aux autres. Cela s'explique en comparant le nombre moyen d'itérations, sans la phase de préparation, qui est de 1,16 comparé aux autres groupes, qui est de plus de 1,5. Cette différence rend le gain plus difficile à obtenir. Le résultat du groupe entre 400 et 500 lignes de code modifiées s'expliquent aussi par le petit nombre de revues de code qu'il contient.

5.3 Impacts sur la durée

Une diminution globale de 8 % du nombre d'itérations moyen est observée. Bien que les Figure 4-2 et Figure 5-1 sous-entendent un lien entre le nombre d'itérations et la durée, rien ne garantit qu'une réelle diminution sera observée au niveau de la durée. Plusieurs autres facteurs peuvent venir influencer la durée, tels la complexité de chaque tâche en revue de code, les vacances et congé des réviseurs et auteurs, leur charge de travail. Néanmoins, nous observons une diminution intéressante sur la durée moyenne d'une

revue de code. C'est cette diminution qui se traduit en gain pour l'entreprise. Cela dit, l'analyse des données, par groupe, ne nous permet pas d'affirmer qu'il existe bel et bien un lien entre ces deux variables. Les groupes A et E ont tous les deux eu une diminution de 6 % sur le plan du nombre d'itérations alors que la différence sur la durée est assez significative, 34 % de gain versus 59 %. Cette différence est difficile à expliquer. Peut-être est-ce le fait que le groupe E a vu son nombre de revues de code diminuer de 20 % ?

Tableau 5-4 Durée moyenne d'une revue de code, par groupe

Groupe	Différence nombre d'itérations (%)	Durée moyenne sans préparation (jours)	Durée moyenne avec préparation (jours)	Différence durée (%)
Groupe A	-6%	6,95	4,58	-34%
Groupe B	-14%	4,97	3,11	-37%
Groupe C	-3%	2,97	3,13	5%
Groupe D	-12%	1,97	1,39	-30%
Groupe E	-6%	5,07	2,07	-59%
Total	**-8%**	**4,64**	**2,93**	**-37%**

5.4 Groupes non participants

Le Tableau 5-5 présente le nombre d'itérations moyen ainsi que la durée moyenne d'une revue de code (en jours) pour les groupes non participants. Ces données sont comparées pour les mois d'avril à novembre de l'année historique (2017) et de l'année de l'expérimentation (2018).

Il est intéressant de regarder ces résultats afin de s'assurer que les gains des participants sont bel et bien attribuables à l'ajout de la phase de préparation et non à toute autre modification qui ont pu être mis en place par l'entreprise. Malheureusement, après analyse des résultats, on se rend compte que ces groupes ne peuvent pas être utilisés comme groupe de validation puisque le nombre d'itérations moyen historique est significativement plus bas pour la plupart de ces groupes (sauf un) que pour les groupes participants (1,36 vs 1,21). Ceci fait en sorte que le gain est plus difficile. Les groupes F

et H ont des moyennes historiques qui se rapprochent de celles des groupes participants. C'est pour ces groupes que le gain est le plus significatif. Ceci laisse croire que le gain fait par les équipes participantes n'a pas de lien avec la phase de préparation.

Tableau 5-5 Statistiques pour groupes non participants

	2017			2018				
Groupe	**Nombre moyen d'itérations**	**Durée moyenne revue (en jours)**	**Nombre**	**Nombre moyen d'itérations**	**Durée moyenne revue (en jours)**	**Nombre**	**Diff (%) itérations**	**Diff (%) Durée**
Groupe F	1,31	4,15	177	1,15	2,17	184	-12%	-48%
Groupe G	1,23	2,76	77	1,08	2,29	63	-13%	-17%
Groupe H	1,29	3,98	383	1,07	2,19	480	-17%	-45%
Groupe I	1,19	2,01	365	1,25	2,01	316	5%	0%
Groupe J	1,10	2,00	452	1,13	1,70	571	2%	-15%
Total	**1,21**	**2,82**	**1454**	**1,14**	**1,98**	**1614**	**-6%**	**-30%**

5.5 Revues de code annulées

Tel que mentionné à la section 4.2.1, le nombre d'itérations moyen peut être calculé en considérant, ou non, les annulations. Si l'on considère les annulations de revues de code comme une itération supplémentaire, le gain moyen passe de 8 % à 11 % pour les groupes participants. La différence varie d'un groupe à l'autre, mais est assez stable, environ 2 % à 3 % par groupe.

Pour les groupes non participants, le groupe G se démarque en passant d'une diminution de 13 % à 3 %. Ce qui laisse croire qu'ils ont annulé plus de revues de code en 2018 qu'en 2017. Pour les autres, c'est quasiment identique. Puisque le groupe G a significativement moins de revues de code (4 % du total) que les autres, son influence sur le total des non-participants est moindre, le laissant indemne, à 6 %.

Le fait de considérer, ou non, les annulations ne changent donc pas les résultats de l'expérimentation.

Les résultats détaillés sont présentés dans l'Annexe II.

5.6 Retour sur les hypothèses

Selon les résultats de l'étude, l'hypothèse selon laquel l'ajout d'une phase de préparation est suffisant pour diminuer à un le nombre d'itérations n'est pas soutenue. Il semble que cette hypothèse soit un peu trop optimiste. Une diminution d'environ 8 % du nombre d'itérations moyen a été observée, ce qui est un gain, mais une diminution jusqu'à une seule itération n'a pas été atteinte, telle que stipulée par l'hypothèse. Cette diminution entraîne une réduction de 37 % de la durée moyenne d'une revue de code, ce qui est un gain non négligeable, surtout pour les gestionnaires. Cela dit, l'analyse sur les groupes non participants démontre aussi une diminution importante, de 6 % pour le nombre d'itérations et 30 % pour la durée moyenne d'une revue de code. Il est donc difficile d'en tirer une quelconque conclusion. Il serait intéressant de pousser l'expérimentation plus loin afin d'évaluer la viabilité des résultats sur une plus longue période, et ce, avec des groupes de validation comparable.

Conclusion

Depuis leur présentation, en 1976, les revues de code ont peu ou pas changé. L'implémentation qu'en font les différentes entreprises varie en termes de rigidité mais les étapes demeurent les mêmes. Plusieurs hésitent à les inclure à leurs processus parce qu'elles sont jugées nécessitant trop de temps, temps qui manque de plus en plus pour terminer les projets dans des délais compétitifs du marché. Plusieurs études démontrent le gain qu'apportent les revues de code, justifiant ainsi leur utilisation répandue. Cela dit, il y a toujours place à amélioration.

Une recherche effectuée chez Microsoft [10] semblait un bon point de départ. Celle-ci présente des recommandations d'améliorations à apporter au processus de revues de code, autant pour les auteurs que pour les réviseurs. Ces suggestions sont basées sur les résultats d'une étude qualitative (à l'aide d'entrevues et de sondages).

Il restait donc à déterminer si, ces recommandations impactent réellement le processus de revues de code. Pour effectuer cette validation, l'hypothèse formulée propose de focaliser sur une seule amélioration, l'ajout d'une phase de préparation par l'auteur, afin d'analyser son effet sur le nombre d'itérations moyen nécessaire pour l'acceptation finale d'une revue de code.

La méthodologie utilisée pour valider cette hypothèse consistait à recueillir des statistiques avant et après l'ajout de la phase de préparation pour ensuite évaluer les répercussions de celle-ci sur le nombre d'itérations moyen. Un sondage et une entrevue (en groupe) permettent de définir une phase de préparation (via une liste de contrôle) personnalisée aux besoins des participants. L'anlayse du nombre d'itérations et de la durée totale de chaque revue de code permet d'obtenir une meilleure compréhension de l'impact réel qu'une entreprise pourrait récolter en ajoutant une telle phase.

Les résultats ne permettent malheureusement pas d'affirmer que cette nouvelle phase de préparation réduit le nombre d'itérations à 1, tel que stipule l'hypothèse. Cela dit, une amélioration de 8 % a tout de même été observée. Cette réduction du nombre

d’itérations moyen se traduit par une réduction de 37 % de la durée moyenne d’une revue de code.

Ces résultats sont encourageants, mais très liés aux besoins des participants (détails de la phase de préparation et la liste de contrôle). Des études futures seraient pertinentes afin d’évaluer la reproductivité de ces résultats avec des participants et besoins différents.

Pour permettre de mieux comprendre les résultats, il serait aussi pertinent d’analyser d’autres critères, tel l’impact de cet ajout sur le temps nécessaire pour la révision ainsi que sur le nombre d’anomalies liés à ces modifications.

Liste des références

[1] « Analysis of maintenance work categories through measurement », Proceedings. Conference on Software Maintenance 1991, Software Maintenance, 1991., Proceedings. Conference on, p. 104, 1991.

[2] « IEEE Standard for Software Reviews », *IEEE Std 1028-1997*, p. i‑ 37, mars 1998.

[3] E. P. Doolan, « Experience with Fagan's inspection method », *Softw. Pract. Exp.*, vol. 22, n° 2, p. 173–182, 1992.

[4] M. E. Fagan, « Design and code inspections to reduce errors in program development », *IBM Syst. J.*, vol. 38, n° 2/3, p. 258, mars 1999.

[5] J. M. Stecklein, J. Dabney, B. Dick, B. Haskins, R. Lovell, et G. Moroney, « Error cost escalation through the project life cycle », 2004.

[6] J. Cohen, E. Brown, B. DuRette, et S. Teleki, *Best kept secrets of peer code review*. Smart Bear, 2006.

[7] « Expectations, outcomes, and challenges of modern code review », *2013 35th Int. Conf. Softw. Eng. ICSE Softw. Eng. ICSE 2013 35th Int. Conf. On*, p. 712, 2013.

[8] S. So, Y. Lim, S. D. Cha, et Y. R. Kwon, « An empirical study on software error detection: voting, instrumentation and Fagan inspection », dans *Proceedings 1995 Asia*

Pacific Software Engineering Conference, 1995, p. 345‑351.

[9] A. Aurum, Håkan Petersson, et C. Wohlin, « State-of-the-art: software inspections after 25 years », *Softw. Test. Verification Amp Reliab.*, vol. 12, n° 3, p. 133‑154, 2002.

[10] L. MacLeod, M. Greiler, M. A. Storey, C. Bird, et J. Czerwonka, « Code Reviewing in the Trenches: Understanding Challenges and Best Practices », IEEE Software, vol. 35, n° 4, p. 34‑42, juill. 2018.

[11] P. Weißgerber, D. Neu, et S. Diehl, « Small Patches Get In! », *ICSE Int. Conf. Softw. Eng.*, p. 67‑75, févr. 2008.

[12] O. Baysal, O. Kononenko, R. Holmes, et M. W. Godfrey, « The influence of non-technical factors on code review », dans *2013 20th Working Conference on Reverse Engineering (WCRE)*, 2013, p. 122‑131.

[13] F. Ebert, F. Castor, N. Novielli, et A. Serebrenik, « Confusion Detection in Code Reviews », dans *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2017, p. 549‑553.

[14] M. V. Mäntylä et C. Lassenius, « What Types of Defects Are Really Discovered in Code Reviews? », *IEEE Trans. Softw. Eng. Softw. Eng. IEEE Trans. IIEEE Trans Softw. Eng*, n° 3, p. 430, 2009.

[15] A. Bosu, J. C. Carver, C. Bird, J. Orbeck, et C. Chockley, « Process Aspects and Social Dynamics of Contemporary Code Review: Insights from Open Source Development and Industrial Practice at Microsoft », *IEEE Trans. Softw. Eng.*, vol. 43, n° 1, p. 56‑75, 2017.

[16] G. L. . Jenkins glenn. l. jenkins@smu. ac. u. et O. Ademoye kemi. ademoye@smu. ac. u., « Can Individual Code Reviews Improve Solo Programming on an Introductory Course? », *Ital. Innov. Teach. Learn. Inf. Comput. Sci.*, vol. 11, n° 1, p. 71‑79, juin 2012.

[17] L. Hatton, « Testing the Value of Checklists in Code Inspections », *IEEE Softw.*, vol. 25, n° 4, p. 82‑88, juill. 2008.

[18] D. A. McMeekin, B. R. von Konsky, E. Chang, et D. J. A. Cooper, « Checklist Inspections and Modifications: Applying Bloom's Taxonomy to Categorise Developer Comprehension », 2008, p. 224‑229.

[19] D. A. McMeekin, B. R. von Konsky, E. Chang, et D. J. Cooper, « Checklist based reading's influence on a developer's understanding », dans *Software Engineering, 2008. ASWEC 2008. 19th Australian Conference on*, 2008, p. 489–496.

[20] J. Buckley et C. Exton, « Bloom's taxonomy: a framework for assessing programmers' knowledge of software systems », dans *11th IEEE International Workshop on Program Comprehension, 2003.*, 2003, p. 165‑174.

[21] O. Kononenko, O. Baysal, et M. W. Godfrey, « Code Review Quality: How Developers See It », dans Proceedings of the 38th International Conference on Software Engineering, New York, NY, USA, 2016, p. 1028–1038.

[22] M. E. Fagan, « Advances in software inspections », IEEE Transactions on Software Engineering, vol. SE-12, no 7, p. 744-751, juill. 1986.

[23] L. Hatton, « Testing the Value of Checklists in Code Inspections », IEEE Software, vol. 25, no 4, p. 82-88, juill. 2008.

Bibliographie

P. M. JOHNSON, « A 2 o-year-old inspection technique has served developers well in the quest for software quality improvement. But radical changes are on the horizon that may seriously steer the future offormal review. », vol. 41, no 2, p. 5, 1998.

B. Bernardez, M. Genero, A. Duran, et M. Toro, « A controlled experiment for evaluating a metric-based reading technique for requirements inspection », 2004, p. 257-268.

« A replicated experiment of usage-based and checklist-based reading », 10th International Symposium on Software Metrics, 2004. Proceedings., Software Metrics, 2004. Proceedings. 10th International Symposium on, Software metrics, p. 246, 2004.

M. di Biase, M. Bruntink, et A. Bacchelli, « A Security Perspective on Code Review: The Case of Chromium », 2016, p. 21-30.

Y. Chernak, « A statistical approach to the inspection checklist formal synthesis and improvement », IEEE Transactions on Software Engineering, vol. 22, no 12, p. 866–874, 1996.

J. Shimagaki, Y. Kamei, S. McIntosh, A. E. Hassan, et N. Ubayashi, « A Study of the Quality-impacting Practices of Modern Code Review at Sony Mobile », dans Proceedings of the 38th International Conference on Software Engineering Companion, New York, NY, USA, 2016, p. 212–221.

B. Brykczynski, « A survey of software inspection checklists », ACM SIGSOFT Software Engineering Notes, vol. 24, no 1, p. 82, 1999.

M. Bernhart, A. Mauczka, et T. Grechenig, « Adopting Code Reviews for Agile Software Development », 2010, p. 44-47.

O. S. Akinola et A. O. Osofisan, « An Empirical Comparative Study of Checklist based and Ad Hoc Code Reading Techniques in a Distributed Groupware Environment », sept. 2009.

Sun Sup So, Sung Deok Cha, T. J. Shimeall, et Yong Rae Kwon, « An empirical evaluation of six methods to detect faults in software », Software Testing: Verification & Reliability, vol. 12, no 3, p. 155, sept. 2002.

N. S. Eickelmann, F. Ruffolo, J. Baik, et A. Anant, « An empirical study of modifying the Fagan inspection process and the resulting main effects and interaction effects among defects found, effort required, rate of preparation and inspection, number of team members and product 1st pass quality », dans 27th Annual NASA Goddard/IEEE Software Engineering Workshop, 2002. Proceedings., 2002, p. 58-64.

P. C. Pendharkar et J. A. Rodger, « An empirical study of the impact of team size on software development effort », Information Technology and Management, vol. 8, no 4, p. 253-262, nov. 2007.

S. So, Y. Lim, S. D. Cha, et Y. R. Kwon, « An empirical study on software error detection: voting, instrumentation and Fagan inspection », dans Proceedings 1995 Asia Pacific Software Engineering Conference, 1995, p. 345-351.

« An experimental comparison of checklist-based reading and perspective-based reading for UML design document inspection », Proceedings International Symposium on Empirical Software Engineering, Empirical Software Engineering, 2002. Proceedings. 2002 International Symposium n, Empirical software engineering, p. 148, 2002.

T. Thelin, P. Runeson, et C. Wohlin, « An experimental comparison of usage-based and checklist-based reading », IEEE Transactions on Software Engineering, Software Engineering, IEEE Transactions on, IIEEE Trans. Software Eng., no 8, p. 687, 2003.

J. C. Knight et E. A. Myers, « An Improved Inspection Technique », Communications of the ACM, vol. 36, no 11, p. 51-61, nov. 1993.

« An internally replicated quasi-experimental comparison of checklist and perspective based reading of code documents », IEEE Transactions on Software Engineering, Software Engineering, IEEE Transactions on, IIEEE Trans. Software Eng., no 5, p. 387, 2001.

« Analysis of maintenance work categories through measurement », Proceedings. Conference on Software Maintenance 1991, Software Maintenance, 1991., Proceedings. Conference on, p. 104, 1991.

F. Macdonald, J. Miller, A. Brooks, M. Roper, et M. Wood, « Applying Inspection to Object-Oriented Code », Software Testing, Verification and Reliability, vol. 6, no 2, p. 61-82, juin 1996.

J. Cohen, E. Brown, B. DuRette, et S. Teleki, Best kept secrets of peer code review. Smart Bear, 2006.

J. R. de Almeida, J. Batista Camargo, B. Abrantes Basseto, et S. Miranda Paz, « Best practices in code inspection for safety-critical software », IEEE Software, vol. 20, no 3, p. 56-63, mai 2003.

J. Buckley et C. Exton, « Bloom's taxonomy: a framework for assessing programmers' knowledge of software systems », dans 11th IEEE International Workshop on Program Comprehension, 2003., 2003, p. 165-174.

G. L. . Jenkins glenn. l. jenkins@smu. ac. u. et O. Ademoye kemi. ademoye@smu. ac. u., « Can Individual Code Reviews Improve Solo Programming on an Introductory Course? », ITALICS : Innovations in Teaching & Learning in Information & Computer Sciences, vol. 11, no 1, p. 71-79, juin 2012.

D. A. McMeekin, B. R. von Konsky, E. Chang, et D. J. Cooper, « Checklist based reading's influence on a developer's understanding », dans Software Engineering, 2008. ASWEC 2008. 19th Australian Conference on, 2008, p. 489–496.

D. A. McMeekin, B. R. von Konsky, E. Chang, et D. J. A. Cooper, « Checklist Inspections and Modifications: Applying Bloom's Taxonomy to Categorise Developer Comprehension », 2008, p. 224-229.

N. Kitagawa, H. Hata, A. Ihara, K. Kogiso, et K. Matsumoto, « Code review participation: game theoretical modeling of reviewers in gerrit datasets », 2016, p. 64-67.

J. Czerwonka, M. Greiler, et J. Tilford, « Code Reviews Do Not Find Bugs: How the Current Code Review Best Practice Slows Us Down », dans Proceedings of the 37th International Conference on Software Engineering - Volume 2, Piscataway, NJ, USA, 2015, p. 27–28.

Olalekan S. Akinola et Ipeayeda Funmilola Wumi, « Comparative Study of the Effectiveness of Ad Hoc, Checklist- and Perspective-based Software Inspection Reading Techniques », International Journal of Computer Science and Information Security, Vol 9, Iss 11, Pp 163-172 (2011), no 11, p. 163, 2011.

Z. Abdelnabi, G. Cantone, M. Ciolkowski, et D. Rombach, « Comparing code reading techniques applied to object-oriented software frameworks with regard to effectiveness and defect detection rate », dans 2004 International Symposium on Empirical Software Engineering, 2004. ISESE '04. Proceedings, 2004, p. 239-248.

A. A. Porter, « Comparing Detection Methods for Software Requirements Inspections: A Replicated Experiment », p. 13.

F. Ebert, F. Castor, N. Novielli, et A. Serebrenik, « Confusion Detection in Code Reviews », dans 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME), 2017, p. 549-553.

P. C. Rigby et C. Bird, « Convergent contemporary software peer review practices », dans Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, 2013, p. 202–212.

The PLOS ONE Staff, « Correction: How Long Is Too Long in Contemporary Peer Review? Perspectives from Authors Publishing in Conservation Biology Journals », PLOS ONE, vol. 10, no 9, p. e0139783, sept. 2015.

M. E. Fagan, « Design and code inspections to reduce errors in program development », IBM Systems Journal, vol. 38, no 2/3, p. 258, mars 1999.

B. Meyer, « Design and Code Reviews in the Age of the Internet », Communications of the ACM, vol. 51, no 9, p. 66-71, sept. 2008.

T. Saika, E. Choi, N. Yoshida, S. Haruna, et K. Inoue, « Do Developers Focus on Severe Code Smells? », 2016, p. 1-3.

A. D. Da Cunha et D. Greathead, « Does Personality Matter? An Analysis of Code-Review Ability », Communications of the ACM, vol. 50, no 5, p. 109-112, mai 2007.

D. Rodríguez, M. A. Sicilia, E. García, et R. Harrison, « Empirical findings on team size and productivity in software development », Journal of Systems and Software, vol. 85, no 3, p. 562-570, 2012.

J. M. Stecklein, J. Dabney, B. Dick, B. Haskins, R. Lovell, et G. Moroney, « Error cost escalation through the project life cycle », 2004.

A. Rani et J. K. Chhabra, « Evolution of code smells over multiple versions of softwares: An empirical investigation », dans 2017 2nd International Conference for Convergence in Technology (I2CT), 2017, p. 1093-1098.

« Expectations, outcomes, and challenges of modern code review », 2013 35th International Conference on Software Engineering (ICSE), Software Engineering (ICSE), 2013 35th International Conference on, p. 712, 2013.

E. P. Doolan, « Experience with Fagan's inspection method », Software: Practice and Experience, vol. 22, no 2, p. 173–182, 1992.

T. Baum, O. Liskin, K. Niklas, et K. Schneider, « Factors influencing code review processes in industry », 2016, p. 85-96.

William G. Brozo et Norman A. Stahl, « Focusing on Standards: A Checklist for Rating Competencies of College Reading Specialists », Journal of Reading, vol. 28, no 4, p. 310-314, 1985.

W. G. Brozo et N. A. Stahl, « Focusing on Standards: A Checklist for Rating Competencies of College Reading Specialists. College Reading and Learning Assistance Technical Report 84-04 », janv. 1984.

G. Bavota et B. Russo, « Four eyes are better than two: On the impact of code reviews on software quality », dans 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME), 2015, p. 81-90.

A. Dunsmore, M. Roper, et M. Wood, « Further investigations into the development and evaluation of reading techniques for object-oriented code inspection », dans Proceedings of the 24th International Conference on Software Engineering. ICSE 2002, 2002, p. 47-57.

N. Sae-Lim, S. Hayashi, et M. Saeki, « How Do Developers Select and Prioritize Code Smells? A Preliminary Study », 2017, p. 484-488.

V. M. Nguyen et al., « How Long Is Too Long in Contemporary Peer Review? Perspectives from Authors Publishing in Conservation Biology Journals », PLOS ONE, vol. 10, no 8, p. e0132557, août 2015.

R. T. McCann, « How Much Code Inspection Is Enough? », CrossTalk: The Journal of Defense Software Engineering, vol. 14, nº 7, p. 9, juill. 2001.

R. Oliveira et al., « Identifying Code Smells with Collaborative Practices: A Controlled Experiment », 2016, p. 61-70.

« IEEE Standard for Software Reviews and Audits », IEEE Std 1028-2008, p. 1-52, août 2008.

Institute of Electrical and Electronics Engineers, IEEE standard for software verificiation and validation. New York: Institute of Electrical and Electronics Engineers, 2005.

A. Bosu et J. C. Carver, « Impact of Peer Code Review on Peer Impression Formation: A Survey », dans 2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement, 2013, p. 133-142.

R. L. Glass, « Inspections -- Some Surprising Findings », Communications of the ACM, vol. 42, no 4, p. 17-19, avr. 1999.

O. Kononenko, O. Baysal, L. Guerrouj, Y. Cao, et M. W. Godfrey, « Investigating code review quality: Do people and participation matter? », dans 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME), 2015, p. 111-120.

C. Denger, M. Ciolkowski, et F. Lanubile, « Investigating the active guidance factor in reading techniques for defect detection », dans 2004 International Symposium on Empirical Software Engineering, 2004. ISESE '04. Proceedings, 2004, p. 219-228.

S. Biffl et M. Halling, « Investigating the defect detection effectiveness and cost benefit of nominal inspection teams », IEEE Transactions on Software Engineering, vol. 29, no 5, p. 385-397, mai 2003.

B. Freimut, O. Laitenberger, et S. Biffl, « Investigating the impact of reading techniques on the accuracy of different defect content estimation techniques », dans Proceedings Seventh International Software Metrics Symposium, 2001, p. 51-62.

Ö. Albayrak et J. C. Carver, « Investigation of individual factors impacting the effectiveness of requirements inspections: a replicated experiment », Empirical Software Engineering, vol. 19, no 1, p. 241-266, 2014.

M. Beller, A. Bacchelli, A. Zaidman, et E. Juergens, « Modern code reviews in open-source projects: Which problems do they fix? », dans Proceedings of the 11th working conference on mining software repositories, 2014, p. 202–211.

S. D. Quinn, M. Souppaya, M. Cook, et K. A. Scarfone, « National Checklist Program for IT Products – Guidelines for Checklist Users and Developers », National Institute of Standards and Technology, NIST SP 800-70r3, déc. 2015.

A. Dunsmore, M. Roper, et M. Wood, « Object-oriented inspection in the face of delocalisation », dans Proceedings of the 22nd international conference on Software engineering, 2000, p. 467–476.

R. M. de Mello, R. F. Oliveira, et A. F. Garcia, « On the Influence of Human Factors for Identifying Code Smells: A Multi-Trial Empirical Study », 2017, p. 68-77.

T. Baum, K. Schneider, et A. Bacchelli, « On the Optimal Order of Reading Source Code Changes for Review », dans 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME), 2017, p. 329-340.

A. Dunsmore, M. Roper, et M. Wood, « Practical code inspection techniques for object-oriented systems: an experimental comparison », IEEE Software, vol. 20, no 4, p. 21-29, juill. 2003.

A. Bosu, J. C. Carver, C. Bird, J. Orbeck, et C. Chockley, « Process Aspects and Social Dynamics of Contemporary Code Review: Insights from Open Source Development and Industrial Practice at Microsoft », IEEE Transactions on Software Engineering, vol. 43, no 1, p. 56-75, 2017.

P. M. Johnson, « Reengineering Inspection », Communications of the ACM, vol. 41, no 2, p. 49, févr. 1998.

D. Mishra et A. Mishra, « Simplified software inspection process in compliance with international standards », Computer Standards & Interfaces, vol. 31, no 4, p. 763-771, juin 2009.

P. Weißgerber, D. Neu, et S. Diehl, « Small Patches Get In! », ICSE : International Conference on Software Engineering, p. 67-75, févr. 2008.

K. Owens, « Software detailed technical reviews: finding and using defects », dans WESCON/97 Conference Proceedings, 1997, p. 128-133.

F. Elberzhager, A. Klaus, et M. Jawurek, « Software Inspections Using Guided Checklists to Ensure Security Goals », 2009, p. 853-858.

S. Biffl et M. Halling, « Software product improvement with inspection. A large-scale experiment on the influence of inspection processes on defect detection in software requirements documents », dans Proceedings of the 26th Euromicro Conference. EUROMICRO 2000. Informatics: Inventing the Future, 2000, vol. 2, p. 262-269 vol.2.

M. Ciolkowski, O. Laitenberger, et S. Biffl, « Software reviews, the state of the practice », IEEE Software, vol. 20, no 6, p. 46-51, nov. 2003.

A. Aurum, Håkan Petersson, et C. Wohlin, « State-of-the-art: software inspections after 25 years », Software Testing : Verification & Reliability, vol. 12, no 3, p. 133-154, 2002.

A. L. Jacob et S. K. Pillai, « Statistical process control to improve coding and code review », IEEE software, vol. 20, no 3, p. 50–55, 2003.

M. H. Van Emden, « Structured Inspections of Code », Software Testing: Verification & Reliability, vol. 2, no 3, p. 133-153, sept. 1992.

A. Dunsmore, M. Roper, et M. Wood, « The development and evaluation of three diverse techniques for object-oriented code inspection », IEEE Transactions on Software Engineering, vol. 29, no 8, p. 677-686, août 2003.

G. Rong, J. Li, M. Xie, et T. Zheng, « The Effect of Checklist in Code Review for Inexperienced Students: An Empirical Study », 2012, p. 120-124.

S. McIntosh, Y. Kamei, B. Adams, et A. E. Hassan, « The Impact of Code Review Coverage and Code Review Participation on Software Quality: A Case Study of the Qt, VTK, and ITK Projects », dans Proceedings of the 11th Working Conference on Mining Software Repositories, New York, NY, USA, 2014, p. 192–201.

« The Impact of Design and Code Reviews on Software Quality: An Empirical Study Based on PSP Data », IEEE Transactions on Software Engineering, Software Engineering, IEEE Transactions on, IIEEE Trans. Software Eng., no 4, p. 534, 2009.

O. Baysal, O. Kononenko, R. Holmes, et M. W. Godfrey, « The influence of non-technical factors on code review », dans 2013 20th Working Conference on Reverse Engineering (WCRE), 2013, p. 122-131.

M. S. Krishnan, « The role of team factors in software cost and quality: An empirical analysis », Information Technology & People, vol. 11, no 1, p. 20–35, 1998.

R. de Mello, R. Oliveira, L. Sousa, et A. Garcia, « Towards Effective Teams for the Identification of Code Smells », 2017, p. 62-65.

« Trends in software maintenance tasks distribution among programmers: A study in a micro software company », 2017 IEEE 15th International Symposium on Intelligent Systems and Informatics (SISY), Intelligent Systems and Informatics (SISY), 2017 IEEE 15th International Symposium on, p. 000023, 2017.

D. Izquierdo-Cortazar, N. Sekitoleko, J. M. Gonzalez-Barahona, et L. Kurth, « Using Metrics to Track Code Review Performance », 2017, p. 214-223.

Y. Murakami, M. Tsunoda, et H. Uwano, « WAP: Does Reviewer Age Affect Code Review Performance? », 2017, p. 164-169.

S. Nelson et J. Schumann, « What makes a code review trustworthy? », dans 37th Annual Hawaii International Conference on System Sciences, 2004. Proceedings of the, 2004, p. 10 pp.-.

M. V. Mäntylä et C. Lassenius, « What Types of Defects Are Really Discovered in Code Reviews? », IEEE Transactions on Software Engineering, Software Engineering, IEEE Transactions on, IIEEE Trans. Software Eng., no 3, p. 430, 2009.

R. Oliveira, « When more heads are better than one?: understanding and improving collaborative identification of code smells », 2016, p. 879-882.

K. Hamasaki, R. G. Kula, N. Yoshida, A. E. C. Cruz, K. Fujiwara, et H. Iida, « Who does what during a code review? Datasets of OSS peer review repositories », dans 2013 10th Working Conference on Mining Software Repositories (MSR), 2013, p. 49-52.

Y. Jiang, B. Adams, et D. M. German, « Will my patch make it? And how fast? Case study on the Linux kernel », dans 2013 10th Working Conference on Mining Software Repositories (MSR), 2013, p. 101-110.

Annexe I

Sondage pour l’élaboration de la liste de contrôle

Cette annexe présente le questionnaire fourni à 16 participants afin d'aider dans l'élaboration de la liste de contrôle. On y retrouve les dix questions ainsi que les réponses. À noter que certaines questions et réponses ont été modifiées pour des besoins d'anonymisation de l'entreprise.

Question 1 : Quels commentaires laissez-vous le plus fréquemment ? Classez-les en ordre du plus prioritaire au moins prioritaire.

- Fautes d'orthographe
- Validation de pointeurs null/Division par 0
- Ajout/modification de commentaires dans le code
- Problème de logique/approche organique
- Optimisation du code/Refactorisation
- Ajout/modification de documentation
- Standards de programmation/Nomenclature
- Structure/Organisation du code
- Tests à effectuer/Versions à corriger
- Bogues

	1	2	3	4	5	6	7	8	9	10	TOTAL	SCORE
Fautes d'orthographe	12,50% 2	18,75% 3	0,00% 0	12,50% 2	6,25% 1	12,50% 2	6,25% 1	0,00% 0	12,50% 2	18,75% 3	16	5,50
Validation de pointeurs null/Division par 0	0,00% 0	6,25% 1	6,25% 1	6,25% 1	12,50% 2	12,50% 2	31,25% 5	12,50% 2	6,25% 1	6,25% 1	16	4,69
Ajout/modification de commentaires dans le code	12,50% 2	25,00% 4	18,75% 3	6,25% 1	0,00% 0	0,00% 0	6,25% 1	0,00% 0	12,50% 2	18,75% 3	16	6,13
Problème de logique/approche organique	18,75% 3	6,25% 1	0,00% 0	18,75% 3	18,75% 3	18,75% 3	0,00% 0	6,25% 1	6,25% 1	6,25% 1	16	6,19
Optimisation du code/Refactorisation	0,00% 0	0,00% 0	18,75% 3	18,75% 3	18,75% 3	6,25% 1	18,75% 3	18,75% 3	0,00% 0	0,00% 0	16	5,56
Ajout/modification de documentation	18,75% 3	12,50% 2	25,00% 4	6,25% 1	0,00% 0	12,50% 2	6,25% 1	18,75% 3	0,00% 0	0,00% 0	16	6,88
Standards de programmation / Nomenclature	18,75% 3	18,75% 3	12,50% 2	0,00% 0	12,50% 2	18,75% 3	6,25% 1	0,00% 0	12,50% 2	0,00% 0	16	6,75
Structure/Organisation du code	12,50% 2	12,50% 2	0,00% 0	18,75% 3	12,50% 2	12,50% 2	0,00% 0	25,00% 4	6,25% 1	0,00% 0	16	5,94
Tests à exécuter/Domods à faire	0,00% 0	0,00% 0	6,25% 1	12,50% 2	6,25% 1	0,00% 0	6,25% 1	0,00% 0	43,75% 7	25,00% 4	16	3,13
Bogues	6,25% 1	0,00% 0	12,50% 2	0,00% 0	12,50% 2	6,25% 1	18,75% 3	18,75% 3	0,00% 0	25,00% 4	16	4,25

Question 2 : Y a-t-il d'autres types de commentaires que vous laissez souvent ? Si oui, spécifiez.

- Justification qui accompagne les choix, référence vers les standards, explications des bonnes pratiques.
- Utilisation adéquate des comptes de références (pour gestion de la mémoire)
- Ajout de tests d'erreurs
- Rappeler le développeur d'ajouter le # de tâche dans la revue de code.
- Code de projet, Numéro de tâche TFS, documenter le rapport de bogue
- Ne pas oublier de mettre à jour la demande de révision et de test.
- Questions ou remise en question. Compliments

Question 3 : En moyenne, quel est le délai nécessaire au traitement d'une itération revue de code (approbation ou refus) ?

- Moins de 24 h – 37,5 %
- Entre 24 et 48 h – 31,25 %
- Moins d'une semaine — 6,25 %
- Autres (spécifiez) — 25 %
    - Aux alentours d'une semaine
    - Selon le type de RDC ça peut être moins de 24 h et 1 semaine
    - Idéalement, j'essaie entre 24 et 48 h. Mais ça arrive rarement, alors je dirais moins d'une semaine. :(
    - Selon la période de l'année, ça peut aller de 1 j à 15 j

Question 4 : D'après vous, quels sont les impacts de la taille de la modification sur la revue code ? Choisissez les énoncés qui s'appliquent.

- Plus elle est grosse, plus c'est long à commencer la revue — 18,75 %
- Plus elle est grosse, plus c'est long à valider — 25 %
- Plus elle est grosse, plus c'est difficile à suivre – 31,25 %

- Aucun impact — 0 %
- Autres (spécifiez) — 25 %
    - Plus c'est long ET difficile à approuver
    - Toutes ces réponses
    - (... plus c'est difficile à suivre). Si elle est trop grosse, je la rejette parfois sans même la regarder. Surtout avec les nouveaux pour être sûrs qu'ils ne se ressaient pas une 2e fois. C'est extrêmement important (surtout dans la plateforme) d'être capables de scinder les développements en « petites » modifications séquentielles.

Question 5 : Voudriez-vous avoir plus de commentaires de la part de l'auteur pour vous aider à comprendre la modification :

- Oui — 68,75 %
- Non — 12,5 %
- Ça m'importe peu – 18,75 %

Question 6 : Quelles sont, selon vous, les raisons les plus communes du rejet d'une revue de code ?

- Le développeur n'a pas compris le problème — 0 %
- La solution ne fonctionne pas — 0 %
- Trop d'ajustements à faire (de commentaires à revoir) – 62,5 %
- Autres (spécifiez) — 37,5 %
    - Moitié-moitié solution ne fonctionne pas ou trop d'ajustements
    - Pas certain de si le développeur a bien compris quoi ajuster
    - Le correctif n'est pas à la bonne place. Compréhension du code et des interactions entre les classes, responsabilités des classes.
    - Trop d'ajustement avec la solution ne fonctionne pas
    - Code fonctionne mais placé au mauvais endroit, manque le traitement de certains cas d'utilisation

- Trop d’ajustements à faire ou bien je ne comprends pas sa solution

Question 7 : Qu’est-ce qui pourrait vous aider à réduire le temps nécessaire pour la revue ?

- Développeurs valident leur idée et solution avec le réviseur avant de les implanter, tout simplement.
- Un diff side-by-side (les ++++ ---- imbriqués sont pas toujours évidents à interpréter). Côté développeur, un outil de revue qui permet de merger automatiquement les changements du réviseur, 90 % des commentaires sont à prendre tel quel et on perd du temps à recopier, en plus d’ajouter une chance de se tromper lors de la copie. Multisoumission, par étape, structure générale, détails, commentaires/doc (Pour grosses modifications), pourraient faire partie du processus d’analyse.
- Un résumé du changement, ce que manquait, et dans quel cas. Use case.
- Pour les modifications complexes : 1. Voir une courte démo par le développeur 2. Avoir une explication du problème et de la solution AVANT de coder pour donner ses premières impressions en général, avoir des outils de validations automatiques. Par exemple : 1. Validation automatique de la syntaxe 2. Validation de l’ordre des fonctions 3. Validation de conversions/doc intégrée dans les RDC
- Des petites modifications
- Autoapprobation avant d’envoyer la modification.
  Bien découper la tâche en plusieurs modifications.
  Indiquer si des cas n’ont pas été testés.
  Bien se relire pour les fautes d’orthographe.
  Écrire des commentaires concis et clairs (disant le pourquoi)
  Globalement, qu’ils se responsabilisent en ayant la volonté de soumettre du travail de qualité. Le travail du réviseur devrait être seulement un filet de sécurité pour des détails. Lorsqu’on doit demander 2-3-4 révision d’une même modification, il y a un problème...
- Décomposer les grosses modifications en d’autres plus petites et plus ciblées.
- Avoir des tooltips qui affichent la desc de la fonction pointée.

- Avoir l’explication sur le problème et la solution, cas testés avec la RDC. Ne pas avoir à regarder les détails comme standard, pointeurs null. Bref, se concentre sur la logique, sur qu’est-ce qu’il faut faire.
- Réduire la grosseur des revues de code. Dans le cadre d’une tâche, bien documenter les modifications effectuées. Pour les corrections de bogues, documenter la source du problème ainsi que la solution suggérée. Ne pas hésiter à faire valider leur solution avant de coder.
- Plus la modification est petite, plus c’est rapide à valider.
- Liste des tests effectués par le développeur pour me donner un sentiment de confiance. Petites modifications.
  Outil permettant au développeur de trier l’ordre des fichiers dans un ordre logique pour la revue.
- Que le code soit parfait en partant.

Question 8 : Avez-vous des conseils à donner aux développeurs pour les aider à faire approuver leurs revues du premier coup ?

- Bien checker le code soumis pour les erreurs évidentes : mauvaises indentations — documentation manquante - commentaires manquants — respect des standards.
- S’autoréviser (rapidement) avant de soumettre permet souvent de voir des erreurs ou de remettre en question certains aspects.
- Be perfect ! Relire les fichiers soumit, mais je ne crois pas que ce soit réalisable qu’une revue soit acceptée du premier coup de façon régulière. Sinon établir un plan, et réduire la taille des RDC, quitte à en faire plusieurs.
- Utiliser et comprendre des patterns : 1. Patterns d’utilisation de pointeurs intelligents 2. Patterns de documentation de fonctions (ex : « Return true if X, false otherwise » pour documenter le retour). Se faire une check-list personnelle d’erreurs qui reviennent souvent. Elle peut contenir seulement 3-4 items, mais elle aide à identifier et corriger ses erreurs pour s’améliorer.
- Le développeur révise son code à soumettre avant de l’envoyer

- S’assurer de bien tester. Écrire les tests réalisés est une bonne idée. S’assurer d’avoir relu la documentation.
- Autoapprobation avant d’envoyer la modification.
  Bien découper la tâche en plusieurs modifications.
  Indiquer si des cas n’ont pas été testés.
  Bien se relire pour les fautes d’orthographe
  Écrire des commentaires concis et clairs (disant le pourquoi.) Globalement, qu’ils se responsabilisent en ayant la volonté de soumettre du travail de qualité. Le travail du réviseur devrait être seulement un filet de sécurité pour des détails. Lorsqu’on doit demander 2-3-4 révision d’une même modification, il y a un problème....
- Faire valider la solution trouvée avant de la soumettre si celle-ci est grosse et/ou risquée et/ou pas tout à fait claire.
- Oublier ce qu’ils savent et relire leur documentation sans lire entre les lignes.
- Documenter le bogue/la tâche : cause, problème, solution, cas testé avant l’envoi de la revue de code. Valider solution avant de coder pour les cas complexes.
- Réduire grosseur des modifications.
- Faire les choses « fonctionner » n’est pas suffisant : il faut penser à la maintenabilité. Donc, je les suggère d’avoir une petite curiosité à comprendre le code qu’ils modifient. Ainsi, ça va éviter le refus de plusieurs mauvaises solutions.
- Mis à part documenter la solution comme mentionnée au point 7, le développeur pourrait aussi documenter qu’est-ce qu’il a testé. Cela va lui forcer à faire quelques tests, puis il pourra trouver des bogues.
- Regarder la revue de code et se mettre à la place de celui qui l’approuve. Comme on travaille toujours avec les mêmes personnes, on sait en regardant le code quel type de commentaires on risque d’avoir.
- Relire le code comme s’il était l’approbateur.
- TESTS !!!
- Demander des questions au futur réviseur durant le développement pour s’assurer d’être sur la même longueur d’onde.

Faire un effort pour écrire du code clair, avec de bons noms de fonction et variable, une bonne structure, de bons commentaires pertinents et une bonne documentation.

- Des tonnes. J’espère qu’il ne faut pas les documenter ici.

Question 9 : Voudriez-vous valider le texte de la documentation fonctionnelle ?

- Oui — 62,5 %
- Non — 37,5 %

Question 10 : Est-ce que vous voudriez que le développeur soit en charge de créer/modifier les demandes de révisions et de tests ?

- Oui — 53,33 %
- Non — 6,66 %

Dans certains cas seulement. Je vais l’aviser au besoin. — 41,17 %.

Annexe II

Résultats détaillés

La base de données, le logiciel d'extraction ainsi que le chiffrier excel qui ont servis pour receuillir les résultats de cette étude seront conservés pour une durée maximale d'un an.

Par groupes participants

	Sans préparation				Avec préparation					
Groupe	**Nombre moyen d'itérations**	**Écart Type**	**Nombre moyen d'itérations (avec annulations)**	**Nombre RDC**	**Nombre moyen d'itérations**	**Écart Type**	**Nombre moyen d'itérations (avec annulations)**	**Nombre RDC**	**Différence (%)**	**Différence avec annulations (%)**
Groupe A	1.25	0.74	1.40	319	1.18	0.55	1.26	348	-6%	-10%
Groupe B	1.43	0.93	1.58	274	1.24	0.68	1.32	210	-14%	-16%
Groupe C	1.32	0.81	1.42	219	1.28	0.67	1.36	254	-3%	-4%
Groupe D	1.36	0.95	1.48	252	1.20	0.60	1.26	205	-12%	-15%
Groupe E	1.41	0.87	1.53	502	1.32	0.74	1.39	394	-6%	-9%
Total	**1.36**	**0.86**	**1.49**	**1566**	**1.25**	**0.66**	**1.32**	**1411**	**-8%**	**-11%**

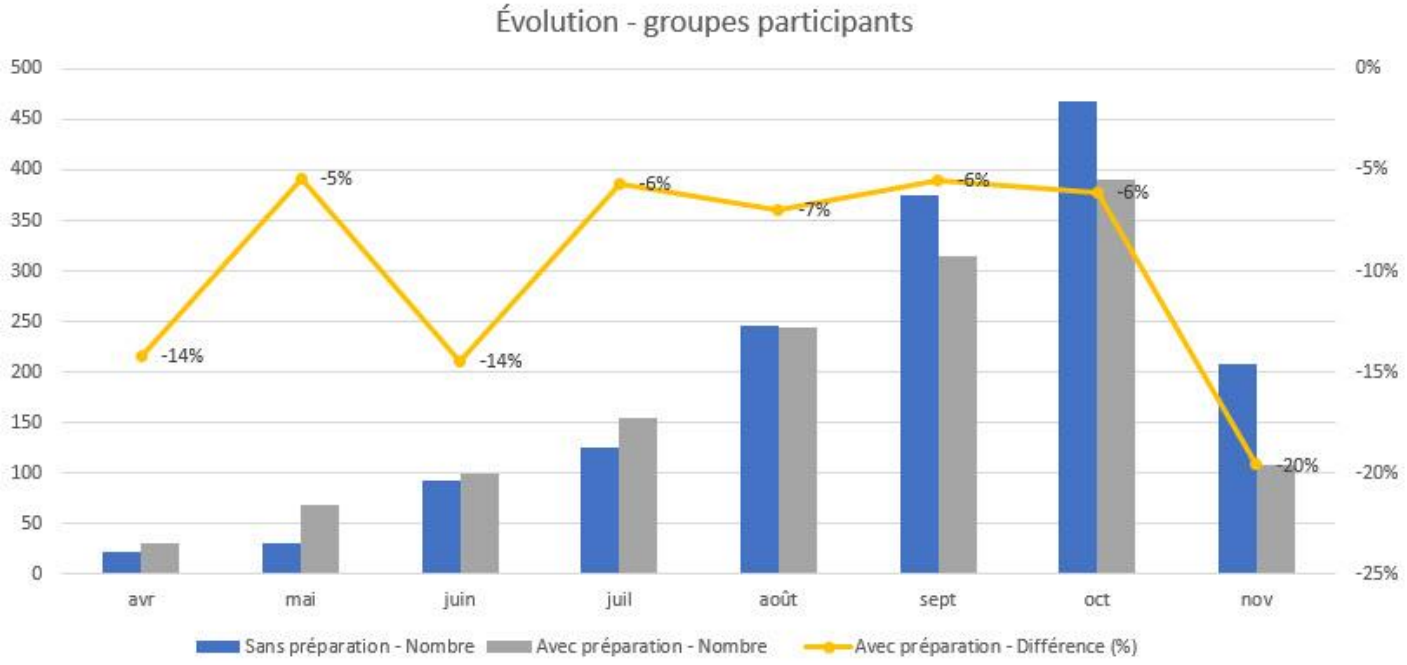
	Sans préparation		Avec préparation		
Groupe	**Durée moyenne revue (jours)**	**Écart Type**	**Durée moyenne revue (jours)**	**Écart Type**	**Différence (%) durée**
Groupe A	6.95	11.35	4.58	11.39	-34%
Groupe B	4.97	9.88	3.11	6.95	-37%
Groupe C	2.97	6.73	3.13	5.85	5%
Groupe D	1.97	5.29	1.39	4.01	-30%
Groupe E	5.07	15.64	2.07	4.47	-59%
Total	**4.64**	**11.62**	**2.93**	**7.37**	**-37%**

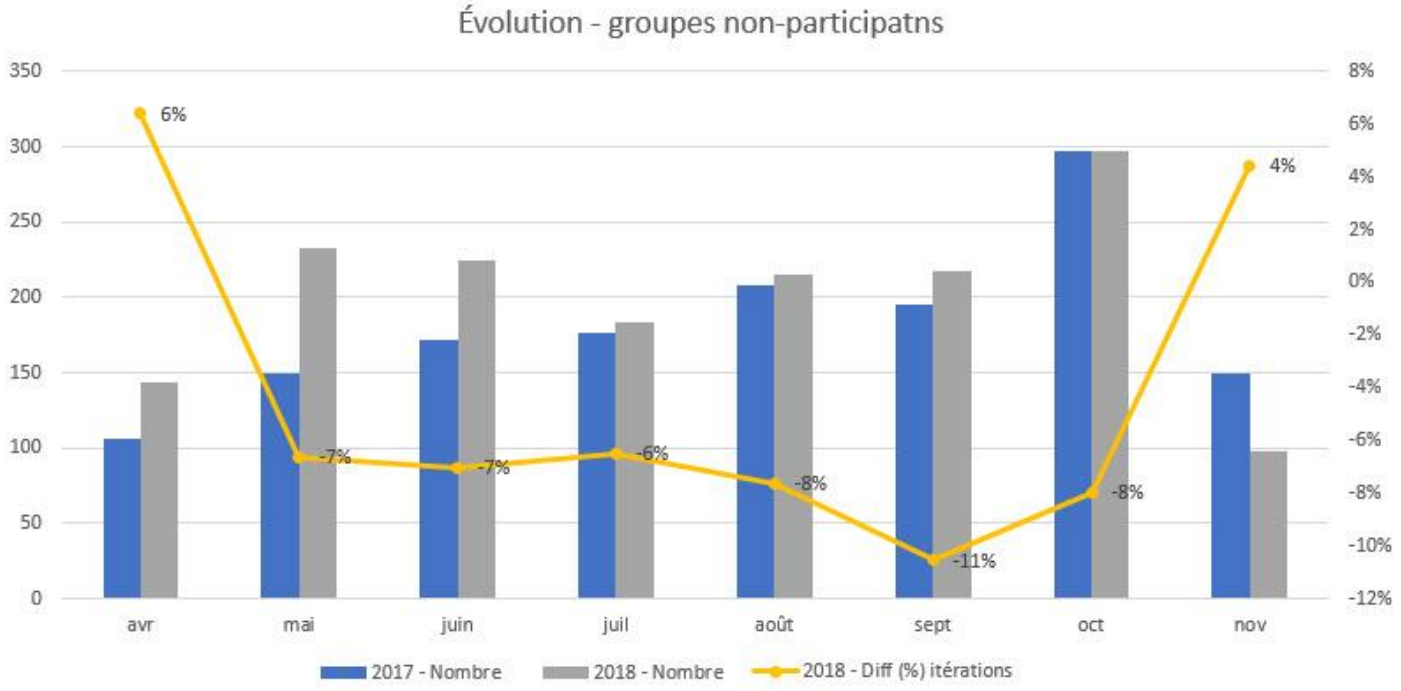
Par groupes non-participants

	Sans préparation				Avec préparation					
Groupe	Nombre moyen d'itérations	Écart Type	Nombre moyen d'itérations (avec annulations)	Nombre RDC	Nombre moyen d'itérations	Écart Type	Nombre moyen d'itérations (avec annulations)	Nombre RDC	Différence (%)	Différence avec annulations (%)
Groupe F	1.31	0.67	1.38	177	1.15	0.44	1.22	184	-12%	-12%
Groupe G	1.23	0.51	1.27	77	1.08	0.27	1.24	63	-13%	-3%
Groupe H	1.29	0.59	1.41	383	1.07	0.29	1.14	480	-17%	-19%
Groupe I	1.19	0.47	1.27	365	1.25	0.60	1.31	316	5%	4%
Groupe J	1.10	0.35	1.18	452	1.13	0.43	1.20	571	2%	2%
Total	**1.21**	**0.51**	**1.29**	**1454**	**1.14**	**0.44**	**1.21**	**1614**	**-6%**	**-6%**

	Sans préparation		Avec préparation		
Groupe	Durée moyenne revue (jours)	Écart Type	Durée moyenne revue (jours)	Écart Type	Différence (%) durée
Groupe F	4.15	12.58	2.17	3.50	-48%
Groupe G	2.76	7.17	2.29	5.99	-17%
Groupe H	3.98	14.55	2.19	4.00	-45%
Groupe I	2.01	4.91	2.01	4.42	0%
Groupe J	2.00	6.89	1.70	3.45	-15%
Total	**2.82**	**9.96**	**1.98**	**3.95**	**-30%**

Par mois


Évolution - groupes participants
500
450
400
350
300
250
200
150
100
50
0
0%
-5%
-10%
-15%
-20%
-25%
-14%
-5%
-14%
-6%
-7%
-6%
-6%
-20%
avr
mai
juin
juil
août
sept
oct
nov
Sans préparation - Nombre
Avec préparation - Nombre
Avec préparation - Différence (%)


Évolution - groupes non-participatns
350
300
250
200
150
100
50
0
8%
6%
4%
2%
0%
-2%
-4%
-6%
-8%
-10%
-12%
6%
-7%
-7%
-6%
-8%
-11%
-8%
4%
avr
mai
juin
juil
août
sept
oct
nov
2017 - Nombre
2018 - Nombre
2018 - Diff (%) itérations