UNIVERSITÉ DE MONTRÉAL

LA GESTION DE LA CONNAISSANCE DES ÉQUIPES DE DÉVELOPPEMENT LOGICIEL

MATHIEU LAVALLÉE
DÉPARTEMENT DE GÉNIE INFORMATIQUE ET GÉNIE LOGICIEL
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

THÈSE PRÉSENTÉE EN VUE DE L'OBTENTION
DU DIPLÔME DE PHILOSOPHIÆ DOCTOR
(GÉNIE INFORMATIQUE)
FÉVRIER 2017

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Cette thèse intitulée :

LA GESTION DE LA CONNAISSANCE DES ÉQUIPES DE DÉVELOPPEMENT LOGICIEL

présentée par : LAVALLÉE Mathieu
en vue de l'obtention du diplôme de : Philosophiæ Doctor
a été dûment acceptée par le jury d'examen constitué de :

M. GUIBAULT François, Ph. D., président
M. ROBILLARD Pierre N., D. Sc., membre et directeur de recherche
M. KHOMH Foutse, Ph. D., membre
M. HARDY Simon, Ph. D., membre externe

DÉDICACE

À mes parents,
qui ont su m'encourager à aller vers
des destinations que je croyais
moi-même hors de portée.

REMERCIEMENTS

Ce travail n'aurait pas été possible sans l'accord des multiples personnes étudiées et qui ont permis de mieux comprendre le développement logiciel en général, et leur expérience personnelle en particulier. L'auteur désire leur envoyer ses remerciements les plus chaleureux.

Les résultats n'auraient pas non plus été possible sans l'immense expérience et, surtout, l'incommensurable patience du directeur de cette thèse, le Dr. Pierre N. Robillard. Il a toujours su se rendre disponible pour les questions et les inquiétudes de ses étudiants, et a su fournir encouragement et compassion en réponse aux frustrations quotidiennes de ses pupilles. L'auteur désire le remercier en particulier pour sa patience et son attention au détail dans la revue d'articles soumis par ses étudiants.

Ce travail a aussi été possible grâce au soutien du Conseil de recherche en sciences naturelles et en génie du Canada, à travers la bourse Alexander Graham Bell pour études graduées, sous le numéro 361163.

RÉSUMÉ

Contexte : Le développement logiciel est un travail d'équipe manipulant un produit essentiellement invisible. En conséquent, le développement logiciel nécessite des échanges de connaissances importants entre développeurs afin que l'équipe effectue une résolution de problème adéquate. Cette résolution de problème résulte en une prise de décision qui aura un impact direct sur la qualité du produit logiciel final. **Objectif** : Ce travail doctoral a pour objectif de mieux comprendre ces interactions entre développeurs et comment ces interactions peuvent être liées à des problèmes de qualité logicielle. Cette meilleure compréhension du phénomène permet d'améliorer les approches actuelles de développement logiciel afin d'assurer une meilleure qualité du produit final. **Méthodologie** : Premièrement, des revues de littérature ont été effectuées afin de mieux comprendre l'état actuel de la recherche en gestion de connaissance dans le génie logiciel. Deuxièmement, des analyses de code source et des discussions avec les développeurs ont été faites afin de mieux cerner les causes de problèmes classiques de qualité logicielle. Finalement, des observations faites dans l'industrie ont permis de comprendre la prise de décision collective, et comment cette prise de décision impacte la qualité logicielle. **Résultats** : Les observations effectuées ont démontré que la qualité logicielle n'est pas qu'un problème d'éducation ; l'essentiel des problèmes de qualité ont été introduits par les développeurs en toute connaissance de cause afin de répondre à d'autres impératifs plus urgents au moment de la prise de décision. Améliorer la qualité des logiciels demande de revoir la manière dont les projets de développement logiciel sont gérés afin d'assurer que les décisions prises sur le terrain n'auront pas de conséquences négatives trop coûteuses à long terme. **Conclusions** : Il est recommandé que les organisations se dote d'un nouveau palier décisionnel faisant la jointure entre besoins techniques (i.e. qualité logicielle) et administratifs (i.e. ressources disponibles). Ce nouveau palier décisionnel se situerait au niveau de la base de code (« codebase »), soit entre le palier organisationnel et le palier de gestion de projet. Une base de code étant modifiée de manière concurrente par plusieurs projets en parallèle, il devient nécessaire d'avoir un meilleur contrôle sur les modifications effectuées sur celle-ci. Ce nouveau palier serait le gardien des connaissances en lien avec la base de code, selon le principe « you build it, you run it » favorisé dans certaines organisations. Ce nouveau palier serait responsable d'assurer que la base de code reste d'une qualité suffisamment bonne pour supporter les activités de l'organisation dans l'avenir.

ABSTRACT

Context: Software development is a process requiring teamwork on an essentially invisible product. Therefore, software development requires important knowledge exchanges between developers in order to ensure a proper problem resolution. This problem resolution affects the decision making process, which will have a direct impact on the software quality of the final product. **Objective**: This thesis work aims to better understand these interactions between developers and how they can be linked to software quality problems. With a better understanding of the relation, it will be possible to improve the current software development management practices in order to ensure a better quality of the final software product. **Method**: First, literature reviews were made with the objective to understand the current state of the research in knowledge management in software engineering. Second, source code analyzes and discussions with the developers were executed in order to better understand the causes of typical software quality issues. Finally, observations were made in an industrial context in order to observe collective decision making in the field, and to understand how these decisions impacts software quality. **Results**: The observations made demonstrated that software quality is not only an educational problem; most of the quality problems found were introduced voluntarily by the developers in order to answer a more urgent requirement at the time. Improving software quality therefore requires a review of how software development projects are managed in order to ensure that the decision made in the field do not have overly costly consequences in the long term. **Conclusions**: It is recommended that organization assign a new decision level linking the technical requirements (i.e. software quality) with administrative requirements (i.e. available resources). This new decision level would be situated at the codebase level, between the organizational strategy level and the project management level. A codebase being modified concurrently by multiple projects, it is therefore necessary to have a better control of the modifications made on it. The people at this new decision level would be the knowledge repository related to the codebase, under the "you build it, you run it" principle popular in some organizations. This new decision level would be responsible of ensuring that the codebase remains of a sufficient quality in order to support the future activities of the organization.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

LISTE DES SIGLES ET ABRÉVIATIONS

APA	American Psychological Association
ISO	International Standard Organization
ROI	Return on Investment
SPI	Software Process Improvement
SPEM	Software & Systems Process Engineering Metamodel

LISTE DES ANNEXES

CHAPITRE 1 INTRODUCTION

Le travail en génie est essentiellement un travail en équipe (Flanagan et al., 2007), notamment parce qu'il n'est pas facile de répondre à la complexité des besoins des utilisateurs en travaillant seul. La citation suivante illustre cette observation pour le développement logiciel pour des fins scientifique.

> As scientists' needs for computational techniques and tools grow, they cease to be supportable by software developed in isolation. (Turk, 2013)

Le travail en équipe implique des échanges entre coéquipiers, un transfert de connaissances d'une personne à l'autre. Pour le génie traditionnel, ce transfert se fait à travers les échanges entre parties prenantes, mais aussi à travers les produits, prototypes et ébauches concrètes. La difficulté en génie logiciel provient du fait que les produits construits sont essentiellement invisibles (Brooks, 1987). La seule représentation complète du produit logiciel se trouve dans la tête des parties prenantes, représentation mentale qui peut être incohérente, incomplète ou erronée d'une personne à l'autre (Kruchten, 2010).

Assurer que ces représentations mentales sont cohérentes entre toutes les parties prenantes est donc un élément important pour le succès d'un projet de développement logiciel. En conséquent, il s'agit d'un domaine qui a bénéficié d'une somme de publications importantes dans les dernières décennies (Dingsøyr et al., 2009). La gestion de connaissance englobe présentement une vaste gamme de pratiques spécifiques introduites dans les processus de développement logiciel d'une organisation. Notons par exemple (Bjørnson and Dingsøyr, 2008) :

- Maintenir un entrepôt de données dans un gestionnaire de contenu (ex. : Sharepoint),
- Identifier les flots de connaissances importants au sein de l'organisation,
- Analyser les problèmes liés aux connaissances dans le cadre d'une revue de projet post-mortem,
- Observer les interactions entre les individus impliqués dans un projet,
- Etc.

1.1 Définition de la gestion de connaissance

La gestion de la connaissance (« knowledge management ») est définie théoriquement comme :

> A method that simplifies the process of sharing, distributing, creating, capturing, and understanding a company's knowledge. (Dingsøyr et al., 2009)

De manière concrète, l'objectif de la gestion de connaissance est de tirer profit des connaissances dans la tête de chacune des parties prenantes, que ce soit les développeurs, les clients, les testeurs, etc. Par tirer profit, on entend minimiser le travail inutile des développeurs logiciels. Par exemple :

— Écrire du code qu'il faudra jeter parce que les besoins ont été mal compris,
— Concevoir une architecture inappropriée parce que les contraintes ont été mal documentées,
— Exécuter les tests sur le mauvais module parce que les modifications n'ont pas été transmises,
— Faire construire un module deux fois parce que le travail n'a pas été bien assigné,
— Etc.

Malheureusement, les pratiques testées jusqu'à présent ont eu des résultats parfois décevants, notamment en produisant des artéfacts (Conradi and Dingsøyr, 2000) ou bien des activités (Dingsøyr et al., 2009; Lehtinen et al., 2014; Lavallée and Robillard, 2012) souvent incompatibles avec la réalité du développement logiciel.

> Companies developing information systems have failed to learn effective means for problem solving to such an extent that they have learned to fail. (Lyytinen and Robey, 1999), cité par Bjørnson and Dingsøyr (2008)

La gestion de connaissance, afin d'apporter réellement une plus-value au développement logiciel, se doit d'être cohérente avec les besoins en connaissances de l'équipe de développement logiciel. Il n'est pas suffisant d'accumuler la connaissance, il faut aussi comprendre à quoi peut-elle servir, en quoi elle influence la résolution de problèmes.

1.2 La résolution de problèmes en génie logiciel

L'objectif de la gestion de connaissance est d'améliorer la résolution de problèmes des équipes d'ingénierie. L'équipe d'ingénierie qui possède les connaissances appropriées, au moment approprié, dans le format approprié, peut prendre de meilleures décisions et ainsi éviter le travail inutile.

Le livre de Reed (2007, pages 326-331) catégorise les problèmes à résoudre en trois catégories, dépendant du niveau de connaissances nécessaires pour les résoudre.

— Les problèmes d'agencement : Réorganiser des éléments afin d'obtenir un agencement répondant aux critères demandés. Demande des connaissances sur les agencements possibles. Par exemple, un développeur travaillant sur la conception architecturale

pourrait bénéficier d'une liste de patrons de conception (Gamma et al., 1995).

— Les problèmes d'induction de structure : Découvrir la relation existante entre les éléments. Demande des connaissances sur des relations fréquentes. Par exemple, une tâche de débogage pourrait être facilitée si le développeur connait des causes de bogues communs.

— Les problèmes de transformation : Rechercher comment atteindre un état désiré à partir d'une situation de départ. Demande une expertise sur le travail à faire. Par exemple, construire un logiciel à partir d'une liste d'exigences peut être facilité si un expert ayant déjà construit des exigences similaires est présent.

Il y a donc un besoin de mieux comprendre comment les équipes de développement travaillent afin de déterminer comment les décisions sont prises. Comprendre ces approches de prise de décision nous permettra par la suite de déterminer plus clairement les besoins en connaissances. Répondre plus adéquatement aux besoins en connaissances permettra de prendre de meilleures décisions, ce qui devrait prévenir le travail inutile.

Par exemple, une mauvaise décision pourrait être d'abandonner le modèle d'architecture MVC (modèle-vue-contrôleur) en cours de projet parce que les développeurs ne savent pas comment l'implémenter adéquatement (Lavallée and Robillard, 2011). Il s'ensuit un produit logiciel avec une architecture déficiente, qui aura dès son lancement une dette technique (McConnell, 2013) importante.

1.3 Approche de recherche

Une des limitations actuelles dans le domaine de la gestion de connaissances est la quantité d'observations prises sur le terrain présentement disponibles.

> As the field of software engineering matures, there is an increased demand for empirically-validated results and not just the testing of technology, which seems to have dominated the field so far. (Bjørnson and Dingsøyr, 2008)

L'approche choisie a donc été une approche d'observation non-participative sur un mode exploratoire, l'objectif étant d'obtenir une meilleure compréhension de ce qui se passe réellement sur le terrain. Le domaine de recherche sur la prise de décision collective en génie logiciel est encore trop peu cartographié pour permettre la formulation d'hypothèses solides.

1.4 Objectif de la recherche

La motivation derrière la recherche sur la résolution de problème et la prise de décision en génie logiciel se base sur le fait que les problèmes de qualité logicielle ne sont pas toujours liés à une ignorance des bonnes pratiques (Lavallée and Robillard, 2011). En fait, certains problèmes de qualité logicielle sont introduit sciemment par les développeurs, en toute connaissance de cause, à cause de facteurs en-dehors de leur contrôle (Lavallée and Robillard, 2015).

L'objectif général de ce projet de recherche est de mieux comprendre comment les décisions sont prises durant le développement logiciel, avec un focus particulier sur les causes derrière les mauvaises décisions prises. Par mauvaise décision, on entend les décisions créant du travail inutile en créant une dette technique, c'est-à-dire en introduisant des problèmes de qualité logicielle. Identifier les causes derrière ces mauvaises décisions permettra d'identifier des solutions appropriées assurant que les connaissances nécessaires soient disponibles au moment propice.

Les objectifs spécifiques sont donc :

1. Évaluer le lien entre des problèmes de qualité logiciel et la prise de décision collective,
2. Identifier d'où proviennent les influences sur la prise de décision dans une équipe de développement logiciel,
3. Proposer des améliorations permettant d'éviter certains problèmes de qualité logicielle récurrents.

1.5 Structure de la thèse

Le chapitre suivant présente une revue de la littérature dans le domaine de la gestion de connaissance, et comment les publications précédentes expliquent le travail en équipe. Le chapitre 3 présente des approches méthodologiques utilisées dans la recherche en génie logiciel. Les approches sélectionnées pour ce projet de recherche sont décrites et justifiées. Le chapitre 4 présente une synthèse des publications faites dans le cadre de ce projet de recherche et comment ces publications s'articulent ensemble. Le chapitre 5 présente une discussion des publications et présente des recommandations pour les futurs projets de développement logiciel. Le chapitre 6 résume les conclusions de ce projet de recherche, de même que les avenues futures de recherche dans le domaine. Finalement, les textes complets des articles publiés se retrouvent dans les annexes. L'annexe A présente brièvement une liste des articles en annexe avec leur contribution au travail de recherche.

CHAPITRE 2 REVUE DE LITTÉRATURE

Le logiciel évolue de manière opportuniste, en fonction des besoins immédiats dictés par le contexte (Robillard et al., 2014a). En conséquent, les décisions sont souvent prises dans l'impulsion du moment, avec les connaissances disponibles sur le moment.

> Designers actually follow well-structured plans as long as they find nothing better to do. When knowledge is not readily available, some explanatory mechanisms are required. The explanatory process is called "opportunistic", because at various points in the process the designer makes a decision or takes action depending on the opportunities presented. (Robillard, 1999)

La raison est que le développement logiciel travaille avec des problèmes dit complexes (« wicked problems »), c'est-à-dire des problèmes sans définition claire et sans solution optimale (Waddington and Lardieri, 2006). L'approche idéale serait de faire une étude détaillée du problème et une recherche exhaustive des solutions.

> In rational decision-making it is assumed that when people have complete information about a problem, they can identify all possible solutions, and then chose the one that will maximize the outcome of their efforts. (Moe et al., 2012)

Dans le développement logiciel commercial cependant, il est important de respecter les échéanciers imposés (Luthiger, 2005). Les ressource nécessaires pour faire une analyse exhaustive sont donc rarement disponibles. La prise de décision se fait donc sur la base de l'expérience professionnelle des participants et des informations disponibles.

> Naturalistic decision-making shifted the concept of human decision-making from a domain independent general approach to a knowledge-based approach exemplified by decision makers with substantial experience. The decision-making process was expanded to include the stage of perception and recognition of situations as well as generation of appropriate responses, not just a choice from given options. (Moe et al., 2012)

En pratique, la prise de décision ne privilégie pas l'approche naturalistique plus que l'approche rationnelle, mais suit généralement une approche mixte (Zannier et al., 2007; Moe et al., 2012) dépendant de la nature du problème. Un problème mieux défini favorisera une approche plus rationnelle, alors qu'un problème plus complexe (« wicked ») favorisera une approche basée sur l'expérience (Moe et al., 2012). Il y a donc un besoin de s'assurer que la prise de décision

bénéficie des connaissances appropriées au moment nécessaire. Comment peut-on s'assurer que la prise de décision de l'équipe de développement logiciel s'effectue dans des conditions idéales ?

2.1 Littérature sur la gestion de connaissance

Il faut commencer par avoir une meilleure compréhension de la gestion de connaissance en génie logiciel. Comment la gestion de connaissance est-elle définie et structurée dans le domaine ? L'article de référence en génie logiciel est la revue systématique de littérature effectuée par Bjørnson and Dingsøyr (2008). Leur revue a présenté une vue horizontale de la gestion de la connaissance, répertoriant les articles selon la structure suivante :

- École technocratique,
  - Systèmes : codification de la connaissances dans des bases de données (26 articles),
  - Cartographique : accessibilité des experts dans l'organisation (1 article),
  - Ingénierie : processus de gestion de connaissances (21 articles).
- École comportementale,
  - Organisationnelle : réseaux organisationnels de partage des connaissances (5 articles),
  - Stratégique : approches de gestion de connaissances destinées à donner un avantage compétitif face aux autres organisations (12 articles).
- Gestion de connaissance en général : autres études de gestion de connaissance (4 articles).

Afin de compléter cette revue de Bjørnson and Dingsøyr (2008), les sections suivantes vont présenter le domaine suivant une approche verticale, soit de la gestion de connaissance individuelle jusqu'à la gestion de connaissance organisationnelle. Une brève revue verticale a déjà été élaborée par Rus et al. (2001), mais se limitait aux niveaux individuels et organisationnels. Cette revue ajoute les niveaux de l'équipe et du projet, de même que plusieurs articles pertinents en psychologie cognitive

2.1.1 Gestion de connaissance individuelle

La problématique de la représentation de la connaissance a été initialement étudiée en philosophie. Les philosophes ont développé plusieurs modèles afin de représenter la manière dont un individu emmagasine, catalogue et récupère ses connaissances (Morton, 2003). D'un point de vue pratique, la recherche empirique tente de confirmer ou de réfuter quels modèles philosophiques expliquent le plus de phénomènes réels.

La connaissance est parfois aussi présentée de manière hiérarchique (Rowley, 2007), où les données brutes (niveau 1) sont agencées pour former des informations (niveau 2), qui sont ingégrées par les personnes sous forme de connaissances (niveau 3). Selon cette définition, la connaissance est unique à une personne et dépend de ses modèles mentaux internes. Il n'est donc pas possible, biologiquement, d'échanger de la connaissance, seulement de l'information et des données. Conséquemment, dans le contexte de cette thèse, les mentions d'échange de connaissance réfèrent à l'échange d'information, soit la forme que prend la connaissance lorsqu'elle est verbalisée. Cette approche est justifiée par le fait que la littérature actuelle en génie logiciel parle essentiellement de gestion de connaissance (Bjørnson and Dingsøyr, 2008), et que les niveaux du modèle hiérarchique sont encore sujet à controverse.

S'il n'y a pas de consensus sur la modélisation de la connaissance (Robillard, 1999), il est cependant possible de se baser sur certains faits physiologiques :

— L'être humain possède une mémoire à long terme d'une capacité pratiquement illimitée mais pour laquelle la récupération est souvent imparfaite (Atkinson and Shiffrin, 1968) ;
— L'être humain possède une mémoire de travail à court terme qui ne peut manipuler qu'une quantité limitée d'information (Miller, 1956).

Les pratiques visant à améliorer la gestion de connaissance individuelle en génie logiciel doit donc viser à améliorer ces limites physiologiques. C'est ce que font des outils comme l'autocomplétion (ex. : Intellisence de Visual Studio) dans les environnements de développement intégrés (« Integrated Development Environment » ou IDE). Ces outils facilitent la récupération dans la mémoire à long terme en donnant des suggestions de noms de méthodes et de variables (Ye, 2006). De même, les compilateurs modernes indiquent les codes ayant causé une erreur de compilation, permettant au développeur de trouver plus facilement le problème. Le développeur aurait effectivement de la difficulté à mémoriser les millions de lignes de code d'un logiciel moyen (Walenstein, 2003) !

Cette approche d'étude de la gestion de connaissance de l'individu au sein de son environnement s'appelle cognition distribuée (Salembier, 1995; Lavallée et al., 2013). Ce concept reflète le fait que l'individu ne résout pas des problèmes grâce à son seul esprit, mais à travers l'aide d'outils extérieurs. Et en génie logiciel, les outils extérieurs parmi les plus populaires sont les collègues de travail. L'individu ne résout pas ses problèmes seuls, mais grâce à l'aide de ses collègues, sa communauté de pratique (Wenger, 1999), cité par Bjørnson and Dingsøyr (2008).

En logiciel, la qualité du travail individuel peut être évaluée grâce aux outils actuels (ex. : SVN blame), qui permettent d'identifier qui a écrit chaque ligne de code, et de là arriver à

comprendre pourquoi ce code a été écrit. Mais même dans ces cas, dans un contexte de travail collaboratif, un code écrit en solitaire déficient est souvent le résultat d'une décision collective déficiente. Une personne ou une équipe peut aussi se retrouver à prendre des décisions sur la base de connaissances incomplètes, résultant en des décisions sub-optimale. Madni (2010) présente un exemple pertinent basé sur l'observation d'une équipe d'opérateurs dans le cadre de la défense américaine.

> Too often operators and crews take the blame after a major failure, when in fact the most serious errors took place long before and were the fault of designers or managers whose system would need superhuman performance from mere mortals when things went wrong. (Madni, 2010)

L'auteur reproche que l'essentiel de la recherche se limite à l'observation d'un individu, alors que les défis actuels sont plus liés au travail collaboratif (Madni, 2010). Il devient donc intéressant de comprendre comment une équipe peut prendre collectivement une décision ayant un impact négatif sur la qualité logicielle.

2.1.2 Gestion de connaissance d'équipe

L'étude du travail en équipe a un impact aussi important sur la performance que l'étude du travail individuel (Salas et al., 2008). Il est donc important de s'y attarder. Les modèles en psychologie cognitive définissent le travail en équipe sur la base des sept composantes suivantes (Dickinson and McIntyre, 1997).

- Communication entre les individus ;
- Cohésion de l'équipe et motivation envers les collègues et le travail à faire ;
- Leadership ;
- Compétences des individus ;
- Rétroactions et correction sur la base de la performance d'équipe observée ;
- Tendance à aider ou demander l'aide d'un collègue lorsque possible et nécessaire ;
- Coordination du travail individuel sur la base de ce que font les collègues.

La connaissance imprègne chacune de ces composantes, que ce soit à travers les communications transmises, les connaissances qu'un individu peut avoir, ou à travers le détail de l'avancement des collègues.

L'objectif des interactions au sein de l'équipe peuvent être définies en trois catégories. Les catégories et exemples suivants sont adaptés des travaux de Espinosa et al. (2007).

- Objectif technique. Ex. : problèmes durant l'intégration causé par l'incompatibilité des interfaces.

— Objectif temporel. Ex. : retard dans la production d'un module de code affectant le travail d'autres coéquipiers.
— Objectif lié au processus. Ex. : travail fait sans respecter le processus défini, ou processus incompatible avec les besoins de l'équipe.

Les travaux de DeChurch and Memser-Magnus (2010) montrent une différence de performance selon que la connaissance des participants est homogène (« compositional emergence ») ou spécialisée dans un domaine (« compilational emergence »). La performance de l'équipe est meilleure lorsque les participants sont spécialisés (DeChurch and Memser-Magnus, 2010, page 43). En logiciel, cette spécialisation doit cependant être modérée par le besoin de perennité de l'organisation, qui doit s'assurer que des connaissances clés ne sont pas monopolisées par une seule personne qui pourrait quitter l'organisation à tout moment (Williams and Kessler, 2003; Avelino et al., 2015).

Cette homogénéité/différenciation des connaissances au sein de l'équipe définit ce que la psychologie cognitive appelle le référentiel commun (« common ground ») (Carroll et al., 2006), ou le besoin d'un consensus modéré (Chiocchio et al., 2012). Il est important que l'équipe partage un modèle mental commun du travail fait et du travail à faire, particulièrement en logiciel où une grande partie de ce travail est invisible (Brooks, 1987). En psychologie cognitive, la conclusion obtenue est que les meilleures équipes ont un modèle mental commun plus élaboré, couvrant plus de concepts, alors que les équipes moins performantes ont un modèle plus limité, impliquant moins de concepts (Carley, 1997). Dans tous les cas, la littérature s'entend pour dire que la cognition partagée de l'équipe est un facteur important affectant la performance de l'équipe (Salas et al., 2008).

Du côté du génie logiciel, les modèles actuels de gestion de connaissance se concentrent en partie sur les connaissances relatives à une équipe (Bjørnson and Dingsøyr, 2008; Kerzazi et al., 2013). Ces outils permettent d'identifier des lacunes et donc d'introduire des pratiques pertinentes permettant d'améliorer l'échange de connaissances au sein de l'équipe. Par exemple, un manque de transfert de connaissances entre un analyste et un développeur pourrait être comblé grâce à des séances de programmation en paires (Williams and Kessler, 2003).

Ce point de vue d'équipe peut par la suite être transcendé à l'ensemble des équipes impliquées de près ou de loin dans le projet. Par exemple, si une équipe doit demander à une autre équipe de monter un nouvel environnement de développement, sait-elle qui contacter dans l'organisation (Looney and Nissen, 2007; Lavallée and Robillard, 2015) ?

2.1.3 Gestion de connaissance de projet

Dans un contexte où une équipe travaille seule sur un projet, le niveau de l'équipe et de la gestion de projet sont confondus. Dans la réalité, cependant, les équipes travaillent rarement seules. L'équipe de développement doit interagir avec l'équipe de tests, avec l'équipe de déploiement, avec l'équipe du client, etc. (Lavallée and Robillard, soumis en octobre 2016)

Il y aurait donc un intérêt à assurer une communication optimale entre chaque équipe impliquée. Une meilleure communication impliquerait une meilleure compréhension des problèmes de l'autre, et de meilleures connaissances partagées entre les différentes équipes. Le cloisonnement entre départements semble en effet causer des problèmes lorsqu'une équipe dépend du travail d'une autre dans un autre département (Lavallée and Robillard, 2015). À ce stade, la position du gestionnaire de projet est cependant de protéger son équipe de développement des interactions provenant de l'extérieur, plutôt que de travailler en collaboration :

> Protecting the team from outside distractions was thought to be high of importance in the team. Sales department, customers, upper management or people from other projects constantly interrupts the team to get support from them. One respondent said that the team should be protected by the leader from outside distractions to keep them focused on their project tasks. (Kozak, 2013, page 45)

Or, il semble que ces interactions apportent souvent des informations importantes pour l'équipe de développement (Lavallée and Robillard, soumis en octobre 2016). Demander que ces interactions soient à sens unique (seulement bénéfiques à l'équipe) pourrait alors mener à une désolidarisation entre les équipes, et donc à voir ces équipes travailler en compétition les unes envers les autres.

Un autre obstacle au niveau de la gestion de connaissance de projet est que celle-ci n'est pas encouragée par le contexte technologique actuel.

> Technology's fast pace often discourages software engineers from analyzing the knowledge they gained during the project, believing that sharing the knowledge in the future will not be useful. (Rus and Lindvall, 2002)

L'introduction de principes comme l'organisation en apprentissage (« learning organization ») peut mitiger cette perte de connaissances à la fin d'un projet (Nonaka and Takeuchi, 1995), en construisant des communautés de pratiques liées à un domaine particulier (Rus et al., 2001, pages 13-14). Mais est-ce que cette approche est réellement efficace ? Comme l'écrivait Rajlich (2006) :

> New programmers joining the project must absorb project knowledge and current documentation systems fail to capture this knowledge adequately, making the entrance of the new programmers into an old project difficult. (Rajlich, 2006)

La recherche dans le domaine de la gestion organisationnelle recommande une meilleure gestion de ce transfert de connaissances post-projet, afin d'assurer que celles-ci ne soient pas perdues.

> However, projects are temporal organizations and after their end documents and contact persons are hard to access. Therefore, the acquired knowledge and experiences have to be identified, prepared and distributed through specific actions to bridge the boundaries between one project and other projects or the firm's permanent organization. (Disterer, 2002)

2.1.4 Gestion de connaissance organisationnelle

Le livre de Nonaka et Takeuchi, the Knowledge-Creating Company, décrit comment la connaissance suit tout le parcours décrit dans les sections précédentes. Il décrit les connaissances créées dans la tête d'un individu (implicites) vers des connaissances concrètes appliquées (explicites) au sein de toute une organisation. La position des auteurs est que le rôle de l'organisation est de fournir un contexte permettant de faciliter la création et la distribution de la connaissance (Nonaka and Takeuchi, 1995, pages 73-74).

En conséquence, plusieurs articles étudient les flots de connaissances au sein d'une organisation, que ce soit dans un objectif de compréhension de phénomènes (Lavallée and Robillard, soumis en octobre 2016), dans un objectif d'utilisation optimale des ressources (Hansen and Kautz, 2004), ou bien dans l'identification des réseaux de connaissances (Looney and Nissen, 2007).

> The problem is, many otherwise knowledgeable people and organizations are not fully aware of their knowledge network. (Looney and Nissen, 2007)

Au niveau organisationnel, la difficulté est de garder une vue d'ensemble cohérente sur les connaissances provenant de tous les projets, toutes les équipes et tous les individus. Un projet, une équipe ou un individu peuvent penser que l'organisation manque de connaissances dans un domaine particulier, alors que cette connaissance est présente mais inaccessible. La métacognition, les connaissances que nous avons sur nos propres connaissances, devient difficile au niveau organisationnel (Looney and Nissen, 2007). Il n'est donc pas suffisant que les connaissances soient disponibles, car il y a une différence entre connaissances disponibles

et connaissances utilisées lors de la prise de décision. La connaissance peut être présente, mais si l'organisation ne sait pas qu'elle existe, ou si elle ne prend pas le temps de la rechercher (Zannier and Maurer, 2006; Looney and Nissen, 2007), alors cette connaissance ne sera pas utilisée.

2.2 Litérature sur la résolution de problèmes

Cognition et prise de décision marchent main dans la main. Plus le côté cognitif de la prise de décision est important, plus la décision prise a un impact majeur sur le développement logiciel en cours.

> In general we found that the larger the decision, the more the decision maker considered options. (Zannier and Maurer, 2006)

Prise de décision, résolution de problèmes et cognition sont des concepts importants pour le travail collaboratif ; les travaux de DeChurch and Memser-Magnus (2010) démontrent que ces concepts expliquent 6% de la variance dans la performance d'une équipe, tandis que la motivation et la manière de travailler en équipe en expliquent 12% (DeChurch and Memser-Magnus, 2010, table 4).

Les travaux de Howard et al. (2008) présentent le design logiciel comme un long processus itéré de résolution de problème. La conception du logiciel ne dépend pas d'une seule décision, mais d'un ensemble de décisions résultant d'une suite d'analyses, d'évaluations et de génération de solutions. Leur modèle développé s'apparente à l'UML (Object Management Group, 2015), où le design commence d'une spécification fonctionnelle, pour ensuite être enrichi de ses détails comportementaux, et pour finalement dégager une structure (Howard et al., 2008, pages 166 et 168).

Du côté de la psychologie cognitive, le modèle de Simon and Newell (1971) indique qu'un espace-problème bien défini permet d'arriver à une solution plus optimale. L'analogie qu'ils utilisent est qu'il n'est pas nécessaire de fouiller l'entièreté d'une botte de foin si on sait à peu près où l'aiguille se trouve (Reed, 2007, page 334). Plus l'espace-problème est grand, et plus l'effort requis pour l'explorer sera grand. L'exhaustivité de la recherche d'information dépend de la motivation des participants et du contexte donné (De Dreu et al., 2008). Par exemple, une surcharge de travail va diminuer la motivation, tandis qu'un désir de ne pas laisser tomber ses coéquipiers va l'augmenter.

Reed (2007) définit un certain nombre d'éléments d'informations qui permettent de mieux définir l'espace-problème.

1. Les instructions relatives à la tâche qui donne une description du problème et peuvent contenir des informations utiles.
2. Les expériences antérieures concernant la même tâche ou une tâche presque identique.
3. Les expériences antérieures concernant des tâches analogues.
4. Les plans stockés dans la mémoire à long terme généralisables à un ensemble de tâches.
5. Les informations accumulées durant la résolution du problème.

L'espace-problème peut être mieux défini si celui-ci est analysé par un expert (Robillard, 1999). En conséquence, en génie logiciel, la taille de l'espace-problème peut être diminuée à travers la gestion de connaissances. Par exemple, un expert pourrait éliminer d'emblée des solutions qu'il sait irréalisable dans le contexte. Un autre exemple serait la documentation disponible lors du débogage d'une application existante. Une documentation bien montée permet au débogueur d'avoir une meilleure idée où chercher pour trouver la solution au bogue, ce qui limite l'espace-problème. Ultimement, la limite est la capacité humaine à manipuler l'information.

> It was speculated that the code decay is characterized by the situation where complexity of the code outruns the cognitive capabilities of the programmers. (Rajlich, 2006)

2.2.1 Portée de la recherche

Il s'ensuit que ce projet s'insère dans un domaine en émergence peu étudié jusqu'à présent, tel que présenté dans la Figure 1. Une base de code (« codebase ») est définie comme un ensemble d'applications interdépendantes au sein d'une organisation, comme un progiciel de gestion intégré (« enterprise resource planning », ou ERP). Cette base de code peut être affectée par un ou plusieurs projets de maintenance ou de développement ; chaque projet pouvant impliquer une ou plusieurs équipes.

Ce domaine particulier semble de plus en plus important afin d'assurer la pérennité de l'organisation et d'éviter l'accumulation d'une dette technique (McConnell, 2013) qui serait insurmontable. Un exemple d'accumulation de dette technique au sein d'une base de code serait Netscape et son fureteur Netscape Navigator, dont la base de code est devenue si complexe et de si mauvaise qualité qu'elle a contribué aux problèmes de l'organisation (Wired, 1999).

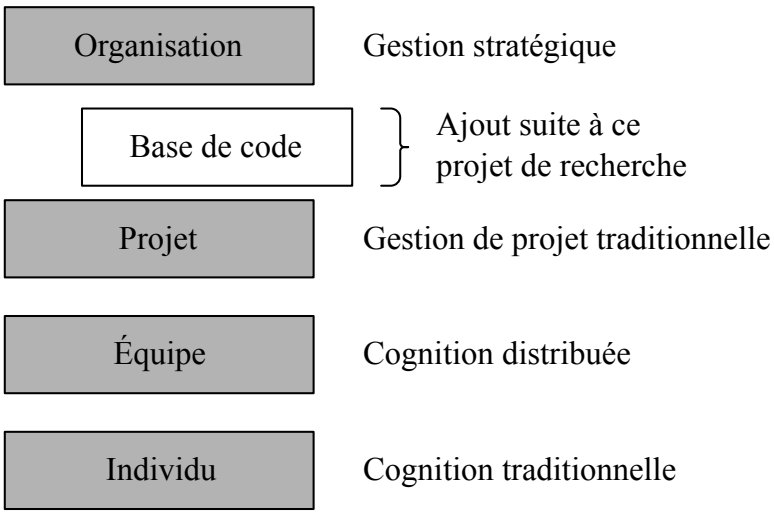

Figure 2.1 Portée de la recherche

2.3 Contexte scientifique

La littérature distingue deux types de tâches Motowidlo and Scotter (1994) et donc deux types de qualité du travail fait, soit celle basée sur la performance de tâche et celle basée sur la performance contextuelle.

— Performance de tâche : Évalue les compétences techniques de l'individu qui effectue le travail.
— Performance contextuelle : Évalue les compétences sociales de l'individu au sein de son organisation.

Cette distinction se reflète aussi dans les travaux de Marks et al. (2001), qui distinguent le travail sur la tâche (« taskwork ») du travail sur l'organisation de l'équipe (« teamwork »). Ce projet de recherche suit donc cette distinction et différencie les concepts de connaissances techniques (liées à la tâche) des connaissances administratives (liées au contexte, à l'organisation du travail) :

— Connaissances liées à la tâche (techniques) : Connaissances liées aux activités en cours dans le cadre d'un projet particulier et ayant un impact sur la manière dont le projet sera mené à sa complétion. Les connaissances opérationnelles sont immédiatement utiles pour la gestion quotidienne du projet de développement logiciel.
— Connaissances liées au contexte (administratives) : Connaissances accumulées sur les projets, les problèmes vécus, les solutions apportées, etc. ayant un impact sur la direction prise par l'organisation dans ses projets futurs. Les connaissances liées au contexte sont utiles pour les décisions affectant le long terme.

La recherche actuelle se concentre surtout sur des approches administratives au détriment des approches techniques (Marks et al., 2001). La recherche se concentre sur un modèle statique de la manière de travailler dans le but de l'améliorer dans de futurs projets, alors que ce projet de recherche suppose la présence d'aspects dynamiques en constants changements.

2.3.1 Connaissances techniques

Les connaissances techniques sont liées aux pratiques techniques (ex. : comment programmer en C++, où se trouvent les fichiers dans le dépôt SVN, etc.). Ces pratiques techniques ne sont cependant pas faciles à cerner :

> First, they are numerous ; second, they are often implicit within a community of practice and strongly linked one to another ; and third, the main distinction among them is the context in which these practices apply. Moreover, these practices are often dynamically constrained in sequences of actions. (Pomerol et al., 2002)

Mais ce sont ces pratiques techniques qui définissent les besoins en connaissances techniques, qui expliquent à leur tour le pourquoi de certaines décisions prises. Un exemple est le concept de conscience de la tâche (Lavallée et al., 2013). Pour qu'un développeur logiciel fasse un travail approprié, il doit être conscient du travail des autres développeurs qui ont un impact potentiel sur le sien. Si un collègue change la signature d'une fonction utilisée par le développeur, celui-ci doit le savoir afin d'en faire la correction dès que possible. La pratique technique de l'utilisation d'un outil de gestion de configuration (ex. : SVN) permet donc de générer des éléments d'information sous formes de soumissions ("commit") de code. Cependant, est-ce que ces éléments d'informations sont intégrés sous forme de connaissances techniques ou, autrement dit, est-ce que le développeur est capable de voir dans la soumission de son collègue que la signature de la fonction a changé ?

Il est probable que ce changement passera inaperçu (connaissance technique manquante). Et lorsque le gestionnaire demandera au développeur une estimation de la complétion de sa tâche, cette estimation sera erronée (décision incorrecte) car il ne considérera pas les corrections requises par les modifications de son collègue.

En conséquence, une nouvelle pratique de gestion de connaissance semble prendre de l'ampleur au sein de la communauté, soit d'ajouter une nouvelle question aux questions traditionnelles du SCRUM (Mountain Goat Software) que les gestionnaire posent aux développeurs :

1. Quelles tâches avez-vous faites dernièrement ?
2. Quelles tâches prévoyez-vous faire dans un proche avenir ?

3. Y a-t-il des obstacles à l'avancement de votre travail ?
4. Prévoyez-vous mettre des obstacles dans le travail des autres ? (Cohn, 2007)

Cette quatrième question permet donc de résoudre un manque de connaissance administratif ou technique. Le gestionnaire apprend donc du collègue que la signature de la fonction a changé, et en informe conséquemment notre développeur, qui peut faire une estimation plus exacte du travail qui l'attend.

2.3.2 Motivation de la recherche

La recherche est motivée par un aperçu obtenu lors de travaux préliminaires, qui ont démontré que des problèmes de qualité logicielles avaient été introduits sciemment dans un produit logiciel, pour des raisons en-dehors du contrôle des développeurs (Lavallée and Robillard, 2011). Cette observation a souligné que s'il est important d'enseigner et de promouvoir la qualité logicielle, il faut encore que les développeurs sur le terrain puisse mettre en place les bonnes pratiques (« best practices ») transmises.

L'objectif de la recherche est donc de mieux comprendre comment les problèmes de qualité logicielles sont introduits dans les produits actuels. Quelles décisions ont menés à l'introduction de ces problèmes ? Pourquoi ces décisions ont-elles été prises ? Qu'est-ce qui a pesé dans le choix des intervenants d'aller de l'avant avec cette approche problématique plutôt qu'une autre ?

Les mécanismes de prises de décisions durant les projets logiciels, et donc la gestion de connaissances au sein des équipes logicielles, seront donc le focus de ce projet de recherche.

CHAPITRE 3 MÉTHODOLOGIE

3.1 Objectifs de la recherche

La revue de littérature a souligné le lien entre la gestion de connaissance et la résolution de problèmes. L'impact de l'approche de résolution de problèmes sur la qualité logicielle semble cependant avoir été peu étudié jusqu'à maintenant, motivant une recherche plus approfondie dans le domaine.

Le développement logiciel étant essentiellement un travail d'équipe (Flanagan et al., 2007), il a été décidé d'étudier la cognition distribuée (Lavallée et al., 2013), au sein des équipes et des organisations, plutôt que la cognition individuelle. Cette approche est justifiée par l'importance des interactions entre développeurs et son impact sur la performance de l'équipe (DeChurch and Memser-Magnus, 2010).

Un des objectifs est de tirer partie de ce qui a déjà été fait dans d'autres domaines, notamment dans le domaine de la psychologie organisationnelle. La recherche vise à découvrir en quoi le génie logiciel ressemble ou diffère des autres domaines précédemment décrits dans la littérature.

Vu le flou important dans la compréhension du fonctionnement des équipes et des organisations, les approches de recherche choisies favorisent des approches exploratoires à des approches plus traditionnelles basées sur la formulation et les tests d'hypothèses. L'objectif de recherche est de présenter suffisamment de détails sur le fonctionnement des équipes de développement logiciel sur le terrain pour que des recherche subséquentes puissent formuler et tester des hypothèses. Ce projet se base sur l'observation qu'il est important d'avoir une compréhension adéquate du terrain afin de formuler des hypothèses susceptibles de mener à des conclusions solides.

3.2 Portée de la recherche

La gestion de la connaissance au niveau organisationnel se transige selon différents mécanismes. L'un de ces mécanismes est la manière dont une idée provenant d'une personne peut se répandre d'une équipe à l'autre, jusqu'à finir par influencer toute l'organisation (Nonaka and Takeuchi, 1995). Ce mécanisme entre dans les principes de l'amélioration continue des organisations, un domaine connu en génie logiciel sous le terme d'amélioration de processus (« Software Process Improvement » ou SPI). Or, ces améliorations de processus représentent

des avancement ponctuels étalés sur une durée importante. Les recherches actuelles en génie logiciel indiquent que le SPI prend en moyenne 18 mois avant d'être intégré de manière systématique par les équipes de développement (Lavallée and Robillard, 2012; Basili et al., 2002). Le cas de Matsushita cité par Nonaka and Takeuchi (1995, page 95) présente comment l'organisation a introduit une nouvelle manière de gérer la connaissance dans ses logiciels, introduction qui s'est étirée de 1984 à 1987.

Cependant, ces changements n'affectent pas la qualité du produit en cours de développement. Il s'agit d'améliorations statiques pour les produits futurs, qui sont importants afin d'assurer la pérennité de l'organisation, mais qui n'ont pas d'impacts dynamiques immédiats sur le processus de développement. L'objectif de ce projet de recherche est de mieux comprendre la prise de décision quotidienne des développeurs, ce qui semble important afin de proposer des améliorations éclairées.

Si la portée de la recherche se limite plutôt à la durée de vie d'un projet, il est aussi possible de limiter la portée de la recherche aux personnes qui interagissent directement avec le projet. En limitant la portée de la recherche seulement aux activités ayant un impact direct sur le projet, il est possible d'écarter beaucoup de gens dans l'organisation qui n'ont que des impacts à long terme sur le processus de développement logiciel. Et cela permet de centrer l'observation sur les connaissances qui affectent directement la qualité technique du produit, plutôt que la direction stratégique et administrative de l'organisation.

3.3 Approches méthodologiques possibles

De manière sommaire, il est possible de séparer le processus scientifique en trois étapes, soit la formulation d'hypothèses, la collecte de données, et l'analyse des données obtenues. Cette approche suit la structure de revue systématique de Biolchini et al. (2005) pour les revues systématiques de littérature, correspondant à la planification, l'exécution et l'analyse des résultats. En conséquent, différentes approches ont été développées pour chacune de ces étapes. Les sections suivantes présentent différentes approches, et en quoi celles-ci sont pertinentes dans le cadre de cette recherche.

3.3.1 Approches théoriques

Les sciences naturelles favorisent la création d'hypothèses, qui peuvent par la suite être testées et mener à la construction de théories. Cette approche fonctionne bien dans un contexte relativement bien connu, où les variables susceptibles d'influencer le phénomène observé sont suffisamment définies.

L'étude du développement logiciel s'apparente cependant beaucoup plus aux sciences sociales, notamment la psychologie et la sociologie. Les sciences sociales étudient des problèmes plus complexes, dans le sens que le phénomène à observer est difficile à évaluer, et que les variables affectant ce phénomène sont nombreuses, difficiles à mesurer, et parfois inconnues. Il a déjà été démontré que le développement logiciel est un problème multivariable (Clarke and O'Connor, 2012) complexe (Brooks, 1987). Il en est de même avec la qualité logicielle (International Standard Organization, 2016). S'il est possible de mesurer certaines variables définissant la qualité d'un logiciel, il est difficile de conclure de manière objective qu'un logiciel est de meilleure qualité qu'un autre.

Il peut donc être bénéfique d'étudier d'autres approches théoriques. Une approche souvent utilisée (Kellner et al., 1999) dans la recherche sur le développement logiciel est l'approche par simulation. Des théories divergentes peuvent être confrontées en simulant théoriquement comment ces approches pourraient fonctionner. Les simulations peuvent ensuite servir de guide pour la formulation d'hypothèses pouvant être testées en laboratoire ou validées sur le terrain.

Une autre approche théorique est la revue de littérature. Une revue de littérature permet d'extraire les théories et hypothèses déjà formulées, et de voir l'étendue des données concrètes supportant ou défaisant chacune d'entre elles (Biolchini et al., 2005). La revue de littérature permet aussi de cartographier un domaine, un type d'étude appelé « mapping studies », afin de découvrir quelles sous-section du domaine ont déjà été bien défrichés et quelles sous-sections du domaine auraient besoin de plus d'études. Une recherche future s'articule généralement sur la base de la littérature, soit en combinant ou modifiant des théories existantes, soit en formulant une nouvelle théorie qui comble les défauts des théories existantes.

3.3.2 Approches de collecte de données

Les travaux de Lethbridge et al. (2005) et de Ton-That et al. (2013) ont déjà présenté des techniques de collecte de données utilisées par les chercheurs en génie logiciel. Ces travaux séparent les approches en niveaux, basés sur la proximité entre les données obtenues et ce qui se passe réellement sur le terrain.

Par exemple, la recherche d'artéfacts à analyser est une approche de troisième degré, tel que défini dans le tableau 3.1, car elle se limite à l'analyse de ce qui a été produit. Une telle technique fait donc abstraction de tout le processus cognitif et décisionnel ayant mené à la création de cet artéfact. Les artéfacts produits durant le développement logiciel sont une bonne source de données pour évaluer le résultat final, mais ne donnent pas une vue très claire sur le processus décisionnel sous-jacent.

Tableau 3.1 Approches de collectes de données, adapté de Lethbridge et al. (2005)

Catégorie	Approche
Premier degré inquisiteur	Remues-méninges, questionnaires, interviews, etc.
Premier degré observationnel	Journaux de travail, observation participative et non-participative, etc.
Second degré	Instrumentation, saisies d'écran, etc.
Troisième degré	Collection d'artéfacts produits durant le développement, analyse statique et dynamique du code, etc.

Par contraste, l'observation de ce qui se passe sur le terrain est beaucoup plus directe. L'observation permet de saisir beaucoup de détails dans les discussions qui n'apparaitront pas dans les artéfacts produits. Il s'agit donc d'une approche de premier degré.

3.3.3 Approches d'analyse de données

Une fois les données amassées, un défi est de déterminer à quoi ces données peuvent servir. La recherche scientifique tend à privilégier les approches statistiques, car elles sont considérées comme plus objectives. Les tests statistiques nécessitent cependant des données quantitatives, ou sinon des données qualitatives pouvant être codifiées de manière quantitatives.

En sciences sociales cependant, les données collectées sont souvent de nature qualitative, par exemple sous forme d'un texte narratif décrivant l'observation d'un phénomène particulier. Cependant, comme il peut être difficile de tirer des conclusions répétables à partir d'observations disparates, des approches ont été développées pour l'analyse de données qualitatives.

3.4 Approches méthodologiques choisies

Étant donné les possibilités de collectes de données disponibles lors de ce projet de recherche, il a été décidé de favoriser certaines approches méthodologiques plutôt que d'autres. Ces choix portent à conséquence : chaque approche méthodologique possédant en effet ses forces et faiblesses. Les sections suivantes présentent les approches choisies et les raisons motivant ce choix.

3.4.1 Approche théorique - la revue de littérature

L'approche de revue utilisée est basée sur l'approche de revue de littérature systématique adaptée au génie logiciel par Kitchenham (2004) et Biolchini et al. (2005). L'approche a été adaptée pour être utilisée par des novices dans un sujet donné, dans l'objectif de faire des revues de littérature pour cartographier un domaine et en obtenir une meilleure compréhension (Lavallée et al., 2014). L'approche a permis de découvrir la pertinence de certaines avenues de recherche dans les domaines explorés.

L'inconvénient majeur de cette approche est qu'elle demande un effort important qui n'est pas toujours fructueux. Malgré le niveau de détails fournis dans le processus de revue de littérature systématique, il reste qu'il s'agit d'un processus qualitatif, et donc susceptible à des biais de la part des chercheurs.

3.4.2 Approche théorique - la modélisation théorique

Un des travaux préliminaires à ce projet de recherche a été la construction d'un outil de modélisation de processus de développement logiciel permettant de modéliser certains aspects de gestion de connaissances (Kerzazi et al., 2013). Cet outil a permis la modélisation de processus théoriques de même que des processus décrits dans la littérature.

Le processus modélisé possède un avantage sur les théories traditionnelles écrites de manière textuelle. Le modèle a permis de confirmer les théories de Hansen and Kautz (2004) sur la présence de goulots d'étranglement (« bottlenecks ») dans les processus de développement logiciel, de même que de visualiser des problèmes potentiels dans le standard ISO 14764 décrivant un processus de maintenance (International Standard Organization, 2006).

L'inconvénient est qu'il s'agit d'une approche théorique dont les conclusions peuvent ne pas correspondre à la réalité. Si la simulation est un outil utile pour tester des théories, celles-ci devraient toujours être confrontées à la pratique.

3.4.3 Approche de collecte de données - recherche d'artéfacts

La recherche a bénéficié de l'accès à des artéfacts produits dans le cadre de développements de logiciels, notamment le code source, mais aussi des artéfacts de gestion de projet, de conception architecturale et d'exigences (Lavallée and Robillard, 2011). L'obstacle principal est que de réels artéfacts produits sur le terrain sont souvent considérés comme secret industriel et il peut être difficile d'obtenir le droit d'y accéder (Lavallée and Robillard, 2015). Dans le cas où les chercheurs peuvent accéder à ces artéfacts, il peut quand même être difficile de distribuer

ces artéfacts à d'autres chercheurs afin de répliquer l'analyse produite. Les chercheurs actuels favorisent donc l'analyse d'artéfacts produits dans le cadre de projets à code ouvert (« open source ») parce que ces artéfacts sont accessibles au grand public.

3.4.4 Approche de collecte de données - observation non-participative

La revue systématique de Bjørnson and Dingsøyr (2008) appelle à plus de recherche ethnographique, se concentrant en particulier sur le niveau organisationnel. Leur argument est qu'il n'y a que peu de projet de recherche dans ce domaine actuellement, et qu'obtenir des détails à ce niveau serait moins dispendieux que la conception, réalisation et l'essai d'outils de gestion de connaissance. Dans ce contexte, la recherche ethnographique dans le cadre de ce projet de recherche se base sur les travaux de Brewer (2000).

> Ethnography is the study of people in naturally occurring settings or "fields" by means of methods which capture their social meanings and ordinary activities, involving the researcher participating directly in the setting, if not also the activities, in order to collect data in a systematic manner but without meaning being imposed on them externally. (Brewer, 2000, page 10)

Le choix d'une approche qualitative dans un contexte de génie logiciel s'explique par le besoin d'aborder un problème peu étudié jusqu'à maintenant où les variables importantes n'ont pas encore été identifiées. Pour reprendre les termes de Looney and Nissen (2007) :

> The present research is exploratory in nature, is not guided by extensive theory, and is approaching a "how" research question. Hence qualitative field research reflects an appropriate method. (Looney and Nissen, 2007)

Comme il n'y a pas de consensus théorique sur la manière dont les connaissances transigent en génie logiciel, il est nécessaire d'observer ce qui se passe réellement sur le terrain. Ces observations permettront de dégager les variables importantes reliant gestion de la connaissance, travail en équipe et qualité logicielle afin de monter des hypothèses qui pourront par la suite être vérifiées.

> Ethnographies [...] encourage a skepticism on the part of the analyst for all "received" wisdom and taken for granted theoretical categories which we might feel tempted to impose on what is observed "in the field". Thus, the ethnographer tends not to be tempted into assuming that the most important aspects of human computer interaction must lie at the user interface or in any other single, fixed location for that matter. (Bowers and Rodden, 1993)

L'avantage d'une approche ethnographique comme l'observation non-participative permet donc d'arriver sur le terrain avec le moins d'idée préconçues que possible.

Dans le cadre de cette étude (Lavallée and Robillard, 2015), étant donné qu'il s'agissait d'un premier contact avec une organisation peu ouverte à partager ses données, le maximum qu'il a été possible d'obtenir est l'accès aux réunions de suivi de l'équipe. Considérant le flou entourant le projet, les gestionnaires eux-mêmes n'ayant pas une vision claire de ce qui se passe durant ces réunions, il a été décidé d'utiliser une approche exploratoire.

Un observateur non-participatif a donc assisté aux réunions d'avancement d'un projet logiciel impliquant une douzaine de personnes pendant dix mois. Cet observateur a noté tous les échanges ayant eu lieu entre les personnes présentes durant cette période. Les interactions ont été notées à la main, vu que l'organisation n'était pas prête à ce que des enregistrements soient faits. Les notes ont tenté de saisir des phrases clés dans les discussions, de même que le sujet discuté. Une fois la réunion terminée, l'observateur a pris ses notes et les a immédiatement transcrites sur l'ordinateur, ajoutant les détails manquants aux notes manuscrites alors qu'il avait encore les discussions en tête. Le tableau I présente un exemple, saisi le 21 août 2014.

Dans le cas précis du tableau 3.2, remarquez que les observation permettent de mieux comprendre les interactions entre développeurs et testeurs. Par exemple, les notes rapportent que les développeurs ne regardaient pas adéquatement leur outil de suivi des bogues, ce qui fait que des bogues rapportés par les testeurs étaient inconnus des développeurs.

3.4.5 Approche d'analyse - analyse statique d'artéfacts

L'attrait principal de cette approche est qu'elle est relativement facile une fois que les documents sont entre les mains des chercheurs (Lethbridge et al., 2005). Les artéfacts peuvent présenter une riche somme d'information qu'il est possible de croiser afin de tirer et confirmer des conclusions.

La contrepartie est qu'il s'agit d'une approche relativement éloignée des développeurs eux-mêmes (Lethbridge et al., 2005), et qui ne reflètent qu'une partie de leur réflexion. Il peut être difficile de comprendre les causes derrières les décisions cristallisées dans ces documents. Il est donc souvent nécessaire de confirmer les conclusions tirées de données provenant d'artéfacts à travers une approche plus directe, par exemple grâce à des entrevues.

3.4.6 Approche d'analyse - théorie à base empirique

L'approche de théorie à base empirique (« grounded theory ») est utilisée dans l'analyse de données qualitatives. Cette approche est souvent utilisée en génie logiciel, notamment dans

Tableau 3.2 Exemples d'observations transcrites

Interlocuteurs	Détail des interactions
Gestionnaire1 + Développeur6	Gestionnaire1 demande à Développeur6 : « Y a-t-il des bogues qui trainent dans le module-R ? »
Développeur5 + Testeur2 + Développeur6	Testeur2 décrit une anomalie qui fait que les bons de travail ne remontent pas jusqu'au module-R.
Développeur6	»On a corrigé ce bogue-là.«
Testeur2	Testeur2 explique que oui, la correction a fonctionné. Cependant, quelque chose d'autre a brisé le fonctionnement par la suite et que les bons de travail ne remontent plus.
Développeur6 + Développeur5 + Développeur7	Discussion sur cette anomalie. Développeur 6 et Développeur7 n'étaient pas au courant de cette anomalie.
Gestionnaire1	Gestionnaire demande aux Développeur6 et Développeur7 : « Avez-vous modifié votre outil de suivi de bogues afin d'être avisé quand des bogues arrivent ? »
Développeur4 + Développeur5 + Développeur6	« Oui. »
Gestionnaire1	À toute l'équipe : « Faites des révisions régulières de la liste des bogues et assignez-vous les bogues qui n'ont pas été assignés. »
Développeur9	« OK, gotta be proactive. »

l'étude des processus de développement logiciel (Lavallée and Robillard, 2012; Coleman and O'Connor, 2006, 2007). L'objectif de l'approche est de monter une catégorisation, taxonomie ou ontologie de concepts basée sur les observations faites sur le terrain. Cette catégorisation peut aussi s'accompagner d'une analyse de la fréquence d'apparition de chaque élément identifié.

En résumé, l'approche suivie s'est basée sur le guide de Strauss (2003), adapté aux données obtenues.

1. Codification libre : Chaque point de donnée (observation, évaluation qualitative, phrase transcrite d'une interview, etc.) est associé à un ou plusieurs codes. Les codes sont associés librement, sans modèle préétabli.
2. Re-codification tant que de nouveaux codes émergent : Tant que les chercheurs trouvent

de nouveaux codes à associer aux données, la codification se poursuit.

3. Regroupement des codes : Les codes similaires sont regroupés. Les chercheurs essaient de trouver un ordre sous-jacent aux codes trouvés, hiérarchique, taxonomique ou autre.
4. Comparaison avec la littérature : La structure trouvée est comparée avec des structures existantes dans le domaine décrite par la littérature (catégorisations, taxonomies, ontologies, etc.). La structure de codification est finalisée.
5. Codification finale : La structure de codification finale est utilisée pour faire une dernière passe de codification des données. Cette codification est généralement faite par deux chercheurs travaillant indépendamment, afin d'assurer que les codes sont suffisamment bien décrits pour produire des résultats répétables.
6. Présentation des résultats : Les données codées sont présentées, par exemple sous forme d'une analyse fréquentielle de chaque code, ou bien sous forme d'une conclusion narrative liant les codes importants ensemble.

L'approche est critiquée parce que l'analyse de données qualitative peut sembler subjective. Cependant, lorsque effectuée correctement, l'approche peut donner des résultats répétables (Robillard et al., 2014a).

3.4.7 Approches de validation des résultats

Dans le cadre d'une recherche basée sur des données qualitatives, il est important de confirmer les conclusions obtenues en comparant différentes sources. Dans le cadre de cette recherche, l'approche de confirmation qui a été favorisée a été l'utilisation d'entrevues (Lavallée and Robillard, 2011; Lavallée and Robillard, 2015). Les conclusions ont été présentées aux développeurs sur le terrain afin que celles-ci soient confirmées ou réfutées. Cette approche a permis de découvrir que certaines conclusions sur les causes de problèmes de qualité étaient fausses.

Une autre approche utilisée a été de présenter les résultats à une autre équipe de développement (Lavallée and Robillard, 2015), de même que le suivi des réactions au sein de la communauté logicielle suite à la publication des résultats (Hacker News; Reddit).

3.5 Conclusions méthodologiques

L'article produit dans le cadre de l'atelier (« workshop ») sur la recherche en milieu industriel présente un processus à suivre dans le cadre d'un premier contact avec une organisation dans le but de faire de la recherche qualitative exploratoire (Lavallée and Robillard, 2015).

Le processus présenté se base en partie sur la littérature dans le domaine, en partie sur l'expérience obtenue en faisant des revues de littérature systématiques (Lavallée et al., 2014).

La conclusion principale est que le travail de recherche devrait idéalement être abordé de la même manière que le développement logiciel, c'est-à-dire de manière incrémentale. Il est préférable de monter une méthodologie progressive, permettant d'obtenir des résultats préliminaires ou intermédiaires, afin de permettre de corriger le tir. La recherche ne se limite pas seulement à élaborer et tester des hypothèses, mais nécessite aussi des corrections méthodologiques afin de s'assurer que les résultats obtenus sont significatifs et répétables.

Dans le cas de ce projet de recherche, cette approche a permis de publier des résultats préliminaires (Kerzazi et al., 2013). Ces résultats préliminaires ont permis de corriger les hypothèses initiales, notamment en démontrant que plusieurs problèmes de qualité logicielle sont parfois introduits sciemment par les développeurs (Lavallée and Robillard, 2011). Cette constatation a motivé une étude sur la prise de décision collective (Lavallée and Robillard, 2015) et donc sur les conclusions de ce projet de recherche.

CHAPITRE 4 SYNTHÈSE DES PUBLICATIONS

Ce travail de recherche est le fruit d'une réflexion quinquennale ayant mené à plusieurs publications. La liste complète des publications est présentée à l'annexe A. Les sections suivantes présentent comment toutes ces publications s'enchaînent et permettent d'obtenir une meilleure compréhension de la gestion de la connaissance dans les équipes de génie logiciel, et comment cette gestion de la connaissance peut avoir un impact sur la qualité logicielle.

4.1 Gérer la connaissance à travers un outil

Initialement, la connaissance en génie logicielle a été abordée comme un problème individuel : est-ce que chaque individu impliqué dans le projet reçoit les connaissances nécessaires pour réaliser sa tâche ? Les connaissances étaient donc considérées comme une ressource pour le gestionnaire de projet, au même titre que les briques et les planches pour un projet de construction. Cette façon de voir la gestion de connaissance a été matérialisée sous la forme d'un outil de modélisation (Kerzazi et al., 2013). L'outil fonctionne de la manière suivante :

1. Le gestionnaire modélise le processus de développement logiciel. Ce processus est basé sur le langage SPEM 2.0 (Object Management Group, 2008), et décrit des activités qui transforment des artéfacts (ex. : document de cas d'utilisation) en d'autres artéfacts (ex. : document d'architecture), activités étant assignés à des rôles.
2. Le gestionnaire détermine les connaissances nécessaires à la réalisation de chaque activité, de même que les connaissances fournies par les artéfacts en entrée et par les rôles assignés. Les connaissances nécessaires sont tirées d'une liste générique provenant d'une taxonomie existante (Anquetil et al., 2007).
3. L'outil retourne au gestionnaire une mesure du degré de cohérence entre les connaissances nécessaire pour réaliser l'activité, versus les connaissances fournies à cette activité.

L'outil amène un bénéfice intéressant car il permet d'identifier les activités problématiques du point de vue des connaissances au sein du processus de développement logiciel. Cette identification, initiée de manière théorique par Hansen and Kautz (2004), permet de découvrir des activités qui demandent trop de connaissances (« black holes »), des activités qui doivent générer trop de documentations diverses (« spring »), des activités qui forment des points critiques de connaissances (« bottleneck », « hub »), etc. Cette identification permet

au gestionnaire d'améliorer le processus en redirigeant les connaissances à d'autres activités plus appropriées.

Malheureusement, des tests plus poussés effectués par la suite ont démontré les limites de cet outil :

— Le travail de modélisation et d'assignation des connaissances requises et fournies demande beaucoup d'efforts. Il semble peu probable qu'un gestionnaire de projet prenne le temps de faire ce travail, à moins de démontrer des bénéfices très significatifs.
— L'assignation des connaissances requises et fournies n'est pas répétable. Deux évaluateurs qui font le même travail d'assignation arrivent rarement aux mêmes résultats.
— L'outil présente une analyse statique. Les besoins en connaissances étant dynamiques (Dickinson and McIntyre, 1997), il devient donc nécessaire de refaire l'analyse à chaque fois qu'il y a un changement.
— La connaissance ne peut pas être gérée comme une ressource matérielle. Pour une activité qui nécessite des connaissances en UML, par exemple, il n'est pas suffisant de fournir un livre sur le langage UML. Un processus d'apprentissage est nécessaire. Le temps requis pour faire cet apprentissage et la qualité de ces apprentissages n'est pas facile à évaluer.

Ces constatations confirment d'autres recherches sur les outils de documentation de la connaissance. Ces outils ont souvent tendance à devenir des « cimetières de données », c'est-à-dire un recueil de données souvent obsolètes ou erronées (Conradi and Dingsøyr, 2000). De plus, la littérature actuelle inclut déjà une quantité importante d'outils de gestion de connaissances variés (Bjørnson and Dingsøyr, 2008) qui ne font pas consensus (Lavallée et al., 2013).

Ces constatations indiquent qu'il est nécessaire de pousser la recherche dans une autre direction. Convaincre des gestionnaires d'introduire une nouvelle pratique de développement logiciel ne peut être motivée qu'à travers un retour potentiel sur l'investissement (McConnell, 2013). Serait-il possible de lier la gestion de la connaissance avec des éléments du génie logiciel ayant une incidence monétaire significative ?

4.2 Lier gestion de connaissance et qualité logicielle

Si depuis 1968, la science du génie logiciel s'est beaucoup approfondie, il en demeure qu'une grande quantité du logiciel produit est toujours de mauvaise qualité. L'impact de ces lacunes de qualité a un impact monétaire important (McConnell, 2013). Il serait donc bénéfique de comprendre pourquoi des problèmes de qualité sont introduits dans un logiciel.

Il a donc été décidé d'analyser du code existant contenant des problèmes de qualité et de

trouver les causes racines (« root causes ») expliquant pourquoi ce code a été mal écrit. La qualité reste cependant un terme très vague, et peut inclure autant une incompréhension des besoin du client qu'une faute de frappe dans l'écriture du code causant un bogue. Il a donc été décidé de se concentrer sur les erreurs commises dans le design technique interne du produit, un concept communément appelé odeur de design (« design smell ») (Moha et al., 2010). De manière plus précise, le code a été analysé suivant deux modèles, soit la taxonomie de Mäntylä and Lassenius (2006) et l'Application Architecture Guide de Microsoft (2009).

L'analyse des odeurs de design a été effectuée sur trois projets de développement logiciels. L'analyse a observé la différence entre le design prévu dans les plans initiaux et le design que l'ont peut observer dans le code source. Les trois projets étaient (Lavallée and Robillard, 2011) :

— Un projet intégrateur (« capstone project ») de quatrième année faits par des étudiants de Polytechnique, dont l'objectif était de créer un outil complètement nouveau.
— Un projet intégrateur (« capstone project ») de quatrième année faits par des étudiants de Polytechnique, dont l'objectif était d'ajouter une interface graphique à un outil existant.
— Un projet à code ouvert (« open source ») possédant des artéfacts de design conçus avant d'écrire le code.

L'analyse a fait ressortir le fait que plusieurs problèmes ont été causés par ignorance, par manque de compétences et par des problèmes de communication au sein de l'équipe. L'analyse a aussi souligné le lien entre ces problèmes de connaissance et des problèmes au niveau de la prise de décision collective. La connaissance a donc un impact sur la qualité logicielle, notamment parce que des connaissances insuffisantes peuvent mener à de mauvaises décisions ayant des impacts à long terme sur la qualité logicielle.

4.3 Étudier la connaissance au sein du travail en équipe

Le lien entre gestion de connaissance et qualité logicielle souligne l'importance de mieux comprendre comment les informations transigent au sein de l'équipe, et l'impact de ces transactions sur la prise de décision collective. Un détour a d'abord été nécessaire vers le domaine de la psychologie organisationnelle.

> Organizational psychologists study and assess individual, group and organizational dynamics in the workplace. They apply that research to identify solutions to problems that improve the well-being and performance of organizations and their employees. (American Psychological Association, 2016)

Les interactions ont été étudiées sur la base d'une taxonomie existante reconnue dans le domaine de la psychologie organisationnelle (Marks et al., 2001). Cette taxonomie a été adaptée au contexte du génie logiciel (Ton-That et al., 2013). La taxonomie adaptée a été utilisée comme base pour l'analyse de projets de développement logiciel (Robillard et al., 2014b). L'analyse se base sur l'étude de journaux de travail (« work diary ») produit dans le cadre de projets intégrateurs d'étudiants terminant leur baccalauréat en génie logiciel à l'école Polytechnique[1]. Contrairement à l'outil présenté initialement, le codage des entrées du journal de travail vers la taxonomie a démontrée être répétable lorsque faite par plusieurs chercheurs.

Cette analyse a permis de détecter quelques problèmes au niveau de la gestion de connaissance dans un projet de développement logiciel. Par exemple, modifier les prévisions à long terme (e.g. des changements dans les exigences) en fin de projet est souvent corrélé avec des pics d'effort subséquents. Un changement majeur dans les prévisions à long terme implique des changements au niveau du design, ce qui peut résulter en beaucoup de changements au niveau du code source. Il y a lieu de se poser des questions sur :

— **La qualité de la décision prise de modifier les prévisions à long terme** : la recherche démontre en effet que face à des pressions, l'équipe tend à privilégier une prise de décision naturaliste, c'est-à-dire sauter sur la première solution possible, plutôt qu'une prise de décision rationnelle où des alternatives sont étudiées (Moe et al., 2012).
— **La qualité des modifications faites au design** : introduire des problèmes dans le design implique ajouter une dette technique au produit logiciel (McConnell, 2013). Sous pression pour respecter les échéanciers, la probabilité que les bons principes de développement logiciels soient abandonnés est élevée.
— **La qualité du code source écrit rapidement en fin de projet** : la recherche démontre que faire travailler un développeur en-dehors de ses heures habituelles causent une augmentation significative du nombre de bogues introduits (Eyolfson et al., 2014).

Le problème majeur de l'utilisation de la taxonomie est que les problèmes identifiés pouvaient l'être sans l'aide de celle-ci. L'apport de la taxonomie dans la détection de problèmes semble négligeable. Une approche observatoire serait potentiellement plus bénéfique car les résultats pourraient potentiellement être plus riches.

1. Voir l'article de Lethbridge et al. (2005) pour plus de détails sur les méthodes de recherche en génie logiciel, notamment l'utilisation de journaux de travail.

4.4 Comprendre la littérature sur la prise de décision collective

Il y a donc eu un besoin de retourner dans la littérature afin de lire les travaux déjà publiés sur la gestion de connaissance collective. Le problème a été de défricher et d'organiser toutes les publications dans un domaine particulier, dans un contexte ou aucun des chercheurs n'avait d'expérience dans le domaine. La difficulté d'aborder et de comprendre un nouveau domaine a souligné l'erreur faite initialement avec l'outil de gestion de connaissance : on ne peut pas simplement donner de la documentation à un développeur et espérer qu'il va en saisir immédiatement l'ensemble.

Les chercheurs ont effectués plusieurs revues systématiques de littérature dans le cadre d'un cours gradué avec des étudiants de Polytechnique faisant un certificat, une maîtrise ou un doctorat. Les leçons apprises durant ces revues systématiques ont permis de développer un processus itératif permettant à des novices d'aborder progressivement un nouveau domaine et d'en cartographier le contenu (Lavallée et al., 2014). Alors que les méthodologies de revue systématiques présentaient une approche par étape (i.e. similaire à un processus de développement logiciel cascade), le travail avec des novices démontre l'importance de pouvoir revenir en arrière, de raffiner le travail déjà fait, donc de procéder par itération (i.e. similaire à un processus de développement logiciel spirale).

Le processus itératif de revue de littérature a mené à plusieurs publications d'études cartographiques (« mapping studies ») de domaines initialement inconnu des chercheurs (Lavallée and Robillard, 2012; Lavallée et al., 2013). La dernière en particulier, sur les études liant cognition distribuée et génie logiciel, nous a permis de mieux comprendre la prise de décision collective.

La cognition distribuée étudie comment une personne résout ses problèmes non pas seulement grâce à ses propres capacités, mais aussi avec l'aide de son environnement. L'environnement inclut les outils logiciels, la documentation disponibles, mais aussi les collègues. Les concepts de cognition distribuées nous permettent de voir l'équipe d'une manière différente, non pas comme une somme d'individus travaillant de manière isolée (e.g. en silos), mais comme un organisme travaillant de concert vers un but commun.

Par exemple, un concept important de la cognition distribuée est la méta-cognition, c'est-à-dire les connaissances qu'une personne a sur ses propres connaissances. Des études en génie logiciel ont démontré l'importance pour une équipe de développement d'avoir une bonne méta-cognition, soit de connaître où se trouvent les connaissances au sein de l'équipe. Une équipe qui connait les véritables experts et leur domaine respectif sait à qui poser des questions lorsque nécessaire. Il s'agit d'un facteur important quand on sait que la première source

de réponses d'un développeur est au sein de sa communauté de pratique (Wenger and Snyder, 2000).

Un autre concept important en cognition distribuée est la conscience de la situation, c'est-à-dire à quel point une personne est au courant de ce qui se passe autour d'elle. Dans le travail en équipe, ce concept s'évalue sur le niveau auquel les développeurs sont au courant des informations provenant d'ailleurs et qui pourraient avoir un impact sur leur tâche. Par exemple, un collègue change les valeurs retournées par une fonction logicielle utilisée par notre développeur. Quand est-ce que notre développeur est mis au courant de ce changement qui l'affecte ? Par quels mécanismes est-il informé ?

Ce point de vue a souligné la problématique de la prise de décision collective. La complexité du développement logiciel (Brooks, 1987) impose la construction d'équipes non-homogènes où chaque personne apporte son expertise particulière. La recherche démontre qu'une équipe avec des expertises hétérogène est plus performante qu'une équipe avec des expertises homogènes (DeChurch and Memser-Magnus, 2010). L'approche rationnelle suppose que des alternatives sont étudiées, mais comment l'équipe peut-elle faire une étude rationnelle des alternatives si des informations cruciales ne se retrouvent pas sur la table au moment de prendre la décision ? Il semble donc y avoir un besoin de distribuer certaines informations, qui doivent être intégrées sous forme de connaissance par les personnes participant à la prise de décision. Il n'est pas nécessaire que toutes les connaissances soient parfaitement partagées, mais certains détails doivent l'être afin de supporter une approche décisionnelle mixte, telle que décrite par Moe et al. (2012) et Zannier et al. (2007). Le Saint-Graal de la gestion de la connaissance semble donc de pouvoir distribuer la bonne information, aux bonnes personnes, au bon moment, dans le bon format.

La question qui est soulevée est : Est-ce que le développement peut être planifié comme une horloge, comme un procédé industriel ? Ou bien est-ce que le travail de développement logiciel se fait de manière plus opportuniste ? L'analyse statique de code source et de d'artéfacts de génie logiciel indique que les changements sont faits selon ce qui semble le plus pertinent au moment de la prise de décision (Robillard et al., 2014a). Il ne semble y avoir que peu d'opportunités pour prendre du recul sur le travail fait et pour décider si la vue d'ensemble fait toujours du sens, si le travail se dirige toujours vers une direction adéquate.

L'objectif à ce moment est d'améliorer la cognition distribuée de l'équipe en choisissant précautionneusement les changements à apporter dans le processus de développement logiciel. L'important est donc de viser les moments où des prises de décision critiques sont prises, et la manière dont ces décisions sont prises. L'obstacle sera alors de vendre aux développeurs et gestionnaires de projets logiciels une amélioration potentielle capable d'avoir un impact

positif observable sur leur prise de décision.

4.5 Comprendre la littérature sur les changements de processus

Cette dernière constatation a illuminé un aspect important de la recherche actuelle en génie logiciel. Beaucoup de chercheurs recommandent aux équipes de développement d'intégrer un nouvel outil, une nouvelle pratique, un nouvel artéfact, bref de changer leurs habitudes de développement logiciel. Cette accumulation de tâches supplémentaires est d'ailleurs une des causes du manifeste Agile, qui dénonçait cette tendance à toujours demander plus de documentation (Beck et al., 2001). Une revue cartographique de la littérature, basée sur la méthode présentée précédemment, a donc été faite sur les améliorations de processus de développement logiciel (« software process improvement » ou SPI).

La revue a conclut que, malheureusement, il n'est pas facile de convaincre une équipe de développement logiciel de changer ses manières de faire. La revue de littérature systématique effectuée sur le sujet (Lavallée and Robillard, 2012) a souligné le paradoxe suivant :

— D'un côté, pour être accepté par les développeurs, le changement doit présenter des bénéfices tangibles.
— D'un autre côté, il faut environ 18 mois pour qu'un changement dans le processus soit suffisamment intégré dans les équipes de développement et commence à montrer des bénéfices mesurables.

Cela explique en partie pourquoi l'industrie a de la difficulté à intégrer les recommandations faites par la littérature : il est difficile de convaincre un gestionnaire de financer un changement qui ne présentera un retour sur l'investissement (« return on investment » ou ROI) potentiel qu'après 18 mois.

La revue a de plus souligné que beaucoup d'améliorations imposées ne s'attaquent pas aux véritables causes d'un problème. Par exemple, une organisation qui produit du logiciel avec beaucoup de bogues pourrait demander à ses équipes de faire plus de tests afin d'améliorer la qualité de ses produits (solution technique). Cependant, la cause du manque de qualité est que l'administration impose des échéanciers trop agressifs qui obligent les développeurs à tourner les coins ronds (cause organisationnelle) (Hall et al., 2001).

Or, il est difficile d'évaluer la performance d'une équipe (Dickinson and McIntyre, 1997), et donc encore plus d'évaluer la performance d'une pratique de gestion de connaissance. Faute de résultats tangibles, il devient difficile de convaincre gestionnaires et développeurs d'accepter un changement dans leurs manières de faire. Une amélioration de processus liée à la gestion de la connaissance doit donc, ou bien imposer un changement minimal, ou bien tirer profits

d'outils et pratiques déjà en place.

4.6 Comprendre le lien entre gestion organisationnelle et gestion de connaissances

Le livre de Nonaka and Takeuchi (1995) décrit comment une innovation provenant d'un individu peut se propager à l'ensemble de l'organisation. Le livre décrit l'importance de maximiser ce genre de décisions stratégiques au sein d'une organisation. Mais qu'en est-il des décisions opérationnelles ?

La recherche en production manufacturière indique qu'il y a des différence entre les visions stratégiques et les besoins opérationnels (Hanna and Jackson, 2015). L'opération manufacturière a besoin de produits répondant à des exigences spécifiques, alors que la stratégie organisationnelle a besoin de minimiser les coûts de production. Il s'ensuit des incohérences lorsque des maillons de la chaîne de production sont sous-contractées à l'extérieur de l'organisation afin de minimiser les coûts, et que ces maillons ne produisent pas des produits répondant aux besoins opérationnels de la manufacture.

Est-il possible d'assurer que les décisions opérationnelles, qui en logiciel ont souvent des incidences stratégiques en créant une dette technique (McConnell, 2013), soient mieux supportées ? Une étude de dix mois en industrie a donc observé les réunions de suivi hebdomadaire d'une équipe de développement logiciel terminant un projet de deux ans (Lavallée and Robillard, 2015). Il a été découvert que pour les décisions opérationnelles, l'équipe de développement dépendait beaucoup d'informations provenant d'autres équipes. Des mauvaises décisions affectant négativement la qualité logicielle sont souvent prises en toute connaissance de cause afin de satisfaire des décisions stratégiques prises au niveau organisationnel (échéanciers agressifs, budgets serrés, besoin de travailler avec des sous-contractant difficiles, etc.) (Lavallée and Robillard, 2015).

Les décisions opérationnelles faites de manière opportunistes sans considérer les alternatives auront des impacts à long terme. Ces impacts seront supportés par d'autres projets futurs ; l'équipe de développement actuelle ne se sent pas concernée car elle doit répondre à d'autres impératifs plus urgents. Les décisions stratégiques sont peut-être adéquatement supportées, mais les décisions opérationnelles ne se prennent pas dans des conditions idéales. Il en résulte des produits logiciels inadéquats, et qui resteront un irritant pour l'organisation car pour paraphraser Parnas (1994), « les seuls logiciels qui ne changent jamais sont ceux qui ne sont pas utilisés ».

4.7 Lier prise de décision et communauté de pratique

Il s'ensuit que les décisions opérationnelles prises par l'équipe de développement logiciel peuvent avoir des impacts stratégiques sur l'organisation. Un outil logiciel mal conçu peut être un handicap pour l'organisation, qui doit composer avec les limites de cet outil dans ses opérations quotidiennes.

Le problème est que des décisions critiques sont prises par l'équipe de développement dans l'urgence du moment. Il y a une déconnexion entre l'administration qui impose des contraintes, sans comprendre toute la portée des impacts stratégiques de ces contraintes. Ces décisions critiques pour la conception du logiciel ne devrait pas être prises seulement par l'équipe. Il n'est donc pas suffisant de s'assurer que l'équipe possède toute les connaissances nécessaires pour faire son travail. Même avec toutes les connaissances disponibles, l'équipe prendra de mauvaises décisions parce que l'éventail d'alternatives qui lui sont accessibles est trop restreint. Il devient donc nécessaire de cultiver un interlocuteur, possédant à la fois les compétences techniques nécessaires pour comprendre la base de code affectée, et les compétences administratives nécessaires pour comprendre les motivations de la direction.

Ce niveau existe dans la littérature au niveau de ce qu'on appelle la « communauté de pratique » (Wenger, 1999). La communauté de pratique d'une organisation regroupe les personnes possédant un bagage technique suffisamment similaire pour pouvoir se comprendre dans le jargon du domaine. Ce qui a été observé sur le terrain cependant semble indiquer qu'une telle communauté dans une grande organisation serait trop disjointe, et prompte à des conflits internes (Lavallée and Robillard, 2015). Par exemple, une organisation comme Microsoft possède une grande communauté de pratique d'ingénieurs logiciels, mais à quel point les ingénieurs qui travaillent sur le logiciel Windows comprennent le logiciel Word ?

CHAPITRE 5 DISCUSSION GÉNÉRALE

5.1 Lien entre structure organisationnelle et structure du logiciel

Le lien entre organisation et structure du produit a été explicité sous la forme de la « loi de Conway ». L'énoncé de la loi indique que :

> Any organization that designs a system (defined broadly) will produce a design whose structure is a copy of the organization's communication structure. (Conway, 1968)

Le lien entre structure organisationnelle et structure du produit logiciel est cependant plus évident à travers la citation suivante :

> Software engineering researchers have proposed many design principles and mechanisms such as modularization, encapsulation, information hiding, object-orientation, aspect-orientation to divide system into smaller units that are relatively independent of each other. Such small units, or modules, are then assigned as tasks to individual developers in the hope of reducing the needs and cost of communication and coordination among software developers. (Ye et al., 2009)

Une solution plus récente est la popularisation des architectures par microservices. Les microservices représentent des modules de code pouvant fonctionner indépendamment les uns des autres, potentiellement dans des langages différents et utilisant des bases de données indépendantes (Lewis and Fowler, 2014). Conséquemment, chaque service peut être développé par une équipe indépendante, avec un minimum de coordination inter-équipes.

Certaines organisations poussent l'approche par microservices plus loin, en imposant aux équipes de développement la responsabilité de supporter le microservice durant toute sa vie utile, un principe appelé familièrement par Amazon comme « You build it, you run it » (Lewis and Fowler, 2014; O'Hanlon, 2006; Cockcroft, 2013).

> The traditional model is that you take your software to the wall that separates development and operations, and throw it over and then forget about it. (O'Hanlon, 2006)

L'approche « You build it, you run it » assure que les connaissances accumulées durant le développement ne sont pas perdues au moment de passer en maintenance. Cela impose aussi

à l'équipe de développement de penser à la qualité à long terme, car les développeurs devront faire la maintenance, potentiellement pour des années à venir.

Ultimement, ces modules ou microservices doivent être intégrés ensembles, ce qui requiert un certain degré de coordination. Pour des modules développés localement, cette coordination peut être invisible à travers des interactions ad hoc (Herbsleb and Grinter, 1999; Gutwin et al., 2008). Cette coordination peut devenir plus difficile lorsque les équipes ne travaillent pas dans le même espace. Même deux équipes travaillant dans le même bâtiment mais dans des bureaux différents peuvent avoir des problèmes de communication (Lavallée and Robillard, 2015).

> It should also be noted that modularization has drawbacks. For example, [...] modularization can lead to integration problems because it encourages "minimal communication between teams responsible for interdependent modules". (Bailey et al., 2013) qui citent (Cataldo and Herbsleb, 2008)

Peu importe l'architecture choisie, l'important est d'avoir une cohérence entre la structure du produit (i.e. l'architecture) et la structure de l'organisation.

> When looking to split a large application into parts, often management focuses on the technology layer, leading to UI teams, server-side logic teams, and database teams. When teams are separated along these lines, even simple changes can lead to a cross-team project taking time and budgetary approval. A smart team will optimise around this and plump for the lesser of two evils - just force the logic into whichever application they have access to. (Lewis and Fowler, 2014)

La problématique à ce niveau est que la structure du produit est instable. Des imprévus surviennent, qui demandent de changer la structure planifiée initialement. Idéalement, la structure de l'organisation doit aussi s'adapter en conséquence, afin d'éviter que celle-ci soit incohérente avec la structure du code.

> The solution for the organization is two-fold and simple, at least in concept. First, self-recognition that some problems exist within the organization that have to be dealt with at the structural level, and second, when such problems occur that the organization must make an honest attempt to change structurally in reaction. (Bailey et al., 2013), basé sur (Schaefer, 2005)

5.2 Problème de déconnexion entre organisation et produit

L'approche de Nonaka and Takeuchi (1995), soulignant l'importance du transfert de connaissances de l'individu vers l'organisation est intéressante, mais les recherches récentes démontrent que ce transfert doit être bidirectionnel. Les connaissances doivent circuler aussi à l'inverse, de l'organisation, des autres équipes, vers l'équipe de développement et l'individu. Des décisions sont prises à tous les paliers, autant par l'individu, par l'équipe, par un département, et par l'organisation, mais ces décisions ne semblent pas basées sur toutes les connaissances pertinentes.

Il y a, par exemple, une déconnexion importante entre la direction administrative et les équipes techniques en ce qui a trait aux impacts à long terme des décisions prises aujourd'hui. Le changement de terme récent de vieillissement du logiciel (« software aging ») vers le terme de dette technique (« technical debt ») semble refléter un besoin de reconnecter les techniciens du logiciel avec les gestionnaires administratifs de l'organisation (McConnell, 2013).

Dans les faits, il semble manquer en logiciel un palier décisionnel, un point de jonction, entre les équipes techniques et l'administration de l'organisation. Ce palier décisionnel aurait la responsabilité d'évaluer et de maintenir une vision globale sur une base de code particulière. Pour faire un parallèle avec l'industrie manufacturière, l'administration d'une usine s'assure que l'ensemble de l'usine fonctionne adéquatement, autant du point de vue technique (production d'une qualité adéquate) qu'administrative (production rentable). Ce pallier intermédiaire sert de jonction entre la haute direction de l'organisation et les équipes techniques de production. Ce type de pallier intermédiaire semble être un des avantages de l'approche par microservices et du « You build it, you run it », c'est-à-dire de s'assurer que toutes les connaissances accumulées durant le développement sont maintenues durant la maintenance.

Un tel palier pourrait cependant être utile même dans des projets plus volumineux qu'un microservice. Un tel pallier éviterait l'accumulation de dette technique. Par exemple, le projet observé dans le cadre des travaux de Lavallée and Robillard (2015) soulignent le cas d'un module qui avait douze enveloppes (« wrappers ») parce qu'aucun gestionnaire de projet ne voulait investir dans la correction définitive du module, même si cette correction aurait été bénéfique à long terme.

5.3 Besoin d'une cohérence entre la prise de décision et le design logiciel

Le domaine de recherche a déjà défini un concept proche de celui de pallier décisionnel sur une base de code, soit le concept de communauté de pratique (« communities of practice ») (Bjørnson and Dingsøyr, 2008). Le problème est que la communauté de pratique, telle que

définie actuellement, suppose une communauté de tous les experts techniques au sein d'une organisation. Les observations semblent démontrer que les équipes fonctionnent présentement de manière très isolées, partiellement parce qu'elle sont gérées de manière indépendante, mais aussi parce qu'elles travaillent sur des modules logiciels très disparates.

L'avancement de la recherche en génie logiciel souligne la spécialisation des ingénieurs dans des domaines de plus en plus pointus (e.g. experts en base de données, en sécurité, en assurance-qualité, en interface utilisateur, en infrastructure, etc.). À moins qu'une base de code particulière demande les expertises de toutes ces personnes en même temps, il est peu probable que ces personnes interagissent. Le concept de communauté de pratique pan-organisationnelle reste difficile à supporter sur le terrain. Communauté de pratique implique deux personnes ou plus pratiquant ensemble, donc dans un projet conjoint demandant leur expertise respective. Plutôt que d'essayer de cultiver une communauté pan-organisationnelle, l'organisation devrait se concentrer sur les communautés qui émergent naturellement de par le développement de modules qui demandent des expertises communes. Donc de travailler sur des communautés de pratique au niveau de la base de code, plutôt qu'au niveau de l'organisation complète.

Les observations présentées par Lavallée and Robillard (2015) décrivent cette situation, où une grande quantité de développeurs spécialisées se rencontrent dans la production d'un module logiciel. Ces développeurs sont présentés dans la figure 5.1.

De même, l'analyse faite par Lavallée and Robillard (soumis en octobre 2016) montre la bidirectionnalité de certaines interactions et le degré de proximité de certaines équipes dans le cadre d'un développement logiciel. Cette proximité entre les équipes n'existe que dans le cadre d'un projet donné touchant un module donné. Selon la manière de fonctionner actuelle, cette proximité entre équipes disparait une fois que le projet de développement logiciel est terminé. Les interactions entre les équipes sont présentées dans la figure 5.2.

Un développeur ferait donc partie de plusieurs communautés de pratiques distribués sur plusieurs bases de code, de la même manière qu'en développement à code ouvert ("open source"), un volontaire peut faire partie de plusieurs projets, de plusieurs communautés. Ces communautés de pratiques "orientée-produit" respecteraient alors la loi de Conway, où la structure organisationnelle se reflète dans la structure du produit. Ces communautés de pratique orientée-produit seraient associées à un produit logiciel particulier supporté par l'organisation et seraient responsables des prises de décisions techniques ayant un impact stratégique sur celui-ci. Ces communautés seraient aussi un outil pour les équipes de développement logiciel, afin de convaincre l'administration de lever des contraintes ayant un impact majeur sur la qualité du produit, impact qui pourrait avoir des conséquences sur la

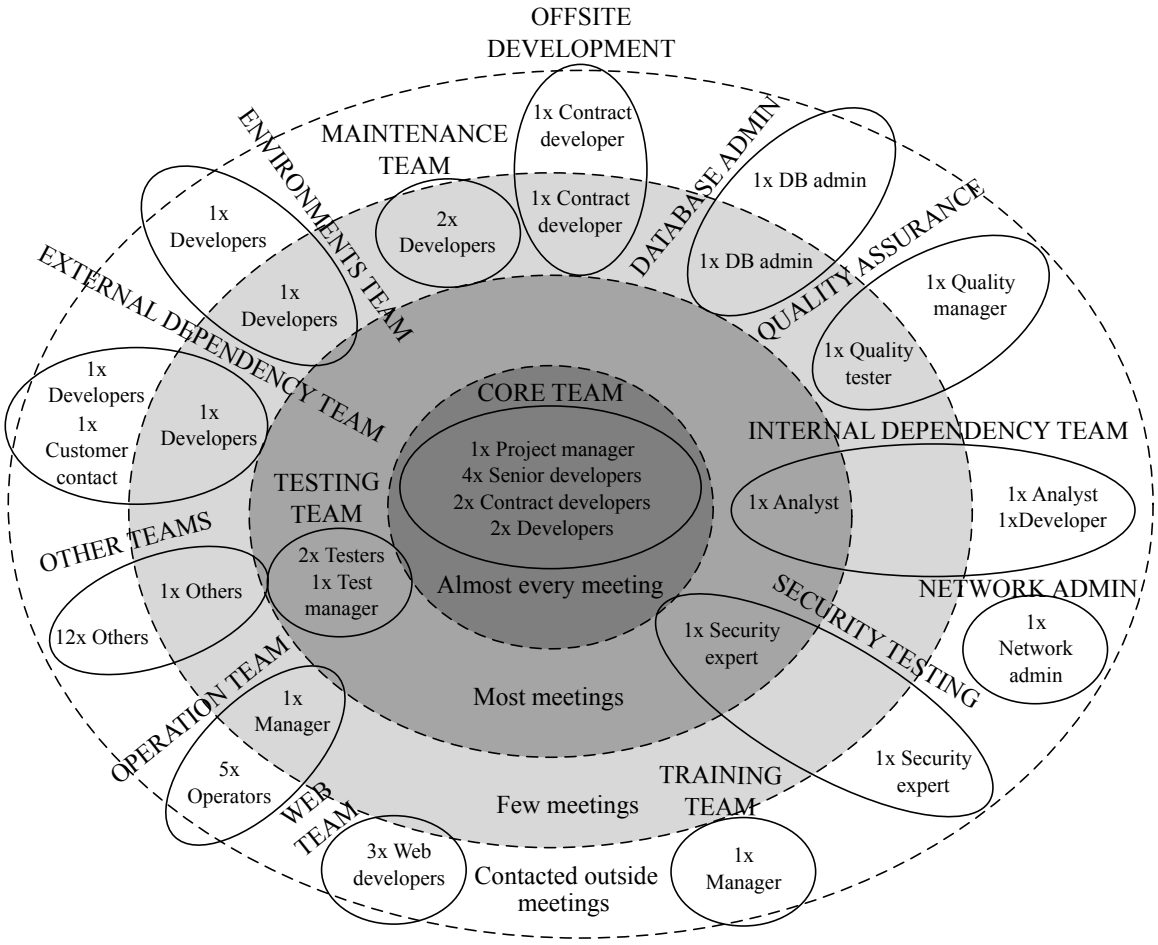

Figure 5.1 Équipes de développeurs spécialisés interagissant dans le cadre d'un projet industriel. Image tirée de Lavallée and Robillard (2015) et utilisée avec permission.

pérennité de l'organisation.

5.4 Impact économique de l'approche actuelle

Les difficultés de Netscape (Wired, 1999), bien qu'elles datent déjà de plus de quinze ans, sont des exemples probant de l'impact qu'une dette technique accumulée peut avoir sur l'avenir d'une organisation. L'accumulation de mauvaises décisions sur le produit Netscape a causé un ralentissement du développement, au point où les nouveaux entrant sur le marché ayant une base de code moins « agée » pouvait fournir de nouvelles fonctionnalités plus rapidement.

Un autre exemple est celui du développement de Longhorn, l'ancêtre de Windows Vista, qui a dut être redémarré en 2004 pour cause de manque d'une vision commune.

> Longhorn was irredeemable because Microsoft engineers were building it just as

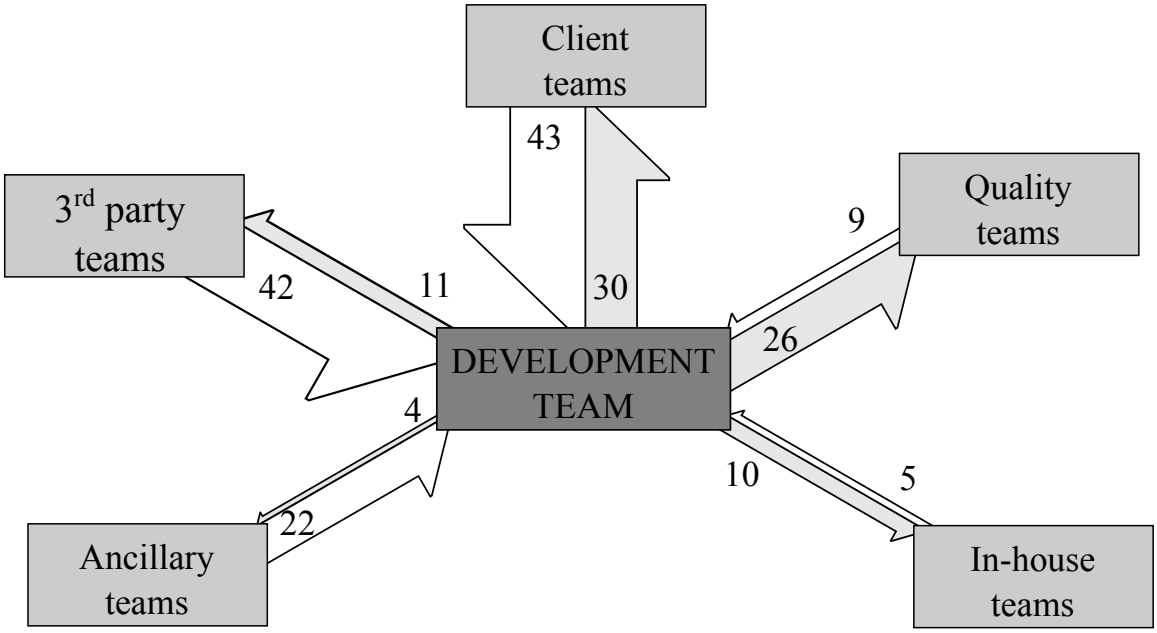

Figure 5.2 Interactions entre les équipes dans le cadre d'un projet industriel. Les valeurs et la taille des flèches représentent le nombre d'interactions entre les équipes. Image tirée de Lavallée and Robillard (soumis en octobre 2016) et utilisé avec permission.

> they had always built software. Throughout its history, Microsoft had let thousands of programmers each produce their own piece of computer code, then stitched it together into one sprawling program. Now, Mr. Allchin argued, the jig was up. Microsoft needed to start over. (Guth, 2005)

Cette approche fut désastreuse, au point où le cœur de l'application a dû être abandonné pour être remplacé par du code provenant de Windows Server 2003. Avec une vision globale plus adéquate, Microsoft a pu finalement livrer Vista en 2007, mais avec un succès mitigé.

Mais s'il s'agit d'une question de vie ou de mort pour une organisation qui distribue du logiciel, qu'en est-il pour les autres ? Une organisation commerciale, par exemple, peut-elle laisser son bloc logiciel ERP accumuler une dette technique, quitte à tout réécrire par la suite ? L'histoire de Kmart semble démontrer le contraire (Information Week, 2003). Le retard technologique du géant de la distribution sur son logiciel SCM (Supply Chain Management) n'a pas pu être rattrapé à temps pour éviter la faillite.

Comment une organisation comme Google s'assure-t-elle qu'une base de code importante, deux milliards de lignes de code selon leurs estimations (Metz, 2015), reste suffisamment de bonne qualité ? Comment une organisation comme Microsoft s'assure-t-elle qu'une base de code logiciel comme Windows reste maintenable ? Pour une organisation dont le pain et le

beurre est la maintenance continue de son logiciel, il est critique de s'assurer que le logiciel reste maintenable, et le restera pour les années, voire les décennies à venir [1]. La clé, dans ces cas, est de déjouer la tendance de vieillissement du logiciel (Parnas, 1994) en évitant de traîner une dette technique (McConnell, 2013) qui pourrait mener à la non-maintenabilité du code, et donc à la mort de l'organisation. Déjouer le vieillissement demande de s'assurer que tout changement sur le logiciel n'a pas d'impacts trop néfastes. Cela ne peut être fait qu'en ayant une vue globale de la base de code.

5.5 Rapprochement entre organisation et produit

Des conclusions précédentes recommandaient la création d'une table de négociation inter-équipes, afin de s'assurer que les changements faits par une équipe n'étaient pas néfastes à une autre (Lavallée and Robillard, 2015). Pour que ces négociations soient faites adéquatement, il est nécessaire d'avoir une vue globale de la situation, des impacts potentiels des changements demandés. L'analyse d'impacts est une pratique reconnue lors de la maintenance (International Standard Organization, 2006). Il s'agit d'évaluer et de minimiser les impacts techniques tout en estimant les coûts et les risques. Une analyse d'impacts appropriée demande d'avoir une compréhension globale.

Avec la complexification des logiciels par contre, avoir une vue globale est de plus en plus difficile (Carroll et al., 2006). Il y a de plus en plus de logiciels, et ceux-ci sont de plus en plus interdépendants. Il devient alors nécessaire de cultiver des personnes qui possèdent cette vue globale et qui peuvent fournir leur rétroaction lorsque nécessaire. Il devient nécessaire d'avoir une communauté de pratique centrée sur chaque base de code particulière, qui s'assure que les connaissances en lien avec cette base de code ne disparait pas lorsque les employés quittent l'organisation.

Cela ne veut pas dire que tout projet logiciel doit inclure ce nouveau pallier administratif/technique. Les praticiens soulignent l'importance du code brouillon, et que beaucoup de code écrit n'est pas destiné à être utilisé sur une longue période (Hacker News). Ce besoin ne serait pertinent que pour les bases de code logiciels complexes dont la maintenabilité à long terme est un besoin critique pour l'organisation.

1. Ce défi a été initialement porté à l'attention de l'auteur grâce à Nicolas Cloutier suite à une discussion par courriel

CHAPITRE 6 CONCLUSIONS ET RECOMMANDATIONS

Certains promoteurs de l'architecture par microservices recommandent que l'équipe qui développe le microservice logiciel soit aussi responsable de sa maintenance et de son opération dans l'environnement de production. Rien n'empêche que ce principe, « You build it, you run it » (O'Hanlon, 2006; Cockcroft, 2013) soit aussi appliqué au développement traditionnel. Ce qu'il faut assurer, c'est que les connaissances accumulées durant le développement et la maintenance ne soient pas perdues au fil du temps, et donc assurer une supervision technique de la base de code.

Cette façon de faire va permettre de développer une communauté de pratique, non pas pan-organisationnelle, mais basée sur un produit logiciel particulier, communauté qui émergera naturellement du fait que des expertises diverses doivent travailler ensemble pour la réalisation initiale du produit. L'organisation devrait donc travailler à encourager ces communautés de pratiques orientées-produit, d'une manière similaire aux projets à code ouvert (« open source »), qui encouragent les développeurs à rester et à partager leurs connaissances avec les nouveaux.

6.1 Synthèse des travaux

Les premiers travaux ont tenté de faciliter la gestion de connaissance à travers l'utilisation d'un outil logiciel. La prévalence de cette approche dans la littérature actuelle (Bjørnson and Dingsøyr, 2008) et les résultats mitigés obtenus (Kerzazi et al., 2013) ont démontré le besoin de suivre une approche différente.

Des études ont donc été faites afin de mieux comprendre l'origine des problèmes de qualité logicielle, et comment ces problèmes peuvent-être liés à la gestion de connaissance (Lavallée and Robillard, 2011; Robillard et al., 2014a). Ces études ont démontré l'importance du partage de la connaissance au sein des équipes de développement logiciel. En particulier, ces études ont souligné l'impact des échanges au sein de l'équipe, et comment ces échanges peuvent mener à des décisions ayant des impacts négatifs sur la qualité logicielle.

Les modèles existants en psychologie organisationnelle ont été étudiés. Des revues de littérature ont été faites, sur la base d'une nouvelle approche d'étude de la littérature (Lavallée et al., 2014). Un modèle de psychologie cognitive en particulier a été adapté au génie logiciel, et validé sur le terrain (Robillard et al., 2014b; Ton-That et al., 2013). Une meilleure compréhension du fonctionnement des équipes, notamment sur comment la prise de décision

se fait par une équipe de développement dans son environnement (Lavallée et al., 2013), a permis de formuler certaines recommandations. Cependant, ces recommandations se doivent d'être compatibles avec les manières de faire des équipes de développement logiciel (Lavallée and Robillard, 2012).

Une équipe professionnelle de développement logiciel a donc été observée pendant dix mois afin d'avoir une vue plus globale des mécanismes de prise de décision (Lavallée and Robillard, 2015). Ces observations ont permis de comprendre que la cause des problèmes de qualité logicielle échappe souvent au contrôle de l'équipe de développement (Lavallée and Robillard, 2015), comme d'autres travaux l'ont suggéré précédemment (Lavallée and Robillard, 2012). Les recommandations menant à des améliorations de processus ne devraient donc pas être limitées à l'équipe seule, toute l'organisation doit être impliquée.

La conclusion finale est que la connaissance accumulée lors d'un projet de développement logiciel doit être maintenue par la suite. Certaines organisations assurent ce maintient à travers le principe « You build it, you run it » (O'Hanlon, 2006; Cockcroft, 2013), qui assure que les personnes qui ont développé un microservice logiciel sont aussi en charge de le maintenir et d'assurer son opération dans l'environnement de production.

Ce principe pourrait être porté en-dehors des microservices vers le développement traditionnel, à travers la construction de communautés de pratiques orientées-produit. C'est-à-dire que plutôt qu'avoir une communauté de pratique pan-organisationnelle, avec des acteurs ayant peu de liens entre eux, il faudrait supporter les communautés de pratique qui émergent naturellement durant le développement logiciel, entre les acteurs qui doivent travailler entre eux pour créer le produit. Cette communauté de pratique serait la gardienne de la connaissance au niveau de la base de code (« codebase ») d'un produit, et s'assurerait que les décisions majeures affectant ce produit se prennent en toute conaissance de cause, c'est-à-dire en évaluant les impacts à long terme des changements proposés. L'objectif est d'éviter des décisions qui profitent immédiatement à une équipe au détriment de toutes les autres travaillant sur cette base de code.

6.2 Limitations de la solution proposée

Le travail fait jusqu'à présent est essentiellement exploratoire. Une seule organisation privée a été étudiée en détails (Lavallée and Robillard, 2015). Cependant, les observations faites ont été confirmées dans une autre organisation publique, de même que dans certains projets étudiants et à code ouvert (Lavallée and Robillard, 2011; Robillard et al., 2014a). Des réactions saisies sur l'internet suite à la publication d'un article (Lavallée and Robillard, 2015) semblent de

plus souligner l'importance de la problématique (Hacker News; Reddit).

Il est donc difficile à ce stade de déterminer dans quels contextes les conclusions et recommandations faites peuvent s'appliquer. Les données obtenues confirment que ces conclusions sont pertinentes dans des contextes variés (organisation privée, organisation publique, etc.), mais les caractéristiques d'un projet logiciel qui justifieraient ce degré d'attention à la prise de décision ne sont pas claires à ce stade. La prochaine étape serait donc de voir à quel point ces recommandations sont bénéfiques dans une variété de contextes.

L'objectif serait de déterminer comment supporter des communautés de pratiques orientées-produit, et dans quels contextes ces communautés auraient un impact positif et pourquoi.

6.3 Améliorations futures

Les résultats démontrent l'importance du partage de connaissance au sein des équipes, en particulier du partage lié à une base de code logicielle (« codebase ») particulière. La recommandation principale de ce projet de recherche est qu'il est nécessaire de garder une vue globale de la base de code, afin de s'assurer que ce code reste dans un état capable de supporter les besoins futurs de l'organisation. Lorsque la base de code ou une partie de celle-ci se retrouve dans un état tel qu'il faut absolument la remplacer, cela ne peut se faire qu'à grand coût et grand risque, étant donné la quantité d'interdépendances entre les modules logiciels.

Nous recommandons donc de monter un nouveau palier décisionnel dédié à la qualité du code, qui n'observe pas seulement le code modifié dans le cadre d'un projet unique, mais la base de code (« codebase ») complète qui est modifiée par un ensemble de projets. Ce palier décisionnel, orientée autour de la communauté de pratique qui aurait construit le code en question, aurait un budget assigné dédié à faire des projets d'amélioration de qualité, afin d'éviter l'accumulation de dette technique, en particulier aux frontières entre les modules (Lavallée and Robillard, 2015). Ce budget permettrait aussi de faciliter les négociations entre les équipes, qui pourraient répondre aux besoins de la base de code sans handicaper leur projet en cours. Cela permettrait d'abaisser les barrières entre les équipes et permettrait d'assurer que toutes les équipes travaillent en coopération plutôt qu'en compétition les unes avec les autres.

Conceptuellement, ce palier décisionnel lié à la communauté de pratique serait le point de jonction entre les côtés administratifs et techniques de l'organisation. Ce point de jonction assurerait la cohérence entre les limites administratives (ressources disponibles, échéanciers) imposés par la direction et le besoin de maintenir un code adéquat pour les années futures. De même que l'organisation doit débourser pour réparer le toit d'une usine qui coule, de

même l'organisation devrait débourser afin d'éviter que le toit de leur code se mette, métaphoriquement, à couler.

RÉFÉRENCES

American Psychological Association, “Industrial and organizational psychology provides workplace solutions”, http://www.apa.org/action/science/organizational/index.aspx, 2016, accessed : 2016-07-12.

N. Anquetil, K. M. de Oliveira, K. D. de Sousa, et M. G. Batista Dias, “Software maintenance seen as a knowledge management issue”, *Inf. Softw. Technol.*, vol. 49, no. 5, pp. 515–529, Mai 2007. DOI : 10.1016/j.infsof.2006.07.007. En ligne : http://dx.doi.org/10.1016/j.infsof.2006.07.007

R. Atkinson et R. Shiffrin, “Human memory : A proposed system and its control processes1”, série Psychology of Learning and Motivation, K. W. Spence et J. T. Spence, éds. Academic Press, 1968, vol. 2, pp. 89 – 195. DOI : http://dx.doi.org/10.1016/S0079-7421(08)60422-3. En ligne : http://www.sciencedirect.com/science/article/pii/S0079742108604223

G. Avelino, M. T. Valente, et A. Hora, “What is the truck factor of popular github applications ? a first assessment”, *PeerJ PrePrints*, 2015. DOI : 10.7287/peerj.preprints.1233v2

S. E. Bailey, S. S. Godbole, C. D. Knutson, et J. L. Krein, “A decade of conway’s law : A literature review from 2003-2012”, dans *Proceedings of the 2013 3rd International Workshop on Replication in Empirical Software Engineering Research*, série RESER ’13. Washington, DC, USA : IEEE Computer Society, 2013, pp. 1–14. DOI : 10.1109/RESER.2013.14. En ligne : http://dx.doi.org/10.1109/RESER.2013.14

V. R. Basili, F. E. McGarry, R. Pajerski, et M. V. Zelkowitz, “Lessons learned from 25 years of process improvement : the rise and fall of the nasa software engineering laboratory”, dans *Software Engineering, 2002. ICSE 2002. Proceedings of the 24rd International Conference on*, May 2002, pp. 69–79.

K. Beck, M. Beedle, A. van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, J. Kern, B. Marick, R. C. Martin, S. Mellor, K. Schwaber, J. Sutherland, et D. Thomas, “Manifeste pour le développement agile de logiciels”, 2001. En ligne : http://agilemanifesto.org/iso/fr/

J. Biolchini, P. Gomes Mian, A. Candida Cruz Natali, et G. Horta Travassos, “Systematic

review in software engineering", Systems Engineering and Computer Science Department, Universidade Federal do Rio de Janeiro, Rapp. tech. ES 679/05, 2005.

F. O. Bjørnson et T. Dingsøyr, "Knowledge management in software engineering : A systematic review of studied concepts, findings and research methods used", *Inf. Softw. Technol.*, vol. 50, no. 11, pp. 1055–1068, Oct. 2008. DOI : 10.1016/j.infsof.2008.03.006. En ligne : http://dx.doi.org/10.1016/j.infsof.2008.03.006

J. Bowers et T. Rodden, "Exploding the interface : Experiences of a cscw network", dans *Proceedings of the INTERACT '93 and CHI '93 Conference on Human Factors in Computing Systems*, série CHI '93. New York, NY, USA : ACM, 1993, pp. 255–262. DOI : 10.1145/169059.169205. En ligne : http://doi.acm.org/10.1145/169059.169205

J. Brewer, *Ethnography*. Open University Press, 2000.

F. P. Brooks, Jr., "No silver bullet essence and accidents of software engineering", *Computer*, vol. 20, no. 4, pp. 10–19, Avr. 1987. DOI : 10.1109/MC.1987.1663532. En ligne : http://dx.doi.org/10.1109/MC.1987.1663532

K. M. Carley, "Extracting team mental models through textual analysis", *Journal of Organizational Behavior*, vol. 18, no. S1, pp. 533–558, 1997.

J. M. Carroll, M. B. Rosson, G. Convertino, et C. H. Ganoe, "Awareness and teamwork in computer-supported collaborations", *Interacting with Computers*, vol. 18, no. 1, pp. 21 – 46, 2006, special Theme Papers from Special Editorial Board Members (Contains Regular Papers). DOI : http://dx.doi.org/10.1016/j.intcom.2005.05.005. En ligne : http://www.sciencedirect.com/science/article/pii/S0953543805000524

M. Cataldo et J. D. Herbsleb, "Communication patterns in geographically distributed software development and engineers' contributions to the development effort", dans *Proceedings of the 2008 International Workshop on Cooperative and Human Aspects of Software Engineering*, série CHASE '08. New York, NY, USA : ACM, 2008, pp. 25–28. DOI : 10.1145/1370114.1370121. En ligne : http://doi.acm.org/10.1145/1370114.1370121

F. Chiocchio, S. Grenier, T. A. O'Neill, K. Savaria, et J. D. Willms, "The effects of collaboration on performance : a multilevel validation in project teams", *International Journal of Project Organisation and Management*, vol. 4, no. 1, pp. 1–37, 2012, pMID : 45362. DOI : 10.1504/IJPOM.2012.045362. En ligne : http://www.inderscienceonline.com/doi/abs/10.1504/IJPOM.2012.045362

P. Clarke et R. V. O'Connor, "The situational factors that affect the software development process : Towards a comprehensive reference framework", *Inf. Softw. Technol.*, vol. 54, no. 5, pp. 433–447, Mai 2012. DOI : 10.1016/j.infsof.2011.12.003. En ligne : http://dx.doi.org/10.1016/j.infsof.2011.12.003

A. Cockcroft, "Now playing on netflix : Adventures in a cloudy future", November 2013, flowcon.

M. Cohn, "Advice on conducting the scrum of scrums meeting", 2007. En ligne : https://www.scrumalliance.org/community/articles/2007/may/advice-on-conducting-the-scrum-of-scrums-meeting

G. Coleman et R. O'Connor, *Software Process in Practice : A Grounded Theory of the Irish Software Industry*. Berlin, Heidelberg : Springer Berlin Heidelberg, 2006, pp. 28–39.

——, "Using grounded theory to understand software process improvement : A study of irish software product companies", *Inf. Softw. Technol.*, vol. 49, no. 6, pp. 654–667, Juin 2007. DOI : 10.1016/j.infsof.2007.02.011. En ligne : http://dx.doi.org/10.1016/j.infsof.2007.02.011

R. Conradi et T. Dingsøyr, "Software experience bases : A consolidated evaluation and status report", dans *Product Focused Software Process Improvement, Second International Conference, PROFES 2000, Oulu, Finland, June 20-22, 2000, Proceedings*, 2000, pp. 391–406.

M. E. Conway, "How do committees invent ?" *Datamation*, 1968.

C. K. W. De Dreu, B. A. Nijstad, et D. van Knippenberg, "Motivated information processing in group judgment and decision making", *Personality and Social Psychology Review*, vol. 12, no. 1, pp. 22–49, 2008. DOI : 10.1177/1088868307304092. En ligne : http://psr.sagepub.com/content/12/1/22.abstract

L. A. DeChurch et J. J. Memser-Magnus, "The cognitive underpinnings of effective teamwork : a meta-analysis", *The Journal of applied psychology*, vol. 95, no. 1, pp. 32–53, 2010. DOI : 10.1037/a0017328

T. L. Dickinson et R. M. McIntyre, "A conceptual framework of teamwork measurement", dans *Team Performance Assessment and Measurement : Theory, Methods, and Applications*, M. T. Brannick, E. Salas, et C. Prince, éds. NJ : Psychology Press, 1997, ch. 2, pp. 19–43.

T. Dingsøyr, F. O. Bjørnson, et F. Shull, “What do we know about knowledge management ? practical implications for software engineering”, *IEEE Software*, vol. 26, no. 3, pp. 100–103, May 2009. DOI : 10.1109/MS.2009.82

G. Disterer, “Management of project knowledge and experiences”, *Journal of Knowledge Management*, vol. 6, no. 5, pp. 512–520, 2002. DOI : 10.1108/13673270210450450. En ligne : http://dx.doi.org/10.1108/13673270210450450

J. A. Espinosa, S. A. Slaughter, R. E. Kraut, et J. D. Herbsleb, “Team knowledge and coordination in geographically distributed software development”, *Journal of Management Information Systems*, vol. 24, no. 1, pp. 135–169, 2007. DOI : 10.2753/MIS0742-1222240104. En ligne : http://www.tandfonline.com/doi/abs/10.2753/MIS0742-1222240104

J. Eyolfson, L. Tan, et P. Lam, “Correlations between bugginess and time-based commit characteristics”, *Empirical Softw. Engg.*, vol. 19, no. 4, pp. 1009–1039, Août 2014. DOI : 10.1007/s10664-013-9245-0. En ligne : http://dx.doi.org/10.1007/s10664-013-9245-0

T. Flanagan, C. Eckert, et P. J. Clarkson, “Externalizing tacit overview knowledge : A model-based approach to supporting design teams”, *Artif. Intell. Eng. Des. Anal. Manuf.*, vol. 21, no. 3, pp. 227–242, Juin 2007. DOI : 10.1017/S089006040700025X. En ligne : http://dx.doi.org/10.1017/S089006040700025X

E. Gamma, R. Helm, R. Johnson, et J. Vlissides, *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1995.

R. A. Guth, “Battling google, microsoft changes how it builds software delay in new windows version drove giant to develop simpler, flexible product”, *Wall Street Journal*, 2005.

C. Gutwin, S. Greenberg, R. Blum, J. Dyck, K. Tee, et G. McEwan, “Supporting informal collaboration in shared-workspace groupware”, *Journal of Universal Computer Science*, no. 9, pp. 1411–1434, 2008.

Hacker News, “Why good developers write bad code : An observational case study”, https://news.ycombinator.com/item?id=9565218, accessed : 2016-08-17.

T. Hall, A. Rainer, N. Baddoo, et S. Beecham, “An empirical study of maintenance issues within process improvement programmes in the software industry”, dans *Software Main-*

tenance, 2001. Proceedings. IEEE International Conference on, 2001, pp. 422–430. DOI : 10.1109/ICSM.2001.972755

V. Hanna et J. Jackson, "An examination of the strategic and operational impact of global sourcing on uk small firms", *Production Planning & Control*, vol. 26, no. 10, pp. 786–798, 2015. DOI : 10.1080/09537287.2014.983580. En ligne : http://dx.doi.org/10.1080/09537287.2014.983580

B. H. Hansen et K. Kautz, *Knowledge Mapping : A Technique for Identifying Knowledge Flows in Software Organisations*. Berlin, Heidelberg : Springer Berlin Heidelberg, 2004, pp. 126–137.

J. D. Herbsleb et R. E. Grinter, "Architectures, coordination, and distance : Conway's law and beyond", *IEEE Softw.*, vol. 16, no. 5, pp. 63–70, Sep. 1999. DOI : 10.1109/52.795103. En ligne : http://dx.doi.org/10.1109/52.795103

T. Howard, S. Culley, et E. Dekoninck, "Describing the creative design process by the integration of engineering design and cognitive psychology literature", *Design Studies*, vol. 29, no. 2, pp. 160 – 180, 2008. DOI : 10.1016/j.destud.2008.01.001. En ligne : http://www.sciencedirect.com/science/article/pii/S0142694X08000173

Information Week, "Now in bankruptcy, kmart struggled with supply chain", *Information Week*, 2003.

International Standard Organization, "Iso/iec 14764 :2006 software engineering – software life cycle processes – maintenance", 2006.

——, "ISO 25000 – System and Software Quality Requirements and Evaluation (SQuaRE)", International Standard Organization, ISO 25000, 2016. En ligne : http://iso25000.com/index.php/en/iso-25000-standards

M. I. Kellner, R. J. Madachy, et D. M. Raffo, "Software process simulation modeling : Why ? what ? how ?" *Journal of Systems and Software*, vol. 46, no. 2–3, pp. 91 – 105, 1999. DOI : http://dx.doi.org/10.1016/S0164-1212(99)00003-5. En ligne : http://www.sciencedirect.com/science/article/pii/S0164121299000035

N. Kerzazi, M. Lavallée, et P. N. Robillard, "A knowledge-based perspective for software process modeling", *E-Informatika Software Engineering Journal*, vol. 7, pp. 25 – 33, 2013.

B. Kitchenham, "Procedures for performing systematic reviews", Keele University, Rapp. tech. TR/SE-0401, 2004.

Y. Kozak, *Barriers Against Better Team Performance in Agile Software Projects*. Gøteborg, Suède : Chalmers University of Technology, 2013.

P. Kruchten, "Where did all this good architectural knowledge go ?" dans *Proceedings of the 4th European Conference on Software Architecture*, série ECSA'10. Berlin, Heidelberg : Springer-Verlag, 2010, pp. 5–6. En ligne : http://dl.acm.org/citation.cfm?id=1887899.1887902

M. Lavallée et P. N. Robillard, "The impacts of software process improvement on developers : A systematic review", dans *2012 34th International Conference on Software Engineering (ICSE)*, June 2012, pp. 113–122. DOI : 10.1109/ICSE.2012.6227201

M. Lavallée, P. N. Robillard, et R. Mirsalari, "Performing systematic literature reviews with novices : An iterative approach", *IEEE Transactions on Education*, vol. 57, no. 3, pp. 175–181, Aug 2014. DOI : 10.1109/TE.2013.2292570

M. Lavallée et P. N. Robillard, "Why good developers write bad code : An observational case study of the impacts of organizational factors on software quality", dans *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1, May 2015, pp. 677–687. DOI : 10.1109/ICSE.2015.83

——, "Are we working well with others ? how the multi team systems impacts software quality", *Software Quality Journal*, soumis en octobre 2016.

M. Lavallée, P. N. Robillard, et S. Paul, "Distributed cognition in software engineering : A mapping study", dans *eKNOW 2013 : The Fifth International Conference on Information, Process and Knowledge Management*, 2013, p. 6.

M. Lavallée et P. N. Robillard, "Causes of premature aging during software development : An observational study", dans *Proceedings of the 12th International Workshop on Principles of Software Evolution and the 7th Annual ERCIM Workshop on Software Evolution*, série IWPSE-EVOL '11. New York, NY, USA : ACM, 2011, pp. 61–70. DOI : 10.1145/2024445.2024458. En ligne : http://doi.acm.org/10.1145/2024445.2024458

——, "Planning for the unknown : Lessons learned from ten months of non-participant exploratory observations in the industry", dans *Proceedings of the Third International Workshop on Conducting Empirical Studies in Industry*, série CESI '15. Piscataway, NJ, USA : IEEE Press, 2015, pp. 12–18. En ligne : http://dl.acm.org/citation.cfm?id=2819303.2819311

T. O. Lehtinen, M. V. Mäntylä, J. Vanhanen, J. Itkonen, et C. Lassenius, “Perceived causes of software project failures – an analysis of their relationships”, *Information and Software Technology*, vol. 56, no. 6, pp. 623 – 643, 2014. DOI : http://dx.doi.org/10.1016/j.infsof.2014.01.015. En ligne : http://www.sciencedirect.com/science/article/pii/S0950584914000263

T. C. Lethbridge, S. E. Sim, et J. Singer, “Studying software engineers : Data collection techniques for software field studies”, *Empirical Softw. Engg.*, vol. 10, no. 3, pp. 311–341, Juil. 2005. DOI : 10.1007/s10664-005-1290-x. En ligne : http://dx.doi.org/10.1007/s10664-005-1290-x

J. Lewis et M. Fowler, “Microservices - a definition of this new architectural term”, http://martinfowler.com/articles/microservices.html, 2014, accessed : 2016-07-29.

J. P. Looney et M. E. Nissen, “Organizational metacognition : The importance of knowing the knowledge network”, dans *Proceedings of the 40th Annual Hawaii International Conference on System Sciences*, série HICSS ’07. Washington, DC, USA : IEEE Computer Society, 2007, pp. 190c–. DOI : 10.1109/HICSS.2007.419. En ligne : http://dx.doi.org/10.1109/HICSS.2007.419

B. Luthiger, “Fun and software development”, dans *Proceedings of the First International Conference on Open Source Systems*, M. Scotto et G. Succi, éds., 2005, pp. 273–278.

K. Lyytinen et D. Robey, “Learning failure in information systems development”, *Information Systems Journal*, vol. 9, no. 2, pp. 85–101, 1999. DOI : 10.1046/j.1365-2575.1999.00051.x. En ligne : http://dx.doi.org/10.1046/j.1365-2575.1999.00051.x

A. M. Madni, “Integrating humans with software and systems : Technical challenges and a research agenda”, *Systems Engineering*, vol. 13, no. 3, pp. 232–245, 2010. DOI : 10.1002/sys.20145. En ligne : http://dx.doi.org/10.1002/sys.20145

M. V. Mäntylä et C. Lassenius, “Subjective evaluation of software evolvability using code smells : An empirical study”, *Empirical Softw. Engg.*, vol. 11, no. 3, pp. 395–431, Sep. 2006. DOI : 10.1007/s10664-006-9002-8. En ligne : http://dx.doi.org/10.1007/s10664-006-9002-8

M. A. Marks, J. E. Mathieu, et S. J. Zaccaro, “A temporally based framework and taxonomy of team processes”, *Academy of Management Review*, vol. 26,

no. 3, pp. 356–376, 2001. DOI : 10.5465/AMR.2001.4845785. En ligne : http://amr.aom.org/content/26/3/356.abstract

S. McConnell, *Managing Technical Debt.* Construx Software Builders Inc., 2013.

C. Metz, “Google is 2 billion lines of code — and it’s all in one place”, *Wired*, 2015.

Microsoft, *Microsoft Application Architecture Guide, 2nd Edition, patterns & practices.* Microsoft Press, october 2009.

G. A. Miller, “The magical number seven, plus or minus two : some limits on our capacity for processing information”, *Psychological Review*, vol. 63, no. 2, pp. 81 – 97, March 1956. DOI : http://dx.doi.org/10.1037/h0043158

N. B. Moe, A. Aurum, et T. Dybå, “Challenges of shared decision-making : A multiple case study of agile software development”, *Inf. Softw. Technol.*, vol. 54, no. 8, pp. 853–865, Août 2012. DOI : 10.1016/j.infsof.2011.11.006. En ligne : http://dx.doi.org/10.1016/j.infsof.2011.11.006

N. Moha, Y. G. Gueheneuc, L. Duchien, et A. F. L. Meur, “Decor : A method for the specification and detection of code and design smells”, *IEEE Transactions on Software Engineering*, vol. 36, no. 1, pp. 20–36, Jan 2010. DOI : 10.1109/TSE.2009.50

A. Morton, *A Guide through the Theory of Knowledge, Third Edition.* Blackwell Publishing, 2003.

S. J. Motowidlo et J. R. V. Scotter, “Evidence that task performance should be distinguished from contextual performance”, *Journal of Applied Psychology*, vol. 79, no. 4, pp. 475 – 480, July 1994.

Mountain Goat Software. En ligne : https://www.mountaingoatsoftware.com/agile/scrum

I. Nonaka et H. Takeuchi, *The Knowledge-Creating Company.* Oxford University Press, 1995.

Object Management Group, “Software & systems process engineering metamodel specification (spem) version 2.0”, Object Management Group, Rapp. tech., 2008.

——, “Omg unified modeling language version 2.5”, Object Management Group, Rapp. tech., 2015.

C. O'Hanlon, "A conversation with werner vogels", *Queue*, vol. 4, no. 4, pp. 14 :14–14 :22, Mai 2006. DOI : 10.1145/1142055.1142065. En ligne : http://doi.acm.org/10.1145/1142055.1142065

D. L. Parnas, "Software aging", dans *Proceedings of the 16th International Conference on Software Engineering*, série ICSE '94. Los Alamitos, CA, USA : IEEE Computer Society Press, 1994, pp. 279–287. En ligne : http://dl.acm.org/citation.cfm?id=257734.257788

J.-C. Pomerol, P. Brézillon, et L. Pasquier, "Operational knowledge representation for practical decision-making", *Journal of Management Information Systems*, vol. 18, no. 4, pp. 101–115, 2002. En ligne : http://www.jstor.org/stable/40398544

V. Rajlich, "Changing the paradigm of software engineering", *Commun. ACM*, vol. 49, no. 8, pp. 67–70, Août 2006. DOI : 10.1145/1145287.1145289. En ligne : http://doi.acm.org/10.1145/1145287.1145289

Reddit, "Why good developers write bad code : An observational case study of the impacts of organizational factors on software quality", https://www.reddit.com/r/programming/comments/36dyx4/why_good_developers_write_bad_code_an/, accessed : 2016-08-17.

S. K. Reed, *Cognition : Théories et applications, 2e édition.* de Boeck, 2007.

P. N. Robillard, "The role of knowledge in software development", *Commun. ACM*, vol. 42, no. 1, pp. 87–92, Jan. 1999. DOI : 10.1145/291469.291476. En ligne : http://doi.acm.org/10.1145/291469.291476

P. N. Robillard, M. Lavallée, Y. Ton-That, et F. Chiocchio, "Taxonomy for software teamwork measurement", *Journal of Software : Evolution and Process*, vol. 26, no. 10, pp. 910–922, 2014. DOI : 10.1002/smr.1641. En ligne : http://dx.doi.org/10.1002/smr.1641

P. N. Robillard, M. Lavallée, et O. Gendreau, "Quality control practice based on design artifacts categories : Results from a case study", dans *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*, série EASE '14. New York, NY, USA : ACM, 2014, pp. 35 :1–35 :10. DOI : 10.1145/2601248.2601249. En ligne : http://doi.acm.org/10.1145/2601248.2601249

J. Rowley, "The wisdom hierarchy : representations of the dikw hierarchy", *Journal of Information Science*, vol. 33, no. 2, pp. 163–180, 2007. DOI : 10.1177/0165551506070706.

En ligne : http://jis.sagepub.com/content/33/2/163.abstract

I. Rus et M. Lindvall, “Knowledge management in software engineering”, *IEEE Software*, vol. 19, no. 3, pp. 26–38, 2002.

I. Rus, M. Lindvall, et S. S. Sinha, “Knowledge management in software engineering : A dacs state-of-the-art report”, Fraunhofer Center for Experimental Software Engineering Maryland, Rapp. tech., 2001.

E. Salas, N. J. Cooke, et M. A. Rosen, “On teams, teamwork, and team performance : Discoveries and developments”, *Human Factors : The Journal of the Human Factors and Ergonomics Society*, vol. 50, no. 3, pp. 540–547, 2008. DOI : 10.1518/001872008X288457. En ligne : http://hfs.sagepub.com/content/50/3/540.abstract

P. Salembier, “Cognition(s) : Située, distribuée, socialement partagée, etc., etc., ...” GRIC ARAMIIHS Toulouse, Rapp. tech., 1995.

R. Schaefer, “The risks of large organizations in developing complex systems”, *SIGSOFT Softw. Eng. Notes*, vol. 30, no. 5, pp. 1–3, Sep. 2005. DOI : 10.1145/1095430.1095444. En ligne : http://doi.acm.org/10.1145/1095430.1095444

H. A. Simon et A. Newell, “Human problem solving : The state of the theory in 1970”, *American Psychologist*, vol. 26, no. 2, pp. 145 – 159, February 1971. DOI : http://dx.doi.org/10.1037

A. L. Strauss, *Qualitative Analysis for Social Scientists*. Cambridge University Press, 2003.

Y. Ton-That, P. N. Robillard, et M. Lavallée, “Episode measurement method : A data collection technique for observing team processes”, dans *Proceedings of the 2013 International Conference on Software and System Process*, série ICSSP 2013. New York, NY, USA : ACM, 2013, pp. 108–117. DOI : 10.1145/2486046.2486066. En ligne : http://doi.acm.org/10.1145/2486046.2486066

M. J. Turk, “How to scale a code in the human dimension”, dans *Scientific Software Days*, 2013.

D. Waddington et P. Lardieri, “Model-centric software development”, *Computer*, 2006.

A. Walenstein, “Observing and measuring cognitive support : Steps toward systematic tool evaluation and engineering”, dans *Proceedings of the 11th IEEE International Workshop*

on Program Comprehension, série IWPC '03. Washington, DC, USA : IEEE Computer Society, 2003, pp. 185–. En ligne : http://dl.acm.org/citation.cfm?id=851042.857038

E. Wenger, *Communities of Practice : Learning, Meaning and Identity*. Cambridge : Cambridge University Press, 1999.

E. C. Wenger et W. M. Snyder, "Communities of practice : The organizational frontier", *Harvard Business Review*, January-February 2000.

L. Williams et R. Kessler, *Pair Programming Illuminated*. Addison-Wesley, 2003.

Wired, "Netscape browser guru : We failed", *Wired*, 1999.

Y. Ye, "Supporting software development as knowledge-intensive and collaborative activity", dans *Proceedings of the 2006 International Workshop on Workshop on Interdisciplinary Software Engineering Research*, série WISER '06. New York, NY, USA : ACM, 2006, pp. 15–22. DOI : 10.1145/1137661.1137666. En ligne : http://doi.acm.org/10.1145/1137661.1137666

Y. Ye, K. Nakakoji, et Y. Yamamoto, *Measuring and Monitoring Task Couplings of Developers and Development Sites in Global Software Development*. Berlin, Heidelberg : Springer Berlin Heidelberg, 2009, pp. 181–195. DOI : 10.1007/978-3-642-01856-5_13. En ligne : http://dx.doi.org/10.1007/978-3-642-01856-5_13

C. Zannier et F. Maurer, "Foundations of agile decision making from agile mentors and developers", dans *Proceedings of the 7th International Conference on Extreme Programming and Agile Processes in Software Engineering*, 2006, pp. 11–20.

C. Zannier, M. Chiasson, et F. Maurer, "A model of design decision making based on empirical results of interviews with software designers", *Information and Software Technology*, vol. 49, no. 6, pp. 637 – 653, 2007, qualitative Software Engineering Research. DOI : http://dx.doi.org/10.1016/j.infsof.2007.02.010. En ligne : http://www.sciencedirect.com/science/article/pii/S0950584907000122

ANNEXE A LISTE DES ARTICLES

Les annexes suivantes présentent les articles publiés dans le cadre du doctorat. Les annexes B à G contiennent les articles discutés dans le chapitre 4 de cette thèse. Tel que présenté dans le tableau A, les annexes sont présentées en ordre inverse d'année de publication.

Tableau A.1 Articles en lien avec cette thèse.

Annexe	Référence complète	Contribution au travail de recherche
B	*Are We Working Well with Others ? How the Multi Team Systems Impacts Software Quality*, **Mathieu Lavallée** et Pierre N. Robillard, Software Quality Journal, article soumis	Cet article décrit les interactions observées entre une équipe de développement logiciel est les autres équipes d'une organisation. Les résultats démontrent que le nombre d'interactions avec les autres équipes suit une loi de Pareto, avec une poignée d'équipe étant la source de la majorité des interactions. Les résultats démontrent aussi l'importance qu'ont les développeurs en tant que point de jonction entre les différentes équipes. Finalement, les résultats montrent aussi l'importance des discussions techniques au niveau organisationnel.

Annexe	Référence complète	Contribution au travail de recherche
C	*Why Good Developers Write Bad Code : An Observational Case Study of the Impacts of Organizational Factors on Software Quality*, **Mathieu Lavallée** et Pierre N. Robillard, International Conference on Software Engineering (ICSE), 2015, ACM SIGSOFT Distinguished Paper Award, doi :10.1109/ICSE.2015.83	Cet article présente des problèmes de qualité majeurs et comment ces problèmes peuvent être liés, non pas à des problèmes au sein de l'équipe comme on pourrait s'y attendre, mais à des problèmes organisationnels. Par exemple, le fait que les processus de développement se sont développés organiquement au fil des échecs de projets fait que certains fragments de processus ne communiquent pas entre eux. Dans ce cas, l'équipe d'assurance-qualité maintient une liste de bogues connus dans les logiciels en production, mais cette liste n'est pas transmise aux développeurs qui font la mise-à-jour de ces logiciels.
D	*Planning for the Unknown : Lessons Learned from Ten Months of Non-Participant Exploratory Observations in the Industry*, **Mathieu Lavallée** et Pierre N. Robillard, Workshop on Conducting Empirical Studies in Industry (CESI), 2015, doi :10.1109/CESI.2015.10	Cet article présente la méthodologie exploratoire utilisée dans l'article de l'annexe C. Les leçons apprises pendant le projet de recherche sont compilées et comparées aux autres leçons apprises publiées dans la littérature pour ce même genre de recherche. Sur la base de toutes ces leçon, un processus de recherche plus formel est proposé.

Annexe	Référence complète	Contribution au travail de recherche
E	*Performing Systematic Literature Reviews With Novices : An Iterative Approach*, **Mathieu Lavallée**, Pierre N. Robillard et Reza Mirsalari, IEEE Transactions on Education, vol.57, no.3, pages 175-181, 2013, doi : 10.1109/TE.2013.2292570	Cet article présente un processus de recherche pour l'exécution de revues systématiques de littérature dans un domaine où aucun expert n'est disponible, ou aucun expert n'existe. Le processus est basé sur nos expérience passées dans la réalisation de revues systématiques avec des novices, de même que sur les leçons apprises publiées dans la littérature.
F	*The Impacts of Software Process Improvement on Developers : A Systematic Review*, **Mathieu Lavallée** et Pierre N. Robillard, International Conference on Software Engineering (ICSE), 2012, doi :10.1109/ICSE.2012.6227201	Cet article présente les résultats d'une cartographie de la littérature dans le domaine des améliorations de processus de développement logiciel. La revue présente les conclusions majeures obtenues dans le domaine. On note en particulier le paradoxe que l'amélioration de processus doit présenter des résultats concrets afin d'être accepté par l'équipe de développement, et que dans bien des cas, ces améliorations peuvent prendre jusqu'à 18 mois avant d'apparaitre.
G	*Causes of Premature Aging During Software Development : An Observational Study*, **Mathieu Lavallée** et Pierre N. Robillard, International Workshop on the Principles of Software Evolution and ERCIM Workshop on Software Evolution (IWPSE-EVOL), 2011, doi :10.1145/2024445.2024458	Cet article présente des problèmes de qualité majeurs ainsi que leurs causes dans deux projets de fin d'étude et un projet à code ouvert initié par une équipe professionnelle (« open source »). On note que les problèmes ne sont pas toujours dûs à un manque d'expertise, mais souvent à des problèmes organisationnels ou de transfert de connaissances.

ANNEXE B ARTICLE 1 : ARE WE WORKING WELL WITH OTHERS ? HOW THE MULTI TEAM SYSTEMS IMPACTS SOFTWARE QUALITY

Type de publication	Référence complète
Journal (soumis)	*Are We Working Well with Others ? How the Multi Team Systems Impacts Software Quality*, **Mathieu Lavallée** et Pierre N. Robillard, Software Quality Journal, article soumis en octobre 2016

B.1 Abstract

There are many studies on software development teams. But what about the interactions between teams ? Multi-team system studies are few in numbers, but current findings suggest that these systems may have a significant impact on the performance of a project. A non-participatory approach was used to collect data on one development project within a large telecommunication organization. Verbal interactions between team members were analyzed using a coding scheme following the Grounded Theory approach. The results show that the interactions between teams are often technical in nature, outlining technical dependencies between departments, external providers, and even clients. They show that the participation of the client is required throughout the project, not only as a source of information for the requirements, but also as providers of technical details pertaining to their specific field. This article concludes that managers of large software project should (1) identify external teams most likely to interfere with their development work, to (2) appoint brokers to redirect external requests to the appropriate resource, and to (3) ensure that there are opportunities to discuss technical issues at the multi-team level. Failure to do so could results in delays, rush work, cancellation of testing activities, and the persistence of codebase-wide issues which could have a major impact on product quality.

B.2 Introduction

Five hundred years ago, John Donne wrote that “no man is an island”. Individuals achieve great things by working together as a team. But many projects require more than an individual team to achieve success. “No team is an island” (Porck, 2013) would be a better description of modern project and organization management.

Teamwork has indeed long been identified as important to project success (da Silva et al., 2013 ; Guzzo & Dickson, 1996 ; Guzzo & Salas, 1995). Teamwork in software development is no different, and software engineering research also highlighted the impacts that software development teams can have. As Watts S. Humphrey wrote, *"Systems development is a team activity, and the effectiveness of the team largely determines the quality of the engineering"* (Humphrey, 2000). Teams rarely work in isolation ; teams are often interdependent of each other and must work together. Recent studies have shown the importance of these interactions between teams, whether on issues such as organization-wide knowledge sharing (Santos, Goldman, & de Souza, 2015), coordination of multiple agile teams (Paasivaara, Lassenius, & Heikkila, 2012) or inter-team communication effectiveness (Martini, Pareto, & Bosch, 2013).

This paper presents insights gained from the analysis of data collected in an ethnographic study. The next section presents the related work (Section 2), with a focus on the organizational psychology concept of multi-team systems and how it applies to software engineering. The methodology (Section 3) presents the context of the study and how the data was collected and analyzed. The results (Section 4) presents the data analysis, while the discussion (Section 5) presents recommendations and limitations to the conclusions of the study. The conclusion (Section 6) summarizes the recommendations and presents future avenues of research. Note that this paper represents an extension of a previous shorter publication (Lavallee & Robillard, 2015b). Some elements of the methodology were reused here, but the results and analyses are new.

B.3 Related Work

The current software engineering literature uses different terms to define a number of concepts related to the interactions between multiple teams : inter-team, multi-team, cross-team, etc. However, these concepts are not always clearly defined, leaving the exact interpretation to the reader. The research field of organizational psychology has fortunately studied this topic extensively, regrouping them under the umbrella of multi-team systems, or MTS (Marks, DeChurch, Mathieu, Panzer, & Alonso, 2005). The MTS are defined as :

> Two or more teams that interface directly and interdependently in response to environmental contingencies toward the accomplishment of collective goals. MTS boundaries are defined by virtue of the fact that all teams within the system, while pursuing different proximal goals (e.g. writing a specific code module), share at least one common distal goal (e.g. creating a complete working software) ; and in so doing exhibit input, process, and outcome interdependence with at least one

> other team in the system. (Mathieu, Marks, & Zaccaro, 2001)

While many studies have been published on team dynamics in recent decades, as far as we could find, there are few publications on the dynamics between teams. What should be done to make teams work together effectively at the organizational level ? This question has already been studied in the field of organizational psychology.

> What is required for success in these kinds of MTSs is coordination both within and between teams [emphasis theirs]. That is, although interventions designed to create a system of strong, cohesive component teams may maximize performance at the team level, when ultimate system-level goals require synchronization between teams, more is needed. [...] MTS interventions must also address interdependencies between teams if performance across these kinds of complex systems is to be maximized. (Asencio, Carter, DeChurch, Zaccaro, & Fiore, 2012)

Studies observing MTS in software engineering are however still limited (A. Scheerer, Hildenbrand, & Kude, 2014), with all studies found limited to Agile contexts and Scrum-of-Scrums meetings, as shown in Table 1.

Mike Cohn, an expert on the Scrum process, recommends a specific point in the agenda of "Scrum of Scrums" meeting, his version of MTS status meetings. Cohn recommends the addition of a question saying : *"Are you about to put something in another team's way ?"* (Cohn, 2007). Cohn's recommendation outlines the importance of MTS and the impact one team can have on another. This recommendation was used in the field within "Scrum-of-Scrums" meetings, but with limited success (Paasivaara et al., 2012) :

> Both case projects started using a model in which only one issue was discussed : impediments. However, this solution did not turn out well.[...] Both case projects still recognized the need for project-wide inter-team synchronization, but did not have any good solutions to the problem. (Paasivaara et al., 2012)

This shows that while the challenge of MTS projects are beginning to be better known, working solutions are still far off.

B.3.1 Known Challenges of MTS Projects

This section presents a non-exhaustive list of challenges of MTS projects, based on what could be found in the literature. These three challenges were found to be most prevalent in the context of this study :

Table B.1 Software Engineering Publications related to MTS.

Ref	Title (publication year)
(Santos et al., 2015)	Fostering effective inter-team knowledge sharing in agile software development (2015)
(A. Scheerer et al., 2014)	Coordination in Large-Scale Agile Software Development : A Multiteam Systems Perspective (2014)
(Bannerman, Hossain, & Jeffery, 2012)	Scrum Practice Mitigation of Global Software Development Coordination Challenges : A Distinctive Advantage ? (2012)
(Lee, 2008)	Forming to Performing : Transitioning Large-Scale Project Into Agile (2008)
(Maranzato, Neubert, & Herculano, 2011)	Moving back to scrum and scaling to scrum of scrums in less than one year (2011)
(Mundra, Misra, & Dhawale, 2013)	Practical Scrum-Scrum team : Way to produce successful and quality software (2013)
(Paasivaara & Lassenius, 2011)	Scaling scrum in a large distributed project (2011)
(Alexander Scheerer, Bick, Hildenbrand, & Heinzl, 2015)	The Effects of Team Backlog Dependencies on Agile Multiteam Systems : A Graph Theoretical Approach (2015)
(Smits & Pshigoda, 2007)	Implementing scrum in a distributed software development organization (2007)
(Sutherland, Schoonheim, & Rijk, 2009)	Fully distributed scrum : Replicating local productivity and quality with offshore teams (2009)
(Tartaglia & Ramnath, 2005)	Using open spaces to resolve cross team issue (2005)

— Finding a compromise between team-level goals and MTS-level goals,
— Enabling effective communications and technical knowledge exchange at the MTS level,
— Planning the work at the MTS level.

One of the main MTS challenge is related to building a compromise between the objective of the local team goal and the overall goals of the MTS. In one software engineering case, the conflicting agendas of team members within different departments led to the failure of the project (Maranzato et al., 2011). This challenge has a major impact on resource allocation. Organizational psychology researchers observed that "having to simultaneously work toward team-level goals along with MTS-level goals creates a demanding work environment." (Asencio et al., 2012). In software engineering, Santos et al., reached a similar conclusion. They studied knowledge sharing between teams in an Agile context (Santos et al., 2015). They noted that the introduction of new MTS support practices requires more resources, which must be provided by the organization, otherwise the practice, and potentially the project, could fail.

Another challenge is the relative difficulty to ensure efficient communications at the MTS level, compared to communications within the team. A survey conducted by Kiani et al. noted that due to "lack of communication, almost fourth of respondents complained that work items they depended on have changed without any notification." (Kiani, Mite, & Riaz, 2013). Some basic Agile principles are also affected in MTS contexts. For example, face-to-face communications are easy at the team level, but are difficult to apply at the MTS level. It requires the organization to mix people from one team to another, which is not always possible (Chau & Maurer, 2004).

In the same vein, dissemination of technical information specific to a field of knowledge is also difficult. Local teams accumulate a significant amount of knowledge about the specific area in which they work. How can this knowledge be effectively communicated to the other teams in the MTS ? This challenge can become costly for a local team in a global development project, where MTS is distributed in different sites around the world (Bannerman et al., 2012). If the project is particularly complex, it may also be difficult to get an overall view of the project (Paasivaara & Lassenius, 2011). Each team knows its own problems, which can be difficult to translate in a form understandable by other teams that might not have the same knowledge of the field.

A third challenge is related to how MTS coordination should be planned. Lanaj et al. found the following.

> Decentralized planning has positive effects on multiteam system performance,

> attributable to enhanced proactivity and aspiration levels. However, [. . .] the positive effects associated with decentralized planning are offset by the even stronger negative effects attributable to excessive risk seeking and coordination failures. (Lanaj, Hollenbeck, Ilgen, Barnes, & Harmon, 2013)

The study of MTS coordination has been identified by one study as "underdeveloped" (A. Scheerer et al., 2014). However, MTS is a concept defined within the domain of organizational psychology. Research in software development already has a large body of knowledge pertaining to inter-team interactions within the domain of global and distributed software development (Verner, Brereton, Kitchenham, Turner, & Niazi, 2012). While a global or distributed development team is a form of MTS, some MTS can be collocated in the same building. The team observed interacted with other teams which were almost all collocated within the same building. The context of this study is therefore different from the study of global and distributed software development, where the issues of geographical distance and temporal distance play a large role.

B.4 Methodology

B.4.1 Industrial Context

The study was performed on a large telecommunications organization with over forty years in the industry. Throughout the years, the organization has developed a large codebase, which must be constantly updated. This study follows one such update project. The outcomes of this study are based on ten months of observation of a software development team involved in a two-year project for an internal client. The project involved a complete redesign of an existing software package used in the organization's internal business processes.

The technical challenge of this update project is that it requires the modification of COBOL legacy software, Web interfaces, mobile device integration and multiple databases. Its purpose is to manage work orders. To do this, it needs to extract data from multiple sources within the enterprise (employee list, equipment list, etc.) and send it to multiple databases (payroll, quality control, etc.).

The project was a second attempt to overhaul this complex package. A first attempt had been made between 2010 and 2012 but was abandoned after the fully integrated software did not work. Because this project was a second attempt, many specifications and design documents could be reused. Accordingly, the development followed a traditional waterfall process, as few problems were expected the second time around. This second attempt began in 2013 and was successfully deployed during October and November 2014.

The organization has no formal MTS coordination practices in place. Coordination at the MTS level is therefore mostly tacit. This means that when a team needs information from another team, a member of the first team has to directly contact another member of the second team. This causes some issues at the MTS level, because most developers in the team observed were new to the company (Lavallee & Robillard, 2015a) when the project started, and in some cases did not know who to contact in the other teams. Despite its tacit nature, a MTS exists. The need for coordination between the projects means that interactions between teams are required to perform the work.

This study observes a development team of nine members : one manager, four senior developers, two junior developers and two contract developers. The team was formed specifically for this project, of which seven are new to the organization (i.e. less than five years).

Note that the nature of this MTS is different from a MTS where several teams are working on the same project (e.g. a Scrum-of-Scrums development project). In the MTS observed, all teams had different projects, with their own goals and objectives. The development project studied was the responsibility of a single team, the team observed. However, to perform that project, that team could not do it alone, and had to seek help from other teams.

The objective of this study is to understand how a development team interacts with other external team to do its work. Therefore, the focus is on the development team. Who does the development team needs to talk to and why ?

B.4.2 Study Approach

The objective of the study was to identify the cause behind the introduction of quality problems during software development. Given the sensitive nature of problem identification within a large organization, it was decided to opt for a neutral approach. Data collection was to be performed using a non-participatory approach, to avoid organizational influence.

Data collection was limited to weekly status meetings because that is the avenue used by the organization to discuss and resolve MTS issues. Although there were certainly discussion between teams outside these weekly status meetings, the most important issues were discussed at these meetings.

The study approach is based on an ethnographical approach based on previous work by Brewer (2000).

> Ethnography is the study of people in naturally occurring settings or “fields” by means of methods with capture their social meanings and ordinary activities, involving the researcher participating directly in the setting, if not also the acti-

> vities, in order to collect data in a systematic manner but without meaning being imposed on them externally. Brewer (2000), page 10

A qualitative approach was chosen to better understand an area where many variables are not fully identified. The approach of this study uses the same rationale as Looney and Nissen (2007).

> The present research is exploratory in nature, is not guided by extensive theory, and is approaching a 'how' research question. Hence qualitative field research reflects an appropriate method. Looney and Nissen (2007)

B.4.3 Data Collection Methodology

This study is based on non-participant observation of the software development team's weekly status meetings. These meetings consisted of mandatory all-hands discussions for the eight developers assigned mostly full-time to the project, along with the project manager. These meetings included, as needed, developers from related external modules, testers, database administrators, security experts, quality control specialists, etc. The meetings involved up to 15 participants, and up to five additional participants through conference calls.

The team discussed the progress made during the previous week, the work planned for the coming week and obstacles to progress. The problems raised concerned resources and technical issues. Few decisions were taken at these meetings, the purpose being to share the content of the previous week's discussions between the different teams.

A round-table format was used, where each participant was asked to report on their activities. The discussions were open and everyone was encouraged to contribute. When a particular issue required too much time, participants were asked to set another meeting to discuss it. Meetings lasted about an hour.

The data presented in this study was collected over seven months during the last phase of the two-year project. It is based on 21 meetings held between January and July 2014. The same observer attended all the meetings and took note of who was involved in each interaction, the topic being discussed, and the outcome. A typical interaction would last between 5 and 30 seconds. The notes were then produced as quasi-verbatim transcripts.

B.4.4 Coding Methodology

Due to the large amount of data collected, it is necessary to summarize the data obtained in order to find patterns. This summarization was performed using a coding methodology

based on the grounded theory approach (Strauss, 2003).

Coding was performed after the observations were completed, based on the meeting notes taken from February 27th, 2014 to July 31st, 2014. Since it can take time for the people observed to be used to the presence of the researcher (Landsberger, 1958), and for the researchers themselves to fully understand the domain knowledge of the project (Lethbridge, Sim, & Singer, 2005), the data from the first two meetings were not kept for this study.

Meetings taking place after July 31st were also removed from this analysis. These last meetings were mostly related to deployment activities and featured very little development interactions. While the analysis of the deployment activities would be interesting, it was decided to keep the development discipline and deployment discipline separate, as the MTS requirements of both disciplines are quite different.

Coding schemes were developed following the Grounded theory approach (Strauss, 2003). In summary, coding was performed using the following steps :

1. Open coding of all entries, going over the data as long as new codes can be added,
2. When no new codes are be added, similar codes are grouped together,
3. Code groups are formalized into schemes,
4. Return to point (1) until no new codes are added and no new schemes can be formed.

After multiple coding iterations, three coding scheme emerged. The first scheme pertains to whether the interaction observation is related to a technical or administrative topic :

— **Technical** : interactions related to technical issues (requirements, bugs, data, etc.),
— **Administrative** : interactions related to administrative issues (deadlines, resources, etc.).

The second scheme pertains to one of the four types of interaction identified :

— **Team demands (inputs)** : These interactions are requests made by team members to someone outside the development team.
— **Team commitments (outputs)** : These interactions are requests made by someone outside the development team to the team or a team member.
— **Team coordination (in-out)** : These interactions are related to meetings which had or will take place between two or more teams on a given issue.
— **Team liaison (brokering)** : These interactions are information request to the development team by someone outside the team. The development team cannot answer themselves and therefore act as knowledge brokers with another team.

The third scheme pertains to the type of team interacted with :

— **Client teams** : These teams are responsible for providing requirements and details on what they need the software to do, along with validation of the final result.
— **3rd party teams** : These teams represents the 3rd party library support teams, which performs corrections on the software based on the service-level agreement (SLA) their 3rd party holds with the organization. Two internal module support teams are also included here, as the interaction with these teams followed a protocol similar to the interaction with support teams outside the organization.
— **Quality teams** : These teams are responsible of quality assurance and quality control within the organization.
— **Ancillary teams within the organization** : The organization has many departments, each with their own expertise and technical competencies. For example, one ancillary team was in charge of the creation and configuration of the development and test environments.
— **In-house development teams** : These teams represent other development teams working in parallel projects on the same codebase.

B.5 Results

Data collection returned a total of 464 topics discussed within the 21 weekly status meetings analyzed. From these 464 topics, 294 were related to external teams. Therefore, about 60

Figure 1 presents the number of interactions between the observed observed development team and all external teams. The teams are split based on the five team types presented in the previous section. The closer a team is to the dark center of Figure 1, the more interactions they had with the observed team, and the closer they were to them. Note that since it was an internal development project for an organization which does not sell software, the actual clients of the package upgraded was the Operations team. The Operations team is in charge of creating and dispatching work orders. Field workers receive the work orders and must on occasion interact with the software. A total of 29 different external teams were contacted during the course of the study.

Table 2 presents the results of the number of interactions with external team members according to their activities, which outlines the amount of interactions and the rationale for interaction (to answer team needs, to fulfill team obligations, etc.). Table 2 does not present the liaison interactions, which are presented in Figure 2. Table 2 shows that there are numerous administrative as well as technical interactions with all the team categories. Note that eight interactions could not be assigned to a specific team, bringing the total in Table 2 to 294 interactions.

Table B.2 Number of interactions with external teams per team category.

	Team Demands		Team Commitments		Team Coordination		Team Liaison		Total
Team Category	Tech	Admin	Tech	Admin	Tech	Admin	Tech	Admin	
Client teams	22	21	19	11	8	7	8	5	101
3rd party teams	35	7	8	3	6	4	4	4	71
Quality teams	3	6	19	7	6	4	10	5	60
Ancillary teams	14	8	3	1	2	0	7	0	35
In-house teams	4	1	8	2	1	1	1	1	19

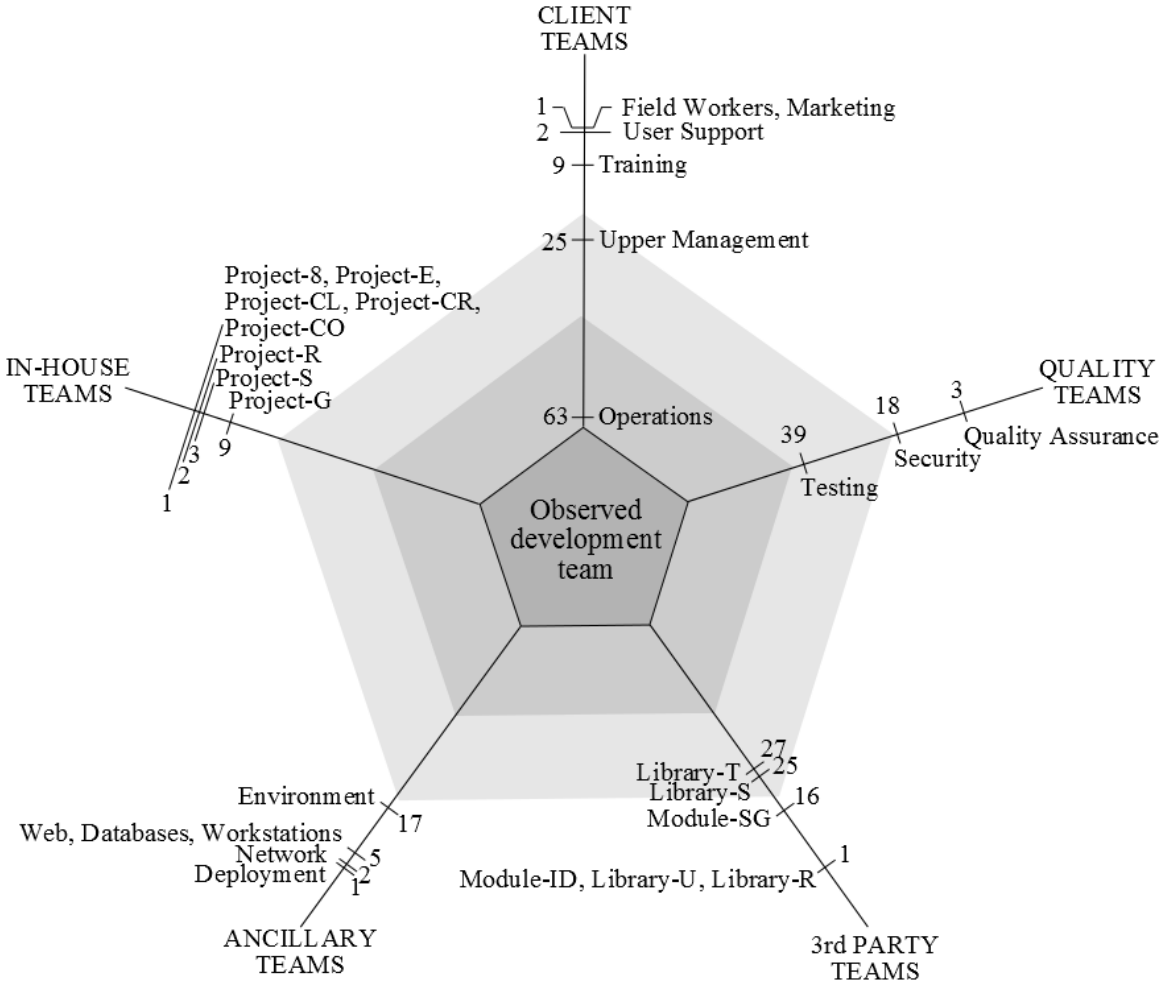

Figure B.1 Proximity of each external team with the observed development team. The number of interactions are posted on the axes.

The Figure 2 shows the occurrences of liaison interactions between two teams in which the observed development team was involved.

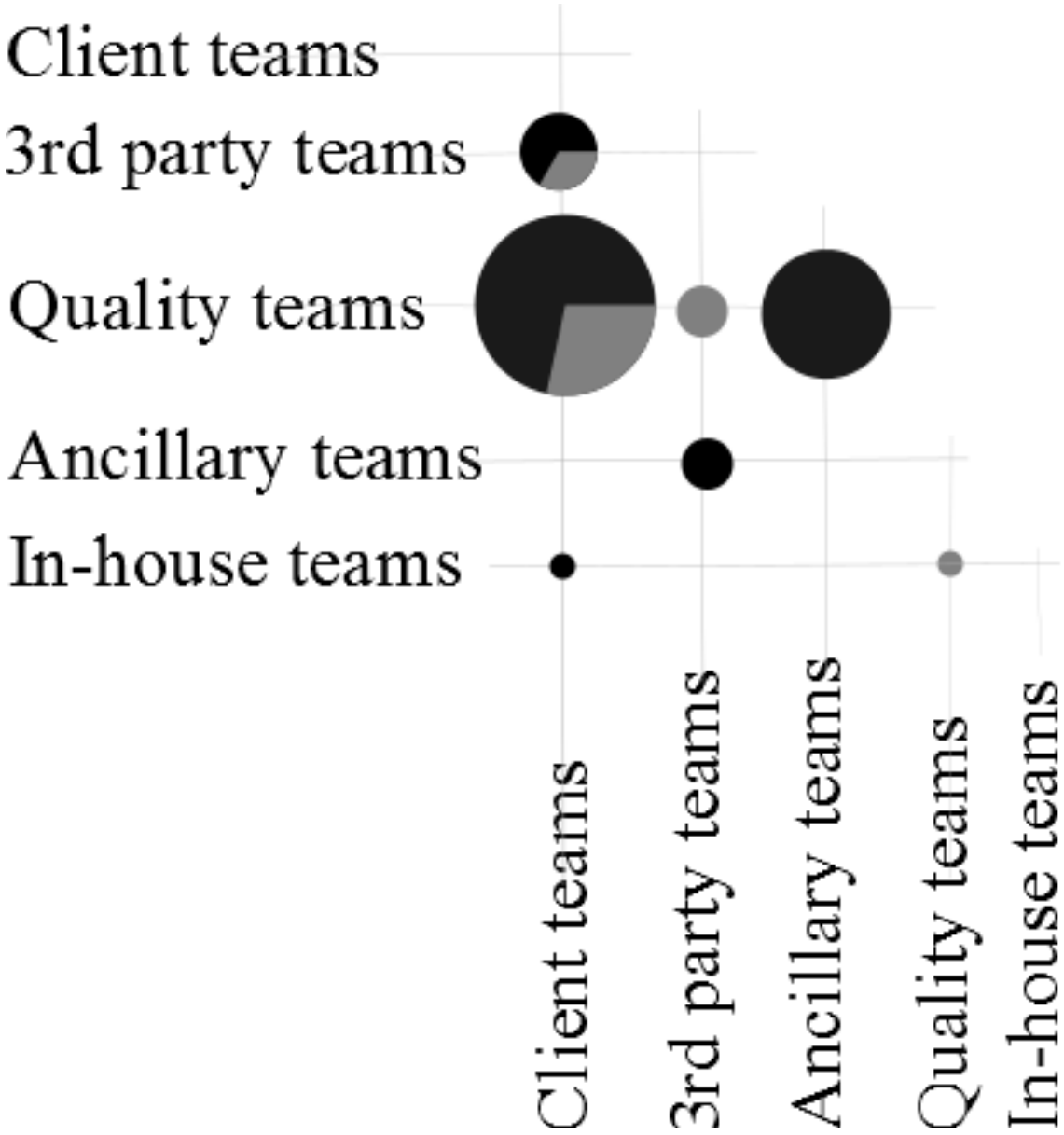

Figure B.2 Liaison interactions (knowledge brokering) between external team categories. Bubble size represents the amount of liaison interactions (from one to seven). Black color represents technical interactions, while grey color represents administrative interactions.

Figure 2 shows that the observed team is pivotal between the client and the quality group. These interactions include requirement clarifications, but also demands by testers to ensure that the initial data in the system are validated by the clients before testing can start. More details about the importance of the client/quality interactions can be found in the next section.

B.5.1 Interaction Purpose Examples

Tables 3 to 6 present a glimpse of the reasons through actual quotes from the development team. Each table covers one of the four types of interactions. The objective is to give an idea how a topic was associated with the appropriate interaction type and the appropriate external team.

B.5.2 Failure of the First Iteration of the Project

As stated earlier, the project observed had already been done once, but failed. A private communication with a manager who witnessed the failure of the first iteration but did not participate in the second one provides some details on the failure. According to the manager, the following factors may have caused the failure of the first iteration :

— Personality conflicts between the development team, the client teams, and the other 3rd party teams.
— Contractual issues between the organization and 3rd party developers. Contract negotiations dragged so long that the contracts were signed moments before the code was scheduled for production.
— Pressure from the project manager to filter interactions with the development team. This manager required that all requests had to be submitted directly to her, resulting in missed or misinterpreted messages.
— Documentation mostly incomprehensible by anyone outside the development team. Only the client teams' documentation could be reused as is.

While this statement is only supported by one witness, it still provides some insight as to why the project initially failed.

B.6 Discussion

This section discusses the results and proposes three solutions, along with their potential impact on quality.

Table B.3 Example Quotes Related to Team Demands.

Demands to	Quote
Client	*"There is a problem with [the client]. We need the configuration data and we have no answer from [the client]. I did some work on this, but I cannot finish by myself."* The team had to ask the client again for the configuration data.
Ancillary	*"Everything has been settled, except for the database configuration. We do not have the access rights [to the environment] to prepare this. [...] This configuration should be done by default ! It's like buying a car and not having a key !"* The team had to ask the environment setup team for the rights to change the database configuration.
In-house	*"We just receive an analysis from Project-G, which is about 60 pages. The analysis is very badly written and is essentially incomprehensible."* The team had to ask the Project-G team a clearer document in order to fulfill the analysis.

Table B.4 Example Quotes Related to Team Commitments.

Commitments to	Quote
Client	Upper Management has approved a new project with a high priority and a very aggressive calendar. It is likely that some developers from the development team will be assigned to this new project. The observed development team must finish their current project as soon as possible, as delays will be unacceptable for upper management.
Quality	*"What do we do if we find bugs ?"* Quality teams need development support during the developers' holiday, in August. The development team cannot go on holiday all at once : someone must stay in place to correct the bugs found by quality teams.
In-house	The development team must replace a function so it can support true/false/maybe values. This is in order to support Project-R, developed by another team, which will be deployed shortly after their current project ends.

Table B.5 Example Quotes Related to Team Coordination.

Coordination with	Quote
Client	The development team needs the business processes from the client so they can code the appropriate functionalities. But the client expects that the development team will explain how the software will work, and therefore adjust their business process in consequence. There is confusion as to whom is responsible for providing the business processes.
3rd party	The development team must discuss with Library-T support to determine which changes will be covered under the current contract and which changes will be charged extra to the project.
Quality	The development team pressures the testing team to start acceptance testing even though integrated testing is not finished. The testing team disagrees : the two teams will need to meet afterward in order to decide what to do. *"How can I start acceptance testing if integrated testing only reach 50% success ?"*

Table B.6 Example Quotes Related to Team Liaison.

Liaison from and to	Quote
From client to quality	The clients need to provide a description of their workflow for the testing team. The testing team are planning acceptance testing and want to design tests which reflects what the client does in its day-to-day work.
From quality to client	A client was assigned to the testing team in order to assist them in their work. However, the client assigned does not answer telephone nor email. The quality team needs to talk to him.
From quality to ancillary	The security team need access to the test environment in order to perform their tests. The Network team needs to open a port for the security team.
From in-house to quality	Testers need to know if they need to perform testing for the integration of Project-G within the current project. So far, the in-house team developing Project-G has not answered.

B.6.1 First Solution : Identification of the critical teams and client implication

This study shows that although interactions with external teams are important, some teams are more important than others. The frequency analysis shows that the interactions of the team loosely follow a Pareto distribution. Approximately 78% of external interactions (229 of 294 interactions) are made from about 28% of all teams contacted (8 of 29 teams). Based on the data in Figure 1, the distribution of these eight teams (categories of the corresponding team in brackets) are :

1. Operations [client team] : 63 interactions,
2. Testing [quality team] : 39 interactions,
3. Library-T [3rd party team] : 27 interactions,
4. Library-S [3rd party team] : 25 interactions,
5. Upper Management [client team] : 25 interactions,
6. Security [quality team] : 18 interactions,
7. Environment [ancillary team] : 17 interactions,
8. Module-SG [3rd party team] : 16 interactions.

While the other 21 teams have less than ten interactions each.

Therefore, project managers should try to identify the teams most likely to have an impact on the project beforehand, and ensure that communication channels with these teams are clear. In the case observed, this issue was somewhat alleviated by making the testing team sits in the same room as the developers towards the end of the project. They could not do the same with their 3rd party developers, which resulted in some serious issues. For example, communication problems with 3rd party support teams, coupled with poor service-level agreements (SLA), required multiple reworks of some simple change requests, each taking one month to perform (Lavallee & Robillard, 2015b).

The Pareto analysis shows that the clients, the Operations team, is by far the external team most contacted. However, the development project followed a waterfall approach, with fixed requirements. Why so many interactions are needed with the clients if the requirements are fixed since the beginning ? Many details and subtleties became evident as the developers progressed into the project. Some requirements have emerged or have changed very late during the project. Some of these changes were client requests, but others were tasks that the client needed to do.

For example, since this project is related to the update of an old package, some of the new databases must be updated with the data already in the old package. However, a lot of the

data in the old package are obsolete : dropdown menu items are no longer used ; database columns are no longer filled, etc. The developers cannot know these subtleties, and rely on clients to tell them which data to port to the new package, and which data to remove. In this case, the clients did not have the resources to do this task for the developers, which leads to multiple delays.

This shows that all projects, whether Agile or disciplined, require continuous interactions with stakeholders. But while Agile principles emphasize flexibility to clients' needs (*"our highest priority is to satisfy the customer"* (Runeson, Stefik, & Andrews, 2014)), this study shows that the clients must also be flexible to developers. Clients have obligations to fulfill. The domain knowledge of the clients was very important in this project. Some delays can be attributed to the unavailability of the client or to late responses to critical requests. The clients were required to provide many details about what the old package did, and why the old package worked that way, and on what the client wants for the new package. The clients' technical expertise was limited, but they knew very well their workflow and how they want the future application to merge with this workflow. For project managers, we propose that any client/provider agreement ensures that the client is willing to actively help developers. For example, the simple task of seeding a database with its initial dataset is difficult to plan ahead : it can be done once the database structure is completed, using data that is usually provided by the client. In this study, delays in obtaining clients responses have led to delays in database configuration, which caused the tests to start late, and ultimately to be shorter than planned.

Previous Agile studies have used client delegates to ensure coordination between the real client and the development teams (Lee, 2008 ; Paasivaara & Lassenius, 2011). Delegation of client duties can prevent constant interruption of the workflow of developers. One study assigned each team with "a support person".

> Supporting the customer using all the solutions that the team had to provide was a critical task. This required a vast amount of knowledge of all moving pieces. Before we had the support person, the customers interrupted subject matter experts [i.e. developers] directly. The subject matter experts typically dealt with too many support requests and ended up context switching in and out of the tasks at hand. (Lee, 2008)

The impact on quality in the case observed is mainly transcribed in terms of delays. The failure to provide answers in a due manner to the questions of the development team led to multiple delays. In this case, these delays cause the testing phase to be greatly reduced. In addition, some code written by 3rd parties could not be reviewed in time for delivery and

was included in the codebase as-is. By the accounts of the developers themselves, a lengthy support process will be necessary post-delivery to ensure that all the issues are sufficiently smoothed out.

B.6.2 Second Solution : Developers as Knowledge Brokers within the MTS

Figure 3 illustrates the two-ways interactions between external teams and the development team, based on the team demands and team commitments found in Table 2. For example, in the interactions between the development team and the client teams, the development team had 30 commitments (grey arrow) toward the client teams, while the client teams answered 43 demands (white arrow) from the development team.

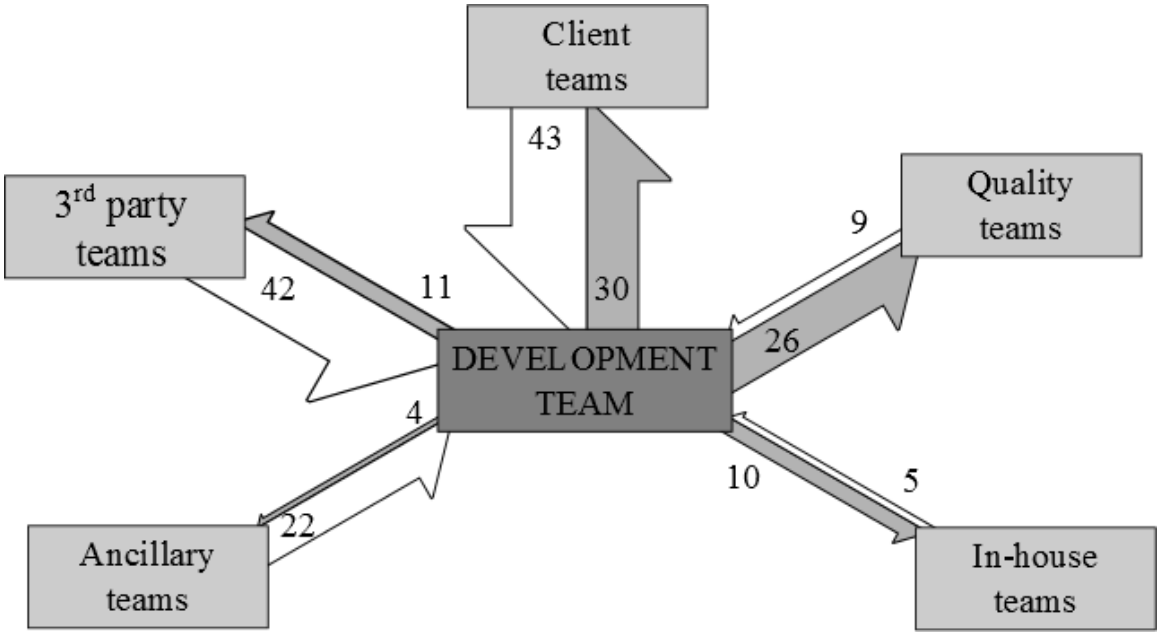

Figure B.3 Chain of commitments between teams. White arrows indicate answers to development team demands. Grey arrows indicate team commitments that the development team must fulfil.

The left hand side of Figure 3 shows the team categories with a majority of demands from the development team (large white arrows) , while the right hand side shows the team categories with a majority of commitments from the development team (large grey arrows). The client teams, being fairly balanced in demands and commitments, remains in the middle. What should be seen from Figure 3 is that demands flows from the left to the right. Ancillary teams fulfill developers' requests, so that developers can fulfill quality teams' requests.

Here is an example taken from the interactions observed. The quality teams needed many test

environments in order to perform their work (acceptance environment, load testing environment, etc.). The development team was therefore committed into building these environments and ensuring that they were coherent with the latest available versions of the package and that they were stable enough to support test activities. While they could do some of the work themselves, they needed the support of the environment setup team, an ancillary team. However, the environment setup team did not fulfill its commitment appropriately, causing a number of issues to the development team. These environmental issues cause the development team to fail in some of their commitments toward the quality teams, causing delays and ultimately, the cancellation of some of the test activities.

Some of these relationships might seem self-evident, but others might not be as well-known. As presented in our previous paper (Lavallee & Robillard, 2015b), managers should be wary of other projects imposing changes to the current project. Project managers should also ensure that all relevant teams (3rd party teams, ancillary teams) are ready to help the development team. In the case presented above, many issues stemmed from poor communications between the development team and the environment team.

The role of the development team in this case is that of a broker. Developers need to redirect the requests they receive to the appropriate team. To take an analogy from the TCP/IP protocol, the development team is the default gateway for the external teams. External teams needing something related to the project will ask the developers first, which will then redirect the team to the appropriate resource when necessary.

This is especially true of the relationship between clients and testers. Clients and testers do not know how the application was built, who was contacted to code the software, what are the dependencies. They are mostly conscious on what they see on their end. When something goes wrong, their only contact is the development team. Clients and testers need some answers but do not know who to ask ; developers know and must assist them.

For project managers, this study shows that testers cannot work efficiently if they are kept completely isolated from the development team. Testers need to ask many questions in order to perform their work, and these questions must be efficiently relayed to the appropriate external team. In the case studied, toward the end of the project, management had the testing team sits directly with the development team. Their goal was to diminish bug resolution times, but it also helped the testers in the setup of the different testing phases and testing environment (integration, acceptance, load, and deployment). The same can be applied to clients. While it might not make sense to put the client in contact with every relevant external team, clients' questions can be distributed by the developers to the relevant external teams.

The need for knowledge brokers have been identified in the literature (Asencio et al., 2012 ;

Lee, 2008).

> Brokers are those individuals who link disconnected subgroups. [Another study] found that system-level coordination is achieved more efficiently when certain key individuals connect different subgroups as opposed to when all individuals are directly connected to one another. Complex MTSs may be more efficiently coordinated if certain individuals act as ambassadors by connecting their team to others within the system. (Asencio et al., 2012)

The question of whom to assign to the role of knowledge broker varies from study to study however. It should be someone who have a widespread knowledge of the system (Lee, 2008 ; Smits & Pshigoda, 2007). It is however unnecessary to have a broker between each team. As presented in the previous section, teams with a potential critical impact on the development team's work should be identified. Knowledge brokers can therefore be assigned only for those critical teams (Davison, Hollenbeck, Barnes, Sleesman, & Ilgen, 2012). Minor teams and modules could be more isolated from the development team under study.

The impact on quality rests on the fact that the development team does not work in isolation. There are many other teams working indirectly on the project which require adequate support to perform their work. Here are a few examples :

— Testers need to obtain real data from the clients in order to perform tests that can be relatable to what goes on in reality.
— Testers need working testing environment with up-to-date code in order to perform adequate tests.
— Third party support teams need to know the type of tests to be performed in order to ensure that their infrastructure will support these tests (e.g. stress testing or security testing cloud storage services). Etc.

Failure to relay the needs of one external team to another can lead to the cancellation of important activities.

B.6.3 Third Solution : Managing Technical and Administrative Interactions

Before this study, the organization managers and the development team were convinced that their meetings were mostly administrative ; discussing deadlines, budget and resources. Observations proved that most of these discussions were actually technical and involved bugs, issues, design, solutions, etc. It is therefore not surprising that most of the interactions with external teams are also technical in nature.

But this information should not be exchanged only from one manager to another. This study's suggestion to project managers is to make sure that developers in different teams are able to talk to each other. Managers have a tendency to protect their developers from outside interference, and it is good to keep an eye on that, as this was an issue with Upper Management in this case (Lavallee & Robillard, 2015b). But developers also need to be able to obtain technical information from other teams, and to plan technical solutions and strategies together.

The literature recommends a layered structure where the lower levels are able to share technical details, while the higher levels are able to share the administrative big picture (Lee, 2008 ; Smits & Pshigoda, 2007).

> Cross team knowledge sharing is difficult. [...] After 1.5 year into practicing Agile, we found the best way to mitigate, is to have weekly Scrum of Scrums (S2) meetings and daily tech leads stand-up meeting. For the stakeholders, Scrum of Scrums of Scrums (S3) was very helpful to get things prioritized. (Lee, 2008)

The impact on quality is that technical issues facing the whole codebase are not discussed anywhere. Individual teams might be aware of the issues, but without a platform to discuss and voice their concern, these issues remain latent and unaddressed. Organizations have administrative strategies, where managers discuss future plans and projects, but how many of them have technical strategies, where engineers can discuss future maintenance challenges and issues ?

B.6.4 Threats to Validity

A threat to the validity with the use of a single study is the generalizability of its conclusions. The objective of this study was however not to build a theory applicable to all software development projects, but to identify new potentially interesting practices and issues from the industry. While this study is limited to a single case, it nonetheless presents new qualitative and quantitative data showing the role of clients during development, the role of developers as knowledge brokers, and the importance of technical coordination at the MTS level.

B.7 Conclusion

This ethnographic study shows the impact interactions within the multi team system can have on project success. Due to the single study nature of this research, future research should look into whether the three propositions presented herein are relevant in other cases.

1. Identify the external teams most likely to have an impact on the development project, based on a Pareto analysis and ensure proper communication channels with the most important ones. Otherwise, slow communications will cause delays during development, which might result in rush development work and shorter testing time.
2. Ensure that knowledge brokers exist within the development team to redirect requests from one external team to the proper other external team. Otherwise, some activities with an indirect impact on the development project (e.g. testing) might be in jeopardy.
3. Ensure that discussion platforms at the multi-team level are not limited to administrative issues. Technical solutions and strategies must be discussed between teams. Otherwise quality issues affecting the whole codebase could remain unaddressed.

Project managers should be aware of the impact of multi team systems on their projects. From a disciplined, plan-driven approach, to an Agile, people-driven approach, there is a need for an integrated, organization-driven approach, where the team is integrated within its organization. Teamwork experts have recommended breaking the isolation between individuals in order to ensure that the whole team works together. We should now see if these recommendations hold at the organization level, in order to ensure that the whole organization works together. There might be no 'I' in 'team'. But how much place for 'us' and 'them' are we willing to work with within the organization ?

B.8 Acknowledgments

This research would not have been possible without the agreement of the company in which it was conducted, which prefers to stay anonymous, and without the generous participation and patience of the software development team members from whom the data were collected. To all these people, we extend our grateful thanks.

This work was supported by the Natural Sciences and Engineering Research Council of Canada, under grant number A-0141.

B.9 References

Asencio, R., Carter, D. R., DeChurch, L. A., Zaccaro, S. J., & Fiore, S. M. (2012). Charting a course for collaboration : a multiteam perspective. Translational Behavioral Medicine, 2(4), 487-494. doi : 10.1007/s13142-012-0170-3

Bannerman, P. L., Hossain, E., & Jeffery, R. (2012, 4-7 Jan. 2012). Scrum Practice Mitigation of Global Software Development Coordination Challenges : A Distinctive Advantage ? Paper

presented at the System Science (HICSS), 2012 45th Hawaii International Conference on.

Brewer, J. D. (2000). Ethnography : Open University Press.

Chau, T., & Maurer, F. (2004). Knowledge sharing in agile software teams Logic versus approximation (pp. 173-183) : Springer.

Cohn, M. (2007). Advice on Conducting the Scrum of Scrums Meeting. Retrieved 2015-08-21, 2015, from https ://www.scrumalliance.org/community/articles/2007/may/advice-on-conducting-the-scrum-of-scrums-meeting

da Silva, F. Q. B., Franca, A. C., Suassuna, M., de Sousa Mariz, L. M. R., Rossiley, I., de Miranda, R. C. G., Espindola, E. (2013). Team building criteria in software projects : A mix-method replicated study. Information and Software Technology, 55(7), 1316-1340. doi : 10.1016/j.infsof.2012.11.006

Davison, R. B., Hollenbeck, J. R., Barnes, C. M., Sleesman, D. J., & Ilgen, D. R. (2012). Coordinated action in multiteam systems. Journal of Applied Psychology, 97(4), 808.

Guzzo, R. A., & Dickson, M. W. (1996). Teams in organizations : recent research on performance and effectiveness. Annual Review of Psychology, 47, 307-338. doi : doi :0.1146/annurev.psych.47.1.307

Guzzo, R. A., & Salas, E. (1995). Team Effectiveness and Decision Making in Organizations. San Francisco : Jossey-Bass.

Humphrey, W. S. (2000). The Team Software Process (TSP) (pp. 51). Pittsburgh, PA : Software Engineering Institute.

Kiani, Z. U. R., Mite, D., & Riaz, A. (2013). Measuring Awareness in Cross-Team Collaborations–Distance Matters. Paper presented at the Global Software Engineering (ICGSE), 2013 IEEE 8th International Conference on.

Lanaj, K., Hollenbeck, J. R., Ilgen, D. R., Barnes, C. M., & Harmon, S. J. (2013). The double-edged sword of decentralized planning in multiteam systems. Academy of Management Journal, 56(3), 735-757. Landsberger, H. A. (1958). Hawthorne Revisited : Cornell University.

Lavallee, M., & Robillard, P. N. (2015a). Planning for the unknown : lessons learned from ten months of non-participant exploratory observations in the industry. Paper presented at the 2015 IEEE/ACM 3rd International Workshop on Conducting Empirical Studies in Industry (CESI), 18 May 2015, Los Alamitos, CA, USA.

Lavallee, M., & Robillard, P. N. (2015b). Why Good Developers Write Bad Code : An Observational Case Study of the Impacts of Organizational Factors on Software Quality. Paper presented at the International Conference on Software Engineering, Florence, Italy.

Lee, E. C. (2008). Forming to Performing : Transitioning Large-Scale Project Into Agile.

Lethbridge, T. C., Sim, S. E., & Singer, J. (2005). Studying software engineers : data collection techniques for software field studies. Empirical Software Engineering, 10(3), 311-341. doi : 10.1007/s10664-005-1290-x

Looney, J. P., & Nissen, M. E. (2007, Jan. 2007). Organizational Metacognition : The Importance of Knowing the Knowledge Network. Paper presented at the System Sciences, 2007. HICSS 2007. 40th Annual Hawaii International Conference on.

Maranzato, R. P., Neubert, M., & Herculano, P. (2011). Moving back to scrum and scaling to scrum of scrums in less than one year. Paper presented at the Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion, Portland, Oregon, USA.

Marks, M. A., DeChurch, L. A., Mathieu, J. E., Panzer, F. J., & Alonso, A. (2005). Teamwork in Multiteam Systems. Journal of Applied Psychology, 90(5), 964-971. doi : 10.1037/0021-9010.90.5.964

Martini, A., Pareto, L., & Bosch, J. (2013). Improving businesses success by managing interactions among Agile teams in large organizations. Paper presented at the 4th International Conference on Software Business, ICSOB 2013, June 11, 2013 - June 14, 2013, Potsdam, Germany.

Mathieu, J. E., Marks, M. A., & Zaccaro, S. J. (2001). Multi-team systems. In N. Anderson, D. Ones, H. K. Sinangil & C. Viswesvaran (Eds.), International handbook of work and organizational psychology (pp. 289-313). London : Sage.

Mundra, A., Misra, S., & Dhawale, C. A. (2013). Practical Scrum-Scrum team : Way to produce successful and quality software. Paper presented at the Computational Science and Its Applications (ICCSA), 2013 13th International Conference on.

Paasivaara, M., & Lassenius, C. (2011). Scaling scrum in a large distributed project. Paper presented at the Empirical Software Engineering and Measurement (ESEM), 2011 International Symposium on.

Paasivaara, M., Lassenius, C., & Heikkila, V. T. (2012). Inter-team coordination in large-scale globally distributed scrum : Do scrum-of-scrums really work ? Paper presented at the 6th ACM-IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM 2012, September 19, 2012 - September 20, 2012, Lund, Sweden.

Porck, J. P. (2013). No Team is an Island : An Integrative View of Strategic Consensus between Groups. (Ph.D.), Erasmus University Rotterdam, Rotterdam, Netherlands. (EPS-2013-299-

ORG)

Runeson, P., Stefik, A., & Andrews, A. (2014). Variation factors in the design and analysis of replicated controlled experiments : Three (dis)similar studies on inspections versus unit testing. Empirical Software Engineering, 19(6), 1781-1808. doi : 10.1007/s10664-013-9262-z

Santos, V., Goldman, A., & de Souza, C. B. (2015). Fostering effective inter-team knowledge sharing in agile software development. Empirical Software Engineering, 20(4), 1006-1051. doi : 10.1007/s10664-014-9307-y

Scheerer, A., Bick, S., Hildenbrand, T., & Heinzl, A. (2015). The Effects of Team Backlog Dependencies on Agile Multiteam Systems : A Graph Theoretical Approach. Paper presented at the System Sciences (HICSS), 2015 48th Hawaii International Conference on.

Scheerer, A., Hildenbrand, T., & Kude, T. (2014, 6-9 Jan. 2014). Coordination in Large-Scale Agile Software Development : A Multiteam Systems Perspective. Paper presented at the 2014 47th Hawaii International Conference on System Sciences.

Smits, H., & Pshigoda, G. (2007). Implementing scrum in a distributed software development organization. Paper presented at the Agile Conference (AGILE), 2007.

Strauss, A. L. (2003). Qualitative Analysis for Social Scientists : Cambridge University Press.

Sutherland, J., Schoonheim, G., & Rijk, M. (2009). Fully distributed scrum : Replicating local productivity and quality with offshore teams. Paper presented at the System Sciences, 2009. HICSS'09. 42nd Hawaii International Conference on.

Tartaglia, C. M., & Ramnath, P. (2005). Using open spaces to resolve cross team issue [software development]. Paper presented at the Agile Conference, 2005. Proceedings.

Verner, J. M., Brereton, O. P., Kitchenham, B. A., Turner, M., & Niazi, M. (2012). Systematic literature reviews in global software development : A tertiary study. Paper presented at the 16th International Conference on Evaluation and Assessment in Software Engineering, EASE 2012, May 14, 2012 - May 15, 2012, Ciudad Real, Spain.

ANNEXE C ARTICLE 2 : WHY GOOD DEVELOPERS WRITE BAD CODE : AN OBSERVATIONAL CASE STUDY OF THE IMPACTS OF ORGANIZATIONAL FACTORS ON SOFTWARE QUALITY

Type de publication	Référence complète
Conférence	*Why Good Developers Write Bad Code : An Observational Case Study of the Impacts of Organizational Factors on Software Quality*, **Mathieu Lavallée** et Pierre N. Robillard, International Conference on Software Engineering (ICSE), 2015, ACM SIGSOFT Distinguished Paper Award, doi :10.1109/ICSE.2015.83

C.1 Abstract

How can organizational factors such as structure and culture have an impact on the working conditions of developers ? This study is based on ten months of observation of an in-house software development project within a large telecommunications company. The observation was conducted during mandatory weekly status meetings, where technical and managerial issues were raised and discussed. Preliminary results show that many decisions made under the pressure of certain organizational factors negatively affected software quality. This paper describes cases depicting the complexity of organizational factors and reports on ten issues that have had a negative impact on quality, followed by suggested avenues for corrective action.

C.2 Introduction and Related Work

Current studies on software development environments have focused on improving conditions for the software development team : implementation of appropriate processes, hiring skills, use of appropriate communication tools and equipment, etc. Among these software development conditions are factors from the organization. However, while such factors are perceived as important, there is little empirical evidence of their effect on the quality of software products [1], [2]. It can therefore be useful to present empirical data on the effects of organizational factors on the development team. What if a project's success is dictated

more by the constraints imposed by the organization than by the expertise and methodology provided by the team ?

Organizational factors include a wide range of contextual factors with a potential impact on the success or failure of software development projects. Early empirical work done in 1998 by Jaktman [3] showed that organizational values can have an impact on software architecture. She presents two interesting cases : the first revolves around a company who made software development dependent on the marketing department. As Jaktman writes : "This resulted in software quality activities being given a low priority" [3]. The outcome was a poor architecture, with a lot of code duplication and high error rates. This jeopardized the company's future, because "maintenance problems prevented new products from being released quickly" [3].

The second case presents a company who was officially committed to quality, but was ambivalent about how to implement it. Management promoted various quality approaches, but ultimately resources failed to materialize. The "frequent changes to product priorities affected the code : incomplete implementation of software changes left dead code and code fragments" [3].

These two cases show the possible relationship between high-level organizational values and low-level product quality. Unfortunately, studies reporting on the impact of organizational factors on software development are scarce. As Mathew wrote in 2007, "it is surprising to note that academic research has largely ignored investigation into the impact of organisational culture on productivity and quality" [1]. This paper answers the need for more empirical data through the presentation of observed cases describing how some organizational factors can impact software quality. Section II describes the context and the approach used to collect data for the study. Section III presents the results of the study, while Section IV discusses the implications of the findings. Finally, Section V presents some concluding remarks and observations on organizational factors.

C.3 Methodology

C.3.1 Context

The study was performed on a large telecommunications company with over forty years in the industry. Throughout the years, the company has developed a large amount of software, which must be constantly updated. This study follows one such project update.

The outcomes of this study are based on observation of a software development team involved in a two-year project for an internal client. The project involved a complete redesign of an

existing software package, which we will call the Module, used in the company's internal business processes.

The technical challenge of the Module is its distributed nature : it includes legacy software written in COBOL, Web interfaces, interactions with mobile devices and multiple databases. Its purpose is to manage work orders. To do this, it needs to extract data from multiple sources within the enterprise (employee list, equipment list, etc.) and send it to multiple databases (payroll, quality control, etc.).

The project was a second attempt to overhaul this complex Module. A first attempt was made between 2010 and 2012 but was abandoned after the fully integrated software did not work. Because this project was a second attempt, many specifications and design documents could be reused. Accordingly, the development has essentially followed a waterfall process, as few problems were expected the second time around. This second attempt began in 2013 and finished in December 2014.

C.3.2 Data Collection

The study is based on non-participant observation of the software development team's weekly status meetings. These meetings consisted of a mandatory all-hands discussion for the eight developers assigned mostly full-time to the project, along with the project manager. They also involved, as needed, developers from related external modules, testers, database administrators, security experts, a quality control specialist, etc. The meetings involved up to 15 participants, and up to five stakeholders through conference calls.

The team discussed the progress made during the previous week, the work planned for the coming week and obstacles to progress. The problems raised concerned resources and technical issues. Few decisions were taken at these meetings, the purpose being to share the content of the previous week's discussions between developers, testers, managers, etc.

A round-table format was used, where each participant was asked to report on their activities. The discussions were open and everyone was encouraged to contribute. When a particular issue required too much time, participants were asked to set another meeting to discuss it. Meetings lasted about an hour. The data presented in this study were collected over ten months during the last phase of the two-year project. It is based on 36 meetings held between January and November 2014. The same observer attended all the meetings and took note of who was involved in each interaction, the topic being discussed, and the outcome. An interaction is defined as a proposition or argument presented by one or more team members. A typical interaction takes between 5 seconds (e.g. "yes, I agree with you") and 30 seconds

(e.g. the presentation of a solution by many team members). A topic (e.g. “should we deploy on Thursday or on Saturday ?”) could be discussed over multiple interactions (e.g. “yes because. . . ” and “no because. . . ”), and sometimes ended with an outcome (e.g. “we will deploy on Saturday”).

C.3.3 Validation

The weekly status meetings provided a lot of information about the developers and the product. However, they provided only the perspective of the meeting participants. To better understand how the organization operates, the observer interviewed two managers from different departments (operations and marketing) on July 22nd in order to obtain their views. This validation was performed in a semi-structured interview. The observer asked the two managers if they agreed or disagreed with the preliminary conclusions made so far.

A second validation was performed at the end of the study. Since it was a non-participant study, the subjects observed were not aware of the conclusions reached by the researchers. A presentation was therefore made in December 2014 in order to confirm or refute the conclusions of the researchers. Results of the validation, when conflicting with our interpretations, are presented in the following and at the end of each case discussion.

Another validation was performed with an unrelated public sector software development department in December 2014. The thirteen workers were presented with the conclusions from this study and asked to rate whether the conclusion applied to their situation. For each of the cases identified in this study, the participants confirmed the occurrence of the issues within their organization.

C.3.4 Threats to Validity

The main issue was that the non-participant observer was not familiar with the technical terms used by developers, a common issue for non-participant observation studies [4]. The observer was present in the organization only during the weekly meetings, which means he was not aware of the interactions taking place outside the meetings. The observer was therefore not always up to date on what was going on. However, several problems recurred many times during the seven-month study, and were thus easy to follow and understand. The issues presented here are glaring problems that were not subject to interpretation and were all confirmed by the two managers during the validation interview.

As a single case study, generalization of the issues identified can be disputed. However, given the current small number of non-participant observational studies in software engineering,

it is surmised that the data collected in this study remain interesting for future researchers. Additionally, given that the issues presented were confirmed during the validation process and in the literature review, it is believed that these issues are relevant for multiple contexts.

C.4 Observations

The Core Team consisted of one manager and eight developers. However, it is surprising how many people are needed to interact with the core team to understand, develop, test and deploy the Module successfully. As shown in Figure 1, the core team interacted with no fewer than 45 people in at least 13 different teams during the ten month study.

Figure 1 also shows the relationship between the Core Team (shown in the middle) and the other teams involved in the project. Each team is represented by an oval shape with a descriptive label and contains the numbers and titles of the interacting members. For example, the Quality Assurance team, third from the top, clockwise, involved a quality tester and a quality manager interacting with the Core Team.

The innermost oval shape represents the Core Team, whose members were present at most meetings. The frequency of interactions with the Core Team is described by the dashed concentric circles. The second concentric circle shows the people who came to most (i.e. at least half) of the meetings. The third concentric circle represents the people who came to only one or two meetings, or who were present through a conference call. The outer concentric circle represents people who did not attend the weekly meeting during the study, but who were referred to during discussions and contacted outside the meetings. For example, the quality tester from the Quality Assurance team attended seven meetings, and the Core Team members referred a few times to the quality manager's requests during meetings. As the study focused on late phases of the development project, the teams in the second circle are mostly related to testing activities. In the early stages of the project, the operations department, being the internal client of the project, was much more involved.

Table I presents the organizational issues and their impacts on quality as observed during the study and confirmed during the validation. The following subsections detail each issue. The generic cases for each issue are discussed in Section IV.

C.4.1 The "documenting complexity" issue

This company has been in the telecommunications business for many decades. The objective of their software development department is to support the main business process and therefore the main concern is to provide software applications. Much of the existing code is

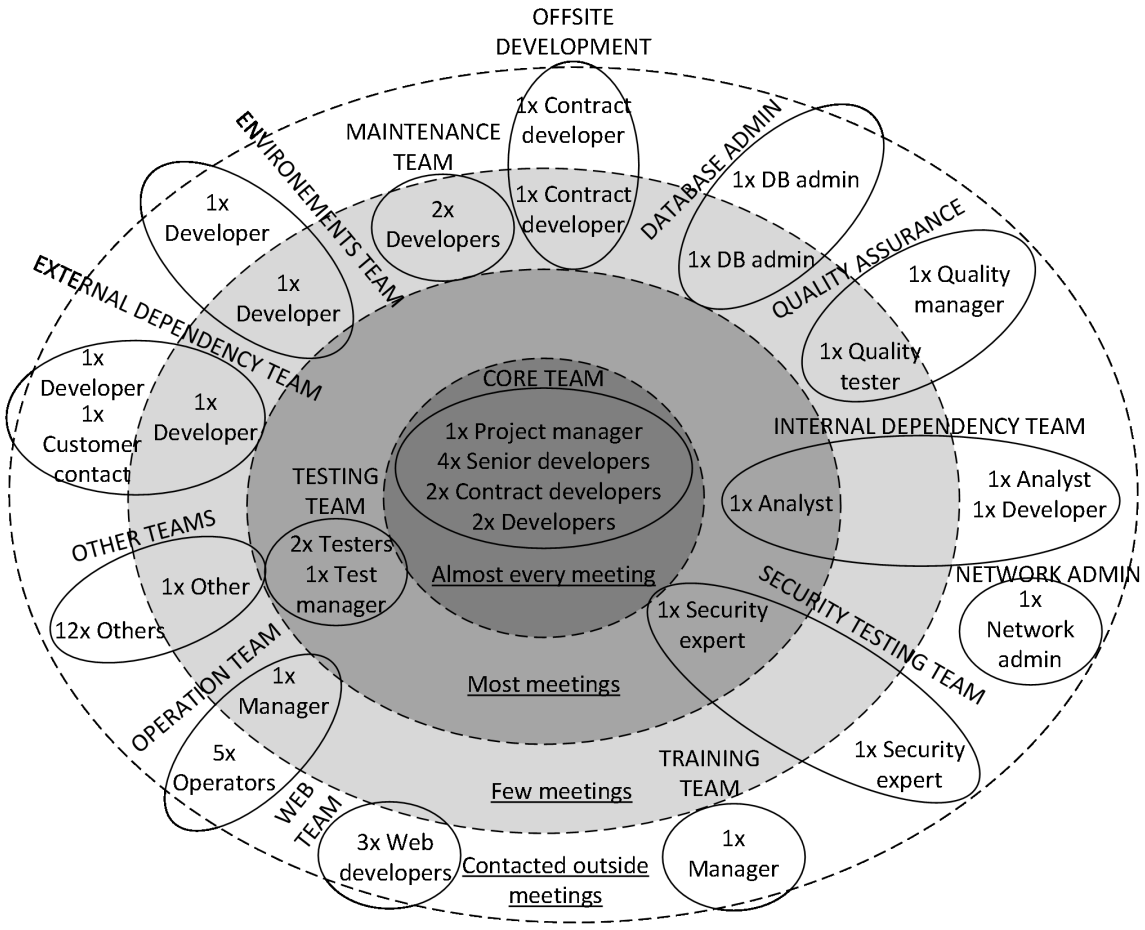

Figure C.1 People and teams involved in the project, and how often they were in contact with the core development team.

poorly structured and difficult to understand.

Over the years, the company's legacy code has become very large, very complex and, like a lot of older software, is poorly documented.

Over time, the organization has encouraged software engineering practices to ensure that new software has sufficient documentation for maintenance and user support teams. However, most software modules are still poorly documented. To make matters worse, many developers have since left the company.

For the Module developed during this study, they have extensive documentation – thousands of pages – from the previous attempt to develop the software. This documentation is incomplete, and its quality is questionable as the previous implementation failed. Nobody knows exactly how it all works because of the size of the documentation. Despite their attempts to understand how the Module works, they assumed that some features are going to fall through

the cracks.

Therefore, during the final stage of development, a senior developer was assigned to what they called the “crack”, or the list of features that were initially overlooked. Some of these missing features were identified as "must-have" for operators and could very well be among the reasons for the failure of the previous project. For example :

— A number of daily maintenance scripts whose purpose is to maintain the database in a healthy state.
— A rarely-used user interface which aims to support the main business process when the main server is inaccessible due to connection issues.
— An archive feature used for quality monitoring.

These features escaped scrutiny, either because they were invisible to the operators of the Module, or because they were rarely used. The difficulty of documenting the complexity of the Module was that these functions were not noticed until late in the development.

The impact on quality was that some of these features were not fully implemented. For example, the rarely-used user interface will not have all the necessary features, but will be limited to a barebones interface. Other features that had fallen into the "crack" and were not identified as "must-have" were simply ignored. Developers have argued that they will be implemented in a future update.

C.4.2 The "internal dependencies" issue

The Module has many dependencies with other modules within the company’s extensive legacy code. In addition, many other modules are simultaneously undergoing perfective maintenance. These interdependencies cause many issues since changes in the Module may affect other modules, and vice versa.

Priority decisions on module changes depend largely on the deployment schedules. For a module deployed later, the schedule must take into account the changes implemented on the previous modules. However, technical or business constraints may force the rapid deployment of certain features. For example, changes in the database must often be propagated early to ensure that the data structure will be coherent in all the linked modules.

Change requests from these internal dependencies have an impact on the quality of the Module. These change requests are often implemented quickly and haphazardly, as they are rarely planned or budgeted. Even in cases where an analysis is performed, allowing the allocation of additional resources, deployment schedules are rarely modified, resulting in greater deadline pressures.

Table C.1 Organizational Issues with an Impact on Quality as Observed during the Study

Documenting complexity	Large amount of old poorly documented software. Difficulty in managing large amounts of documentation.	Some requirements are discovered late in the development process. Some undocumented features were not implemented; others were patched together very quickly.	Once a project is completed, the team must ensure that the "What" and "Why" of each software item are properly documented.
Internal dependencies	Lots of interdependencies between software modules. Conflicts between projects on the scheduling of deployment.	Compatibility between modules is patched quickly and haphazardly.	In the cases of parallel development of inter-dependent software modules set up a negotiation table to solve conflict between the development teams.
External dependencies	Long-term dependencies to third-party libraries. Change requests to third parties results in costly delays.	Contractual obligations with third parties force poor quality design choices in order to minimize change requests.	Make sure that the development team is aware of the CMMI-ACQ or ISO12207 processes for negotiating with third parties.
Cloud storage	Contractual obligations with third parties do not support vulnerability testing.	Vulnerability testing is performed late and quickly.	Make sure that testers are involved when negotiating with a third party for a potentially vulnerable software component.
Organically grown processes	Processes emerge as the need arise, usually after crises. Defects found by users are documented but do not reach developers. Environment setup process unclear for developers. Patches in testing follow a stringent process, even when nothing works.	Frontiers between processes hinder information exchanges; developers must work with missing details.	Plan organization-wide process reviews to detect isolated processes and to promote information flows between processes.
Budget protection	Cheaper to build a wrapper instead of solving the issue once and for all. A software item has twelve such wrappers.	Developers patch instead of fixing issues because fixing would cost too much.	Planned special budget items to support long lasting corrections or corrections that are likely to benefit many modules.
Scope protection	Explicit calls by team members to "protect the scope" of the project. Requirements are transferred to other projects as much as possible.	Project constraints means that teams must protect the scope of their project against change requests from other teams.	Projects with strict deadlines are risky, and should be carefully monitored to avoid last minute unplanned activities.
Organizational politics	Change request refused because the team did not contact the right person. Environmental issues resolved when the right manager was contacted. Inter-module issues resolved when upper manager applied pressure.	Managers and developers obtain better results by calling in favors from outside the software engineering process. Novices who do not know who to contact to obtain information will produce worse software.	Team members should maintain a careful balance between the flows of information within formal development processes and informal human interactions.
Human resource planning	The two developers assigned since the beginning of the project are contractual developers whose contract is about to expire.	The team will lose all knowledge of the beginning of the project, including details on the requirements and analysis phases. Documentation of the Module will be incomplete and/or inaccurate.	Team members should make sure that knowledge is appropriately distributed amongst them. For example, pair programming is a practice which can promote knowledge sharing.
Undue pressure	Managers and senior developers give direct orders and threats to the team.	The issuers of the orders and threats might not have all relevant information which could results in ill-defined priorities. Bypassing the normal team structure undermines its decision-making process.	Any intrusion into the team dynamics by outsiders should be done very carefully.

C.4.3 The "external dependencies" issue

The Module has many dependencies with third party modules, which are based on contract agreements. The Core Team must deal with these modules on a case-by-case basis, since each third party module has different contracts, practices and processes. Change requests ("CRs") are defined differently amongst the various third party contracts. For example, a minor change request on an external module might be considered billable for one third party supplier, or as covered by the maintenance contract for another. To make things worse, the quality of the work performed by third party suppliers fluctuates wildly. In many cases, the CR work had to be sent back to the supplier for rework, because it was not coherent with specifications, or of poor quality. Communication problems with third party suppliers were confirmed by many of the managers who had to deal with them.

These external dependency constraints force the team to make decisions that may have a significant impact on quality, in order to reduce costs and lead times. One practical example is the transformation of a variable type at the boundary between the Module and an external third party module from an enumeration to a free text type. From a quality standpoint, this is likely to introduce problems since the free text type cannot be validated, while the enumeration is self-documenting by the bounded and defined range of values.

So why was this decision made ? Because each time an enumeration value needed to be modified, the third party supplier asked for a billable CR, which meant that changes to the enumeration type soon became prohibitively lengthy and expensive. In the short term, it was deemed cheaper to use a free text-type variable, even if it meant causing future hard-to-diagnose software defects.

C.4.4 The "cloud storage" issue

While not strictly pertaining to cloud computing, this issue is related to code hosted by other entities than the owner of the Module. In this observed case, some contractual agreements with these third party entities did not include all the planned test activities. This meant that some tests could not be performed until the contracts were renegotiated with the relevant clauses.

For example, the initial agreement did not allow vulnerability testing. The vulnerability tests were aimed at finding technical issues, like SQL injections, as well as overall robustness evaluation, like resistance to Denial of Service (DoS) attacks. The risk to quality is potentially huge. Without these tests, it is possible that a vulnerable code segment could remain in the Module, exposing the company to liability should a customer's account be compromised.

Vulnerability testing was thus performed very late in the development process, as the contracts needed reviewing by both parties. A quick superficial test found over 350 vulnerabilities in one third-party library which was already used in production, prompting one security expert to say that "this was certainly written by junior developers".

C.4.5 The "organically grown processes" issue

The best practices of software engineering appeared in parallel with the company's growth. After all, the company has been around for decades, or about as long as software engineering itself. Software engineering processes were thus introduced organically, as the need arose or became apparent in a crisis. This resulted in "islands of formality" : some software development teams within the organization follow a very clear and well-defined process, while others still work in a mostly ad hoc fashion.

For example, the quality control team follows a very well-defined process. The development team is more informal, but has a set of standard practices. And yet both processes evolved organically, so that that the quality control process is very different from the development process. This requires some coordination effort at the organizational level. The development team is aware that it must provide some documentation to the quality control team, but the developers do not know the level of documentation required.

These frontiers between processes resulted in some serious issues. For example, during the acceptance phase, the testers and the developers found that a requirement had not been met. This was a serious setback, as it meant that the developers and testers had to produce code and validate it in a rush. That missing requirement was to fix an issue with the previous version of the Module. The quality control team was aware of this requirement but failed or neglected to propagate the information to the development team.

Another serious issue was with the setup of testing environments. It took weeks for the testing environment to be set up, and the environment was never fully coherent with the specifications given because of misunderstandings between the developers' needs and the environment team's comprehension. It took many more weeks of back and forth iterations before the new environment could be used. Unfortunately, by the time the problems were solved, most of the development team's allowed time for this environment had elapsed and the testing environment was passed to another team. As one developer exclaimed during one of the meetings : "We fought like dogs to get this environment and we can't even use it !"

At the other end of the spectrum, the organization had introduced a new, very stringent process for the submission of code patches to non-development environments. This new pro-

cess was introduced after insufficiently tested code ended up causing havoc in production. Though warranted, it causes a lot of complications when applied to patches in test environments. Code patches cannot be freely submitted to test environments because testers must be constantly aware of how the code changes within their testing environment. The issue is that this procedure is applied even during environmental setup ; that is, when code must be adjusted because the environment is being built for the first time. The new process did not take into account the development of new environments, which caused undue and frustrating delays.

C.4.6 The "budget protection" issue

The organization's culture puts an emphasis on staying on target, meaning deliveries on time and under budget. The pressure on the team is real : an observed example is the practice of patching instead of fixing. One specific external software item was in dire need of a fix. Unfortunately, a fix requires a formal CR—a costly and billable project estimated at about 20 person-days. The software development team instead chose to internally develop a wrapper in order to mash the data to address their need, a cheap patch which could be done for about five person-days.

Currently, that specific software item has about 12 such wrappers. This means that at least 12 other development teams chose the cheap patch solution instead of the costly fix, even if the one costly fix would have obviated the need for 12 cheap patches. In the words of the developers themselves, "Eventually, one team will need to bear the cost of fixing this." No one wants to be that team that will go over-budget, and be criticized for it.

The validation provided a mitigated point of view of this issue. The operations manager said budget was rarely an issue, probably because his team's work was directly linked to the company's main business process. The marketing manager, however, said budgets were tightly monitored and it was very difficult to argue for more time and money. Discussion with the Module's developers outlined that budget pressure was not the main issue, deadline pressure was. The team admitted that they had budget restrictions, but was much more concerned with limiting the scope in order to prevent deadline slippages.

C.4.7 The "scope protection" issue

A large part of the managers' duty is to protect the scope of the project, i.e. prevent modifications to the original specifications of the project. This is closely related to the internal dependency issue, as the scope of the project is mostly assailed by other projects that want

to assign some of their related code changes to the Module. This may cause conflicts between development teams, since each team wants to protect its own project. The following scenarios are often put forward :

— Requiring a change : We ask you to adapt your code to our changes otherwise our project may fail integration testing ;
— Refuting a change : We may ignore your code changes, since they are out of our project scope. If you really want it, we will bill your project for the changes.

The following example occurred one week before going into acceptance testing. The Module operators asked for a new feature, which would require some major changes. The team proceeded to :

— Convince the operators that it was not in the initial specifications, and that it may be implemented in a future project.
— Convince upper management that the change was not as minor as it first seemed, and could not be tested properly before acceptance testing, which would introduce some significant risks into the deployment.

Scope freezes exist within the organization, but are hard to enforce. The organization works in a very competitive domain : If a competitor provides a new service, the organization must support a similar service as soon as possible. Therefore, development priorities can be shifted overnight.

Additionally, compatibility between modules is often raised at the last minute and is therefore patched haphazardly, which may have a dubious impact on the resulting software product. This results in software defects at the boundaries between modules, where each side blames the other for the errors, lengthening the debugging process and making deadline slippage more likely. Therefore, the organization's culture of independent teams working within tight deadlines causes them to work mostly in competition with each other.

C.4.8 The "organizational politics" issue

Within an organization, the manager's most important tool is often his/her list of contacts.

For example, sometimes third party entities would refuse a CR on the basis that the change would violate the core functionality of their libraries. Inexperienced developers or managers would often accept these refusals at face value. However, employees with experience on the capability of the external libraries would often challenge these decisions. One manager confirmed cases where decisions challenged by experienced developers were conceded by the third party. In these cases, talking to the right person at the third party entity was the key factor

of success.

Another example is illustrated by the problems that plagued the team for over two months concerning test environments. After nearly two months of stalling and non-functional test environments, a senior operations manager asked the Module manager to contact him each time a problem occurred with the environments. The problems were resolved within a week.

A similar problem, concerning inter-module communications, was resolved very quickly when the right person—the senior operations manager—applied pressure on the right people in charge of the other module. Quality-wise, this means that many issues dragged on for weeks before being resolved, and that developers did not have as much time as they needed to correct the issues, which is likely to result in more patched-on code.

It also means that even a well-defined software engineering process would not resolve some of these issues, which are dependent on goodwill and personal contacts. This kind of behavior is often deeply rooted within the organizational culture of the company.

C.4.9 The "human resource planning" issue

The composition of the development team evolved throughout the project. Some developers joined when their specific expertise was required, and left when it was no longer needed. This is typical of most development projects. However, what is peculiar in this case is that the only developers who have participated in the project since its beginning, the Module's "team historians" as defined by La Toza et al. ([5], cited by [6]), are two contractual employees whose contract is about to end. This means that the development team will lose all the knowledge of the first few months of the project just before deploying the application on the production server.

The team realized that this could be risky, and the contractual developers were kept until the deployment and asked to frantically write documentation in order to support maintenance activities after their departure. One contractual developer has been assigned a new experienced developer in order to transfer his knowledge of the project to him.

However, these knowledge transfer tasks were assigned on top of the usual activities of the contractual developers. During two weekly status meetings, the new developer simply states that he is "waiting on the developer" to give him the relevant information, hinting that knowledge transfer is not at the top of their priorities.

Human resources should be better planned for long-term software development. To quote Laurie and Kessler on the issue :

> It is not advantageous to have all knowledge in any area of a system known to only one person. [...] This is very risky because the loss of one of a few individuals can be debilitating. [7]

At least the issue was handled before it was too late, but it could have been avoided altogether.

C.4.10 The "undue pressure" issue

As presented earlier, software development is not the main business process of the company. Its bread and butter are the operation of telecommunications services, and as such the most important department is operations. Consequently, operations have a lot of pull when it comes to coercing teams in other departments to perform specific tasks.

The observer has seen two occasions where operations personnel came to the weekly status meeting to put explicit pressure on the software developers. In one of the cases, the operations manager presented a clear threat : “If you do not allow 100% of your time on [a specific activity], I will be very disappointed. Very disappointed. Did I make myself clear ?” The operations department was clearly unhappy about the advancement of an activity and made its disappointment evident.

The “undue pressure” approach supposes that developers are not doing their best. Yet they were already working full time on their assigned tasks and were frantically putting the finishing touches on features that were about to be sent to testing.

The issue is that the pressure, which might be justified in some cases, must be applied in the appropriate context. Undue pressure from upper management or from other departments can coerce the team to work on subjects that are not optimal for the quality of the product, especially when management is not completely up-to-date with the project status.

C.5 Discussion

One of the main issues of observational study research is whether the observations can be generalized to other settings. In the case at hand, the following question can therefore be asked : Is this a one-of-a-kind organization, or does it represent a widespread situation within the industry ?

Similar research performed by Jaktman in 1998 showed that organizational values have an impact on the high-level architecture of software and should therefore not be neglected [3]. A survey of 40 Chinese companies performed by Leung in 2001 shows that quality is the last concern for management, after functionally correct, within budget and on schedule. Simi-

larly, among quality characteristics, maintainability trails behind reliability and functionally correct, with only 35% of managers considering quality to be a key issue [2]. Software engineers want to produce quality software, but it seems that many organizations do not take the proper approach to reach this goal.

Two observation sessions conducted by Allison in 2010 [8] showed the following :

— Case 1 : When a conflict arose between Agile teams within a non-Agile organization, the will of the organization prevailed and the Agile principles more or less disappeared.
— Case 2 : Constant pressure from an organization on one of its development teams forced them to change their practices, despite significant resistance on the team's part.

Allison's conclusion is that the influence of the organization on the team is larger than the influence of the team on the organization.

It can therefore be surmised that organizational factors may impact product quality, as the values of the organization will influence how team decisions are made.

The objective of this observational study is to bring more evidence to existing literature on specific issues related to organizational values or structures, and software quality. The following sections present how each issue could be related to a generic case, and present the literature pertaining to this case, when it is available. We also suggest corrective action which, although case-specific, might prove useful for other practitioners facing similar issues.

C.5.1 The generic case of "documenting complexity"

The "documenting complexity" issue is related to knowledge management processes, especially on how development teams transfer knowledge to support and maintenance. The observed developers had many questions on the legacy software in their organization. These questions can be summarized into the two following queries :

— Why does this specific piece of software exist ? What does it do ? Who needs this software ?
— How does it work ? Which resources (database, internet ports, etc.) does it use ?

These issues are well-known in the knowledge management field. The need to record both the "what" (inner workings) and the "why" (design rationale) of software has been well documented [9], [10]. In the observed case, the complete system is broken down into software items like modules, scripts, libraries, etc. The purpose of each software item within the complete system is often unknown, as few of them are documented. Developers need to know how the software item works, in order to maintain its functionality. But they especially want to know its purpose, in order to ensure that their changes do not break it. To paraphrase

Ko et al., the ideal, cost-effective, documentation system would be “demand-driven” instead of exhaustive [6]. Ko et al. encourages face-to-face interactions with colleagues because the data are available on demand and it requires little effort to record, maintain and transmit.

An up-to-date list of experts within the organization for each software item could also be interesting for future development efforts [11], although it cannot be considered a panacea by itself [12].

C.5.2 The generic case of "internal dependencies"

The “internal dependencies” issue is related to the concurrent development of inter-dependent modules. This is a common issue when using third party libraries, an issue which is resolved by providing multiple version support by designing flexible services [13]. See for example, the multiple PHP language libraries supported [14].

Supporting multiple versions is not always financially possible within private organizations, however. For the observed organization, the software developed is executed only once on the organization’s servers. There is no need to support multiple versions, because only the latest version will be executed at any given time. This may result in some conflicts, as each team wants to push its latest version to the organization’s server, causing rippling issues within other modules currently under development.

This issue has not been directly studied in the literature. At the technical level, it can be related to the design of evolving families of web-services [13]. This approach might be a hard-sell however for an organization where only one version of the software is executed. At the management level, it can be related to inter-team negotiations within global software development contexts [15]. Team negotiations are still an emerging research subject, as Guo and Lim presented in 2007 :

> Despite considerable investigation on negotiation support systems (NSS), such research is largely in dyadic (i.e., one-to-one) context which is challenged by the observations of business practices : negotiating teams often appear at the bargaining table. [15]

This case study confirms the suspicions of Guo and Lim that team negotiations often occur within business contexts, and that these negotiations can lead to conflicts.

As a corrective action, in the case of parallel development of inter-dependent software modules, we suggest setting up a negotiation table to settle conflicts between the development teams. Given the effort required to manage negotiations and track results, efforts should

be focused on dependencies with significant impacts on each project's success. The use of a software tool like the one suggested by Guo and Lim [15] could also help mitigate the effort overhead of negotiation tables.

C.5.3 The generic case of "external dependencies"

The "external dependencies" issue is related to acquisition and reuse processes, especially on how to interact with third party developers. Within the observed organization, the impact of contractual obligations with third party developers is twofold :

— First, the many different entities and contracts are an important knowledge hurdle for new developers, who need to know who to contact to obtain information, and which modifications are possible under the current contract.
— Second, the cost (in time and money) of CRs forces developers to make cost-conscious decisions, instead of quality-conscious ones.

Research on third party acquisition and reuse is abundant : For example, it is the purpose of the CMMI-ACQ [16] and it is covered by the Acquisition process of ISO 12207 [17]. The CMMI-ACQ, especially, outlines the need for training after the acquisition of third party software, in order to avoid the first impact presented above.

The second impact is also covered in the contractual negotiation sections of the two models. The importance of selecting the right third party developers cannot be underestimated : it is important to ensure that third party developers make the requested changes in a cost-effective and timely manner, and that they follow the rules of the trade. In the case observed, many CRs were performed either late or poorly, and did not meet the specifications given. These constant costly delays forced the development team to change their design for the worse, in order to avoid dealing with CRs.

As a corrective action, we suggest that the organization make sure that the team responsible for third party software acquisitions is aware of the CMMI-ACQ or ISO12207 processes, and of known challenges during large-scale software acquisitions [18]. The impact of a poor contract can be serious : in some cases, the customization performed by the company in order to work with the third party software is so extensive that moving to another supplier can be prohibitively expensive. Some of these acquisitions will last many years, or even decades.

C.5.4 The generic case of the "cloud storage" issue

The "cloud storage" issue is similar to the previous one, but has to do with testers instead of developers. The issue here is related to the absence of testers during contract negotiations :

some of the specifications of the development team had not been met by the contract.

However, given the sensitivity of the matter (vulnerability testing), negotiations are bound to be difficult. The issue is that the whole system works as a chain, where an input passes within multiple subsystems before producing an output. Some of these subsystems within the chain are supported by third party entities. The observed organization wants to test for specific security vulnerability, but some of these third party entities are not warm to the idea of having one of their clients literally hacking their systems.

Cloud vulnerability testing is currently an emerging research area. Its importance is undeniable : following the theft of 40 million Target credit card accounts, the company was forced to testify before the U.S. Congress and had to fork over US$60 million in costs to mitigate the issue [19]. The observed organization wanted to avoid a similar catastrophe, which could result in its customers' private information being made available on the Web or elsewhere.

Our suggested corrective action is to make sure that all relevant stakeholders (developers, testers, support, etc.) are involved when negotiating the acquisition of a software component with a third party. The development team should also make sure that all the activities planned for the project are possible within the current contractual obligations. For example, if the entire project hinges on a functionality that a third party entity cannot or will not deliver, the whole project can be thrown into jeopardy.

C.5.5 The generic case of "organically grown processes"

The issue of "organically grown processes" shows that processes typically appear on an as-needed basis, and do not follow a global plan. Allison and Merali describe the appearance of processes in the following way : "Software processes can be considered to emerge by means of a structuring process between the context and the content of the action" [8]. When the context and content of a development activity warrants it, a process element is added. An example would be an unexpected crisis, like a software patch causing a whole system to crash. Crises like these are usually followed by the introduction of process elements dedicated to preventing similar crises.

This issue highlights the importance of building an organization-wide process plan, such as the ones promoted by CMMI-DEV [20] or ISO 15504 [21]. The problem with organically grown processes is that they create islands of formality—zones where various software engineering processes have few interactions between them. Our suggested corrective action is to plan an organization-wide reviews aimed at detecting isolated processes and promoting information flows between processes. Project post-mortems could also serve to detect where

missing information caused a problem, and whether someone in the organization had this information and could have prevented the problem.

To avoid resistance to change from software developers [22], process improvement should focus first on a better use of existing activities and materials. Changes to existing processes should also be gradual, with an emphasis on evaluating and demonstrating whether the change is useful.

C.5.6 The generic case of "budget protection"

Unfortunately, many organizations see only short-term benefits. They want correct functionality, within budget and on schedule, with good quality, in that order [2]. Reducing the cost of future maintenance is rarely a priority. This observational study provides more confirmation that working software is more sought after than quality software.

One of the causes of this problem is that budgets are closely monitored. Development teams are discouraged from performing costly corrections that would benefit software modules from other teams. These types of general-purpose corrections or improvements need to be performed in a distinct project, which can be difficult to get approved since there are no immediately countable benefits. Developers are therefore encouraged to apply quick patches, instead of solving the issue once and for all.

The impact of these quick patches has been widely documented and identified as “software aging” [23] or “technical debt” [24]. Technical debt is defined as a technical shortcut which is cost effective in the short term but expensive in the long term. Developers relying on patches during development can accelerate the accretion of technical debt. Quick patches applied haphazardly over other patches are certainly more prone to create errors than a careful resolution of the problem at hand. Our suggested corrective action is to plan “special” budget items to support long-lasting corrections or corrections that are likely to benefit many modules. This special budget could also benefit inter-team negotiations. It might be easier to reach a settlement if the issue can be resolved in a separate special project.

This case presents the need to reach a compromise between developers who want to write perfect code, and managers who want to avoid “gold plating”. A global view of how the organization’s legacy code should be could help the development teams know where accrued technical debt might do more damage. Inter-team negotiations could be easier if priorities are set in a global software plan.

C.5.7 The generic case of "scope protection"

The “scope protection” issue is strongly related to deadlines. In the observed organization, changes in project scope are usually supported by an extra budget. However, budget changes do not necessarily change the project timeline.

Therefore, this is a direct application of Brooks’ law, which states that adding new resources to a late development project only causes it to finish later [25]. In the case observed, the extra resources were in the form of developers added late in the process. These developers stood mostly idle for some weeks as their mentors were already swamped with work and could not train them properly. The addition of new developers therefore slowed down the development process, as the current developers spent part of their time assisting the newcomers.

Projects with strict deadlines are risky, and should be carefully monitored to avoid last-minute unplanned activities. As Brooks wrote : “A project is one year late one day at the time !” [25]. The problem is that for each task added to an already tight deadline, the quality of the work takes a hit. Our suggested corrective action is either to allow deadline slippages, to enforce scope freezes on development projects, or to implement a more open scope negotiation process between teams involving a neutral judge (either a senior developer or the IT department manager).

C.5.8 The generic case of "organizational politics"

Another, more human issue is related to “organizational politics”. The company’s culture encourages managers to work outside the software process and rely on internal politics. The validation interview confirmed that observation. Results are not obtained solely by following due process, but also by calling in favors from colleagues.

This management approach has been labeled “organizational politics”, because it uses a political approach akin to lobbying [26]. Previous research on this issue has shown that organizational politics are important in some organizations, ensuring that software development teams have the resources they need [26].

This is interesting because it shows that a CMMI or ISO 15504 certified organizational process is not sufficient to increase project success, as the organizational culture encourages employees to find solutions outside the prescribed process. On the one hand, processes can define useful information flows, but on the other hand humans prefer to work face-to-face rather than through documents and forms. Team members should therefore maintain a careful balance between formal development processes and informal human interactions, or as Allison and Merali put it, they must make some “negotiated changes” to the development processes [8].

C.5.9 The generic case of "human resource planning"

Many studies in software engineering pertain to the so-called “truck number”, a metric attributed to Jim Coplien : “How many or few people would have to be hit by a truck (or quit) before the project is incapacitated ?” ([7], p41). For the team studied here, the “truck number” is a real issue, as it is about to lose its only two members familiar with the initial project and specifications.

The development was performed mostly in silos, resulting in only one or two software developers gaining expertise in any given domain, which caused problems throughout the project. Whenever a developer took time off, most work on his/her area of expertise had to be postponed because no one else knew enough about how each piece of the puzzle worked. Similarly, when a developer got overwhelmed with work, the others could not help because no one had the appropriate knowledge. The “truck number” risk is therefore not only a threat to project survival, but can also create delays when the knowledgeable developer becomes a bottleneck.

Team members should make sure that knowledge is appropriately distributed amongst them. Our suggested corrective action is to promote practices which promote knowledge sharing (e.g. pair programming, code reviews). The objective is to break the silos and have the developers work outside their areas of expertise once in a while.

C.5.10 The generic case of "undue pressure"

The “undue pressure” issue refers to priorities imposed by a higher authority, whether upper management, a client, or a respected colleague. However, this higher authority might not have all the knowledge required to impose these priorities. The higher authority dictates priorities according to what he/she perceives as important, and not necessarily what is really important at the team level. This can result in the developers working on something that does not warrant immediate attention, and thus wasting project resources.

The important role of upper management support for the introduction of new practices (e.g. software process improvement) has been confirmed many times in the past. There does not seem to be much research on the relevance of upper management decisions, however, nor on whether these decisions are coherent with the needs of software development teams.

Any intrusion into team dynamics by outsiders should be done very carefully. These intrusions can be useful to correct a problem, such as developers failing to apply a procedure [8]. Our suggested corrective action is to make sure the outsider has all the relevant information before intruding at the team level.

C.6 Conclusion

The objective of this paper is to provide accurate empirical data on the state of software engineering in practice in a professional environment. The list of issues presented in this paper is by no means exhaustive : many more minor issues were observed during the ten month study. Additionally, generalization of the issues to generic cases shows that the main issues presented here are not new ; most of them have been previously discussed in the literature. What this paper is stressing is that any software development project should present few of these issues and ideally none of them.

While these issues might not affect project success, our observation shows that they do affect software quality. And quality factors can have a major impact on maintenance costs. If the Module developed in this project is successfully deployed, it will likely be used for the decades to come. The design flaws introduced because of the organizational issues presented here will no doubt come back to haunt at least a generation of developers to come, as the code written today will be tomorrow's legacy code. Is this the kind of legacy we want to leave for future software developers ?

C.7 Acknowledgment

This research would not have been possible without the agreement of the company in which it was conducted, (which prefers to stay anonymous), and without the generous participation and patience of the software development team members from whom the data were collected. To all these people, we extend our grateful thanks.

C.8 References

[1] J. Mathew, "The relationship of organisational culture with productivity and quality. A study of Indian software organisations," vol. 29, pp. 677-95, 2007.

[2] H. Leung, "Organizational factors for successful management of software development," Journal of Comp. Info. Systems, vol. 42, pp. 26-37, 2001.

[3] C. B. Jaktman, "The influence of organisational factors on the success and quality of a product-line architecture," Proceedings of ASWEC '98 : Aus. Software Engineering Conference, Los Alamitos, CA, USA, 1998, pp. 2-11.

[4] T. C. Lethbridge, S. E. Sim, and J. Singer, "Studying software engineers : data collection techniques for software field studies," Empirical Software Engineering, vol. 10, pp. 311-41, 2005.

[5] T. D. LaToza, G. Venolia, and R. DeLine, "Maintaining mental models : A study of developer work habits," Proceedings of the 28th International Conference on Software Engineering (ICSE '06), Shanghai, China, 2006, pp. 492-501.

[6] A. J. Ko, R. DeLine, and G. Venolia, "Information needs in collocated software development teams," Proceedings of the 29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, United States, 2007, pp. 344-353.

[7] L. Williams and R. Kessler, Pair Programming Illuminated. Boston : Addison-Wesley, 2003.

[8] I. Allison, "Organizational factors shaping software process improvement in small-medium sized software teams : A multi-case analysis," Proceedings of the 7th International Conference on the Quality of Information and Communications Technology (QUATIC 2010), Porto, Portugal, 2010, pp. 418-423.

[9] J. E. Burge and D. C. Brown, "Software engineering using RATionale," Journal of Systems & Software, vol. 81, pp. 395-413, 2008.

[10] P. Avgeriou, P. Kruchten, P. Lago, P. Grisham, and D. Perry, "Sharing and reusing architectural knowledge - Architecture, rationale, and design intent," Proceeding of the 29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, United States, 2007, pp. 109-110.

[11] Y. Ye, "Supporting software development as knowledge-intensive and collaborative activity," Proceedings of the 2nd International Workshop on Interdisciplinary Software Engineering Research (WISER '06), Shanghai, China, 2006, pp. 15-21.

[12] D. W. McDonald and M. S. Ackerman, "Just talk to me : A field study of expertise location," Proceedings of the ACM Conference on Computer Supported Cooperative Work, pp. 315-324, 1998.

[13] S. R. Ponnekanti and A. Fox, "Interoperability among independently evolving Web services," in Middleware 2004. ACM/IFIP/USENIX International Middleware Conference. Proceedings, Berlin, Germany, 2004, pp. 331-51.

[14] T. P. Group. (2014, August 20th). PHP : Downloads. Available : http ://ca1.php.net/downloads.php

[15] G. Xiaojia and J. Lim, "Negotiation support systems and team negotiations : the coalition formation perspective," Information and Software Technology, vol. 49, pp. 1121-7, 2007.

[16] Software Engineering Institute, "CMMI for Acquisition, Version 1.3," Carnegie Mellon University, 2010, p. 438.

[17] IEEE Computer Society, "ISO 12207 :2008 - Systems and software engineering — Software life cycle processes," ISO/IEC, 2008.

[18] A. Al Bar, V. Basili, W. Al Jedaibi, and A. J. Chaudhry, "An Analysis of the Contracting Process for an ERP System " Proceedings of the Second International Conference on Advanced Information Technologies and Applications (ICAITA-2013), S. Vaidyanathan and D. Nagamalai, Eds., CS & IT-CSCP, 2013, pp. 217-228.

[19] M. Riley, B. Elgin, D. Lawrence, and C. Matlack, "Missed Alarms and 40 Million Stolen Credit Card Numbers : How Target Blew It," Bloomsberg Businessweek, 2014.

[20] Software Engineering Institute, "CMMI for Development, Version 1.3," Carnegie Mellon University, 2010, p. 482.

[21] IEEE Computer Society, "ISO/IEC 15504 :2004 - Software Process Improvement and Capability Determination (SPICE)," ISO/IEC, 2004.

[22] M. Lavallee and P. N. Robillard, "The impacts of software process improvement on developers : A systematic review," Proceedings of the 34th International Conference on Software Engineering (ICSE 2012), Zurich, Switzerland, 2012, pp. 113-122.

[23] D. L. Parnas, "Software aging," Proceedings of the 16th International Conference on Software Engineering, Sorrento, Italy, 1994, pp. 279-287.

[24] S. McConnell, "Managing Technical Debt," Construx Software Builders, Inc.2013.

[25] F. P. Brooks, The Mythical Man Month : Essays on Software Engineering : Addison-Wesley, 1975.

[26] A. Magazinius and R. Feldt, "Exploring the human and organizational aspects of software cost estimation," Proceedings of the 2010 ICSE Workshop on Coop. and Human Aspects of Soft. Eng. (CHASE 2010), Cape Town, South Africa, 2010, pp. 92-95.

ANNEXE D ARTICLE 3 : PLANNING FOR THE UNKNOWN : LESSONS LEARNED FROM TEN MONTHS OF NON-PARTICIPANT EXPLORATORY OBSERVATIONS IN THE INDUSTRY

Type de publication	Référence complète
Workshop	*Planning for the Unknown : Lessons Learned from Ten Months of Non-Participant Exploratory Observations in the Industry*, **Mathieu Lavallée** et Pierre N. Robillard, Workshop on Conducting Empirical Studies in Industry (CESI), 2015, doi :10.1109/CESI.2015.10

D.1 Abstract

Convincing industrial partners to support an exploratory study can be difficult, as benefits are often fuzzy at the beginning. The objective of this paper is to present recommendations for industrial exploratory studies based on our experience. The recommendations are based on ten months of observations during a non-participant, exploratory study with a single industrial partner. This study confirms a number of methodological challenges already identified in the software engineering literature. Based on recommendations from the literature and our own experience, we propose a process for future observational exploratory studies.

D.2 Introduction

Exploratory studies are akin to archeological work. A site with potential artifacts is targeted, negotiations with the site's owners are initiated, and archeological digs are excavated where the artifacts are supposed to be. Initial assumptions may be wrong however, and archeologists correct their approach as necessary. Exploration may be surprising. Archeologists can find what they expect, they can return empty handed, or they can find something new and unexpected, like Howard Carter's incredible discovery of the sealed tomb of Tutankhamun in 1922.

While our exploratory studies in software engineering are much less dramatic, we can still gather lessons from archeologists and anthropologists. Lesson one is that it is possible to plan for the unknown, and lesson two is that any such plan will have to be flexible.

This paper presents the insights gained from a ten month non-participant exploratory observation performed in a telecommunication organization. Its aim is to present the lessons learned from the study, as well as suggest recommendations for future exploratory studies in software engineering. The next section presents the industrial context along with the methodological context of the study. Section III presents current methodological recommendations for industrial studies in software engineering. We follow with a presentation of the lessons learned from our own study in section IV. Finally, we present our recommended exploratory study process in section V.

D.3 Study Context

Our exploratory observational study focused on a single software development team and lasted over ten months. The following two subsections describe the context of the organization and the context of the study.

D.3.1 Industrial Context

Our study took place in a private telecommunication organization, employing more than five thousand people across Canada. The software development department is mainly concerned by the support of the main business processes of the organization. Software development budgets are therefore limited since the software does not contribute to the revenues of the organization.

The organization uses software since the ‘60s, and software development practices have thus evolved with the evolution of the software engineering body of knowledge. Therefore, a lot of their older software have acquired a significant technical debt [1], whether in the form of poor design decisions or lack of accurate documentation. The organization is working hard to keep pace with software engineering state-of-the-art, but it is still lagging behind due to the large size of their legacy software and the fact that software development is not their main business process.

Therefore, software development follows a primary based waterfall process. This gate-based approach enables management to decide to continue the project or pull the plug at the end of each phase. Many projects are cancelled in the early stages when managers feel that the costs and risks outweigh potential benefits.

The organization is aware of the importance of face-to-face interactions and tries to place all the members of a development team on the same floor of the same building. Colleagues from other teams are based in different buildings downtown, so exchanges can be made quickly and

face-to-face meetings can be called as needed. The team under observation hired for a short time two contract developers based outside the city : one elsewhere in the province and the other in the United States. Both contract developers were in the same time zone and shared a similar culture, although there were some language issues between the team members and the developer in the USA.

Over time, the organization's codebase has accrued a large number of internal dependencies. The software development team's members must often contact colleagues in other teams in order to coordinate their work. The organization's codebase also uses a number of libraries developed by third party organizations. Software development team's members must sometimes contact these third parties, which can be based anywhere in the world.

To understand the context of our study, it is important to provide details on the people involved [2]. Figures 1 and 2 present the details of the core team observed. It was a team of eleven people, two females and nine males. The team is composed of eight developers, two testers and one manager.

D.3.2 Methodological Context

The initial objective of our study was to observe how team interactions during software development can impact software quality. The researchers had two contacts within the organization's upper management, who acted as champions [3] during the study.

Upper management gave their support, but remained anxious to minimize interference with the normal activities of the software development team, a common challenge for industrial studies in software engineering [2]. That is why a non-participant approach was chosen.

Observations were limited to the weekly one-hour all-hands status meetings in order to minimize interference. Since we did not know what would be discussed during these meetings, the study became exploratory.

Keeping the study design flexible proved critical, as was observed in another industrial study [4]. We did receive a description of what is typically discussed in weekly status meetings, but it was not clear whether there would be enough details to fulfill our initial study objectives. As we observed the first few meetings and became familiar with the topics discussed, the study objective was indeed revised. The new objective was to determine :

— RQ1 : How may the team decision process affect software quality ?

But in order to answer this question, we must first answer the following sub-question :

— RQ1.1 : What is the team decision process ?

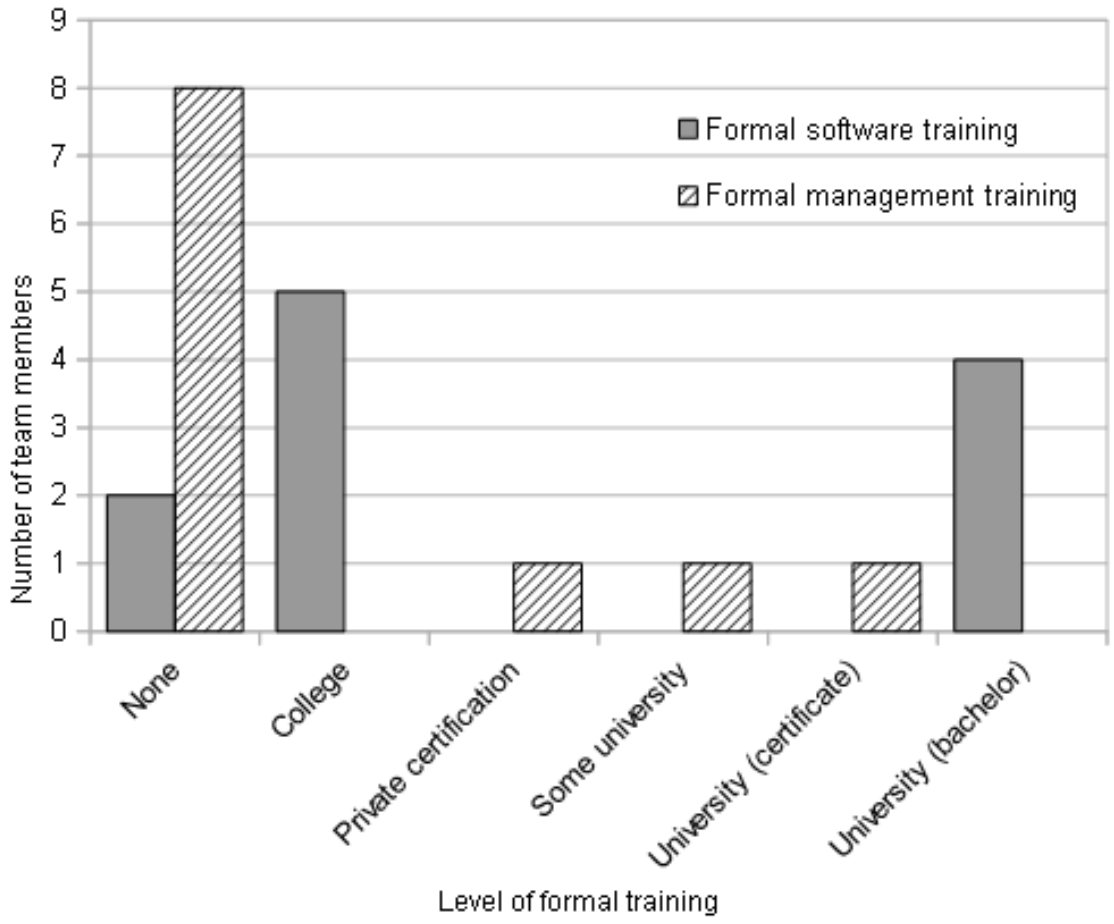

Figure D.1 Formal training of the observed team.

And in order to answer this sub-question, we must first understand the interactions inside and outside the team. These three final questions helped us pinpoint what we should observe in this field study :

— RQ1.1.1 : Who is talking to whom ?
— RQ1.1.2 : Who is referring to which colleague outside the meeting ?
— RQ1.1.3 : Which topics are discussed during weekly status meetings ?

Data collection and analysis was essentially qualitative. A transcript recording would have been ideal, but the team seemed initially uneasy at the presence of the researcher. Coupled with the desire to not provide an interference in the team's activities, it was decided to limit data collection to pen-and-paper transcripts taken live during the meeting.

We wanted to try a more formal data collection approach using a software tool. However, it

Figure D.2 Experience of the observed team.

proved difficult to convince the managers to introduce a new tool within their team, especially since this study was the first contact between the researchers and the organization.

The pen-and-paper transcripts were then recorded and organized as series of “interactions”. An interaction is defined as a statement made by one or more persons and of a typical duration between five and thirty seconds. An ensemble of statements is related to a topic, and often ends with an outcome, such as a decision, a request for more information, etc. Table I presents an example of a topic taken from the study, where ’DevX’ represents a developer, ’ManX’ represents a manager and ’TestX’ represents a tester.

All names in the transcripts were coded, in order to keep the study anonymous. Results were presented in a compiled form for the same reason. The agreement with the organization stipulated that no detail shall be given that would enable anyone to pinpoint a specific individual. The original pen-and-paper transcripts were therefore destroyed, and only an index remained to link individuals with their codes, and that index was kept confidential by the research team.

Table D.1 Interactions on the topic of the bug tracking software.

Speakers	Details
Man1 + Dev6	"Are there still some bugs coming from the third party library ?"
Dev5 + Test2	Test2 and Dev5 describe a bug where an object is not being propagated to all relevant modules.
Dev6	"Yes, we already corrected that bug."
Test2	"Yes, you did correct it, and the correction worked, but it broke something else and now the object is not being propagated."
Dev5 + Dev6 + Dev7	Dev5 describes the bug to Dev6 and Dev7. Dev6 and Dev7 were not aware of this bug.
Man1	"Did you modify your bug tracking software so you receive an alert when bugs are reported ?"
Dev4 + Dev5 + Dev6	"Yes."
Man1	"Revise regularly the list of reported bugs, and assign to yourselves the unassigned bugs."
Dev9	"Ok, I understand, we need to be proactive."

The researchers joined the meetings from January to November 2014. The study took place during the ten last months of a two year project, and it observed the end of the implementation phase, the testing phase, and the deployment phase.

The layout of each meeting (i.e. where each person sat down) was also noted in order to determine who was present at each meeting and to observe subgroups of the team sitting together. Figure 3 presents an example of the layout of one of the meetings of the study, where 'DBX' represents a database administrator, 'SecX' represents a security expert, 'EnvX' represents someone from the environment setup team and 'Unk9' represents someone who did not speak and could not be identified. The square box with a 'T' represents a speakerphone used to contact those who could not attend the meeting. The identifiers next to the circle indicate where they sat down, while the identifiers next to the 'T' box indicate who was available through speakerphone.

Since the data collected is qualitative, a qualitative analysis approach was needed. The grounded theory approach [5] was used in order to regroup the recurring topics into a coherent

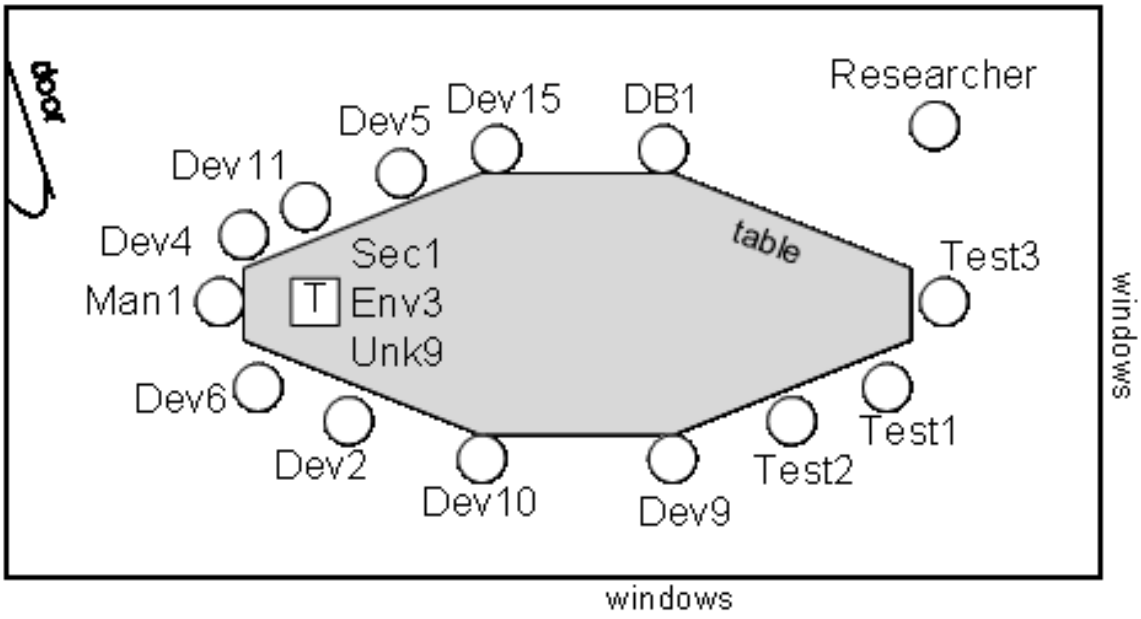

Figure D.3 Layout of a meeting with everyone present.

structure. A preliminary analysis resulted in a paper on how organizational pressures can force the team to take quality-degrading decisions [6].

D.4 Related Work

D.4.1 Industrial Studies in Software Engineering

A number of papers present success factors for conducting industrial studies, and this study can confirm the importance of some of them. For example, Dieste et al. write that :

> The major challenges faced by experimenters are to minimize the cost of running the experiment for the company and to schedule the experiment so as not to interfere with production processes. Companies appear to be disinclined to run experiments because they are not perceived to have direct benefits. [2]

As presented earlier, our study confirmed the importance of this challenge. The organization's perception can even be negative, i.e. the researcher can be perceived as a source of distraction detrimental to software development. The developers observed can also perceive the researcher as a possible proxy for upper management, which can seed mistrust and limit the quality of the information exchanged. As Sherman and Hadar write : "the employees cannot be sure that information provided [...] would not harm them in the future" [7]. This

was also the case in a recent observational case study, where the researchers "decided against shadowing since it might have made the participants concerned about employer surveillance" [8].

Wohlin describes ten challenges researchers can face during an industrial study [3]. Four of these were critical in our case :

— "Trust and respect" : It is difficult for the development team to build up trust and respect toward the researcher during a non-participant observational exploratory study. Since there is no participation from the researcher, the team cannot assess the researcher's expertise. The fact that our study did not present any immediate benefit to the team exacerbated this problem. This factor was also confirmed as critical by Grunbacher and Rabiser [4].
— "Champion" : Our study was made possible because of the buy-in of two managers. The researcher had to contact these "champions" on few occasions to get access to the meetings. One champion had a research academic background and understood the purposes of such activities. This champion's support convinced other managers to go forward with the project.
— "Social skills" : Despite the non-participant approach, it was important not to appear aloof. Developers sometimes discussed with the researcher before and after the meetings, and their questions reflected some of their concerns about the study. Social skills proved useful to build some level of trust and respect despite the study limitations. Additionally, a good relationship with desk security ensured that the researcher could freely access the meeting rooms.
— "Integrate into daily work" : The study started to work well when the presence of the researcher was deemed necessary to start meetings. At the end of the study, the team did not only accept the presence of the researcher, they expected him to be there.

Jain et al. reports in their study a form of "snowball sampling", where an initial sample observed is enhanced when other people interrupt the interview to talk with the person interviewed [9]. They claim that this was beneficial, as this introduced the researchers to more subjects. This study saw a similar form of "snowballing", as meetings often included people from other teams which had an interest in what the team was doing. From the eleven developers in the core team in January, the researcher ended up recording interactions with over fourteen more people by the end of August.

Unkelos-Shpigel and Hadar recommend studies in multiple organizations in order to share knowledge between organizations and therefore to help pass the message that these studies are useful [10]. This is not directly possible for non-participant studies, as non-participation

implies that data cannot immediately be shared with subjects. However, multiple organizations can be used for validation purposes. The conclusions of our study were shared with one other large unrelated organization in order to confirm or refute their generalizabilities.

We also saw "on the record / off the record" problems, as seen by Unkelos-Shpigel and Hadar [10]. For example, the organization had no problem with the researcher taking notes on software defects discussed during weekly meetings, but they did not accept to give access to defect tracking artefacts. Off the record discussions were fair game, but on the record evidence was not.

An issue presented by Witschey et al. for industrial studies is "data overload" [11]. This is an issue in our case, as at the end of our study we ended up with over 3000 transcript interactions. The transcript interactions need proper codification, which requires multiple iterations on all the interactions, per the grounded theory approach [5]. This will also cause problems when new researchers will join the project ("onboarding" [11]) and when the current researchers will leave ("offboarding" [11]).

D.4.2 Observational Studies in Software Engineering

Lethbridge et al. in their seminal compilation of study approaches in software engineering [12], present the advantages and disadvantages of observational studies. Its main advantage is that they are "easy to implement, give fast results, and require no special equipment" [12]. They also mention that observations are a good way to record informal communications. For example, emotional interactions (e.g. frustration, anger, jokes) are not normally recorded in technical artifacts but they can be recorded through observation. Its main disadvantage is the need to have "a good understanding of the environment to interpret the software engineer's behavior" [12]. This proved true in our case as the first meetings were difficult to understand for the researcher. The team used acronyms and terms specific to either the organization or the project. A one-to-one interview with one knowledgeable person proved helpful in clarifying some terms that were still obscure after few weeks. The lack of environmental understanding during the first few weeks means that some interactions were either incorrectly or poorly recorded.

Lethbridge et al. are somewhat critical of manual recording, such as our pen-and-paper data collection approach. The researcher cannot record everything going on, and thus this approach "is the most data sparse method and hence capture the least complete data record" [12]. We mitigated this issue by recording data at a median level of abstraction. In most cases, we do not have the exact sentences uttered, but we have the overall idea transmitted within the interaction. Competences in stenography might mitigate this issue furthermore,

but it would still fail at recording vocal intonations and non-verbal interactions, as an audio/video recording would. Obtaining an agreement to record audio and video is not always easy however, as we experienced ourselves.

Recent methodological reflections recommend the use of multiple methods in order "to leverage their different strengths" [8]. They give an example of questionnaires, which can be used to ask more personal questions that would not be answered in team meetings. We could mitigate our lack of access to project artifacts by meeting different people in different contexts :

— We met the project's client to discuss her perception of the product and the development team,
— We met a colleague in another team within the organization in order to assess preliminary interpretations of the data,
— We sent a questionnaire to all team members in order to collect professional information (e.g. training, work experience).
— We met development teams in one other organization in order to confirm or refute our conclusions.

Patil et al. also present two new challenges that proved important in our case [8] :

— "Access to the team" : Scheduling a study when the team is in a rush is unlikely to succeed. We were lucky to start the observation at a calm period of software development, because some of the later status meetings were hectic, with many angry interactions.
— "Team membership dynamics" : We were also lucky to join the project at a time when the team had stable membership. Later meetings saw the addition of security experts, system and database administrators, operators, client's representatives, deployment experts, etc. Simply remembering all the names proved difficult, as some later meetings had up to 23 participants, including those joining through speakerphone.

D.4.3 Exploratory Studies in Software Engineering

Calvacanti et al. report a similar challenge of convincing subjects of the importance of their exploratory study [13], as only 17 subjects returned their questionnaire out of the 180 contacted. Their recommendation is that "alternative methods for gathering such information must be developed" [13]. We faced a similar challenge when we realized that we could not get access to the project's artefacts.

Li et al. explain why an exploratory approach was chosen, instead of a more traditional

hypothesis-based approach [14]. They found that building a questionnaire from scratch was difficult, and even after two series of tests they were still doing significant changes to its content. They therefore decided to do an exploratory study using face-to-face interviews in order to gather a sufficient amount of data to build a more coherent questionnaire. The flexibility of an exploratory study was also an important factor for us. Our final study questions ended up being significantly different from our initial ones.

D.5 Discussion

This section presents the recommendations for future researchers on the conduct of observational exploratory studies in the industry. People lessons learned prescribe recommendations on how to manage the subjects observed, while research lessons learned prescribe recommendations on how to manage the study itself. We start however with the presentation of a process for observational exploratory studies.

A single study is not enough to provide recommendations to exploratory studies as a whole, but we can still build a process which we could be used in forthcoming exploratory studies. We believe that there are enough data collected, in the form of best practices, success factors and lessons learned, on observational and exploratory studies in software engineering to be able to start a crystallization of those practices into a formal process. This process is :

1. Find a champion : Before any official contact with the organization, a champion needs to be found.
2. Build a preliminary study design : Through discussions with the champion, build the broad strokes of a study design based on the data expected.
3. Sell the study to upper management : Present the study as a "win-win" project where upper management will get useful feedback on their practices. Secure access to the data required (teams, artefacts, etc.).
4. Plan alternatives : The data obtained might not be what was initially expected. The team might not be working on what you need ; the artefacts might be unavailable, etc.
5. Choose the right project : Organizations often have multiple projects running in parallel. Researchers should choose a project compatible with their research objectives. Additionally, it is preferable to choose a project either in its infancy or in a stable phase, as a team caught in deadline pressures might have little patience toward researchers.
6. Build up trust and respect with the team : Data quality will depend on how forthcoming the team is with their interventions. Plan some feedbacks to the team in order

to alleviate concerns about the study. Good social skills can be useful to keep good relationships between team members and researchers.

7. Perform preliminary analyses : The objective is to evaluate data quality. Can analyses be performed on the data collected ? Which details are missing for the analyses ?
8. Correct the study design : Exploratory studies require flexible study design, as the data collected can be different to what was initially expected.
9. Report to the industry : In order to keep a good relationship with the organization and perhaps open the door for more studies. Reports to the industry should however focus on problems with known solutions.

The aim of this process is to serve as a first step toward a more standardized exploratory approach, based on our lessons learned and the related work on industrial observational exploratory studies. We provide also additional details for the seventh step, on preliminary analyses. The analysis of a large collection of qualitative data proved difficult in our case. At some point during the study, an evaluation of the data should be performed in order to determine whether the data collected so far is of any use. Our recommendation is to perform a "quick and dirty" sorting-based analysis in order to evaluate the usefulness of the qualitative data collected :

1. Identify recurring events or problems.
2. Evaluate the potential impacts of these events or problems.
3. Try to find similar events or problems in the literature.
4. Report the most recurring events or problems with the largest potential impact, along with the associated literature (if any).

This preliminary analysis will assess if more details are required for qualitative data collection during the remainder of the study.

D.5.1 People Lessons Learned

Four recommendations can be linked to non-participant observational exploratory studies in general : On champions : Researchers need to find a champion before pitching the project to the organization. A member of the organization with a research background can be especially helpful in convincing management of the importance of industrial studies and of their potential benefits for the organization.

— On trust and respect : The Hawthorne effect, where a subject acts differently when it knows it is being observed [15], might have been an issue in this study. The organization

has a strong culture of management control, as demonstrated by their disciplined waterfall-based process. Our perception is that the team had concerns that the data collected could be used against them. We think this issue became somewhat mitigated as trust and respect were built during the project.

— On multiple sources : Accessing artefacts is difficult. Upper management initially agreed to give us access to documentation, but this agreement never materialized in practice. Since researchers still need to "triangulate" their results [16], [8], it is recommended to prepare an alternative plan. Our alternative solution was to triangulate using other stakeholders instead of project's artefacts.

— On the state-of-the-art : The organization we observed did not follow the state of the art closely. A study might identify new problems with little relationship to the research literature, well-known open problems with no consensual solution, emerging problems with mostly theoretical works, and well-known problems with well-defined solutions. Our validation with another organization showed that the industry wants to know how to solve their existing problems, so these last problems should be outlined. They are not very interested about new and emerging problems.

Performing an exploratory study introduces a number of challenges. The most critical is that it is difficult to present potential benefits of an exploratory study, as study designs are still fuzzy at the beginning. A win-win relationship is required for a successful study ; otherwise it can be difficult to justify our presence. The fact that benefits for both sides were initially fuzzy caused some issues during our study. We foresee that presenting real benefits would better "sell" the study to the upper management and to the observed team.

D.5.2 Research Lessons Learned

The main difficulty in managing an exploratory study involving qualitative data is that the parameters to be observed are not well defined at the beginning of the project. Some parameters to be recorded can be planned in advance, e.g. who attends the meeting, who talks to whom. Other will remain undetermined until the study starts, e.g. what kind of problems are discussed during the weekly status meetings, what is the decision process used. This is because the exact content of these meetings are not clear to the researcher, or even the participants, until the meetings take place. The perception of the team on what goes on in these meetings was also found to be flawed. The organization claimed that they were management meetings, but about half of the time was spent discussing technical problems and solutions, probably because technical problems resulted in management issues (i.e. deadline slippage).

Therefore, we recommend the use of a "humble" approach when recording data of exploratory studies, where we consider all data as potentially useful. For example, we did not record all the jokes exchanged during meetings, as we did not consider them relevant for our study. They could however have proven interesting in evaluating the mental state of the team throughout the project.

This "humble" approach must be compromised by the capabilities of the researcher. When using a pen-and-paper recording approach, it is typically impossible to record everything. We found that focusing on content instead of verbatim proved useful. Complex interventions could be summarized in keywords, which could then be expanded in when the pen-and-paper notes were transcribed electronically.

The positive aspect of an exploratory approach, however, is that we have little prior expectation on results, and thus little confirmation bias. Exploratory results can be interesting by themselves, since there is little pressure to obtain either positive or negative results [17].

D.5.3 Threats to Validity

The main threat to validity is the fact that this is a single case study. We believe however that this constitute an additional data point which might be useful for the confirmation or refutation of past and future theories. While this case cannot be generalized to all observational exploratory studies, it does confirm some challenges presented in the literature.

The proposed process was based on those confirmed challenges, and we think there are sufficient details in the literature to support its broad strokes. However, future work will be needed to improve and test this process.

D.6 Conclusions

Performing exploratory studies can be challenging. It is a foray into the unknown, which can result in, incorrect, incomplete and useless data if performed inadequately. While it is not possible to plan for unknown risks, this does not mean that exploratory studies should be engaged without a plan.

D.7 Acknowledgment

We thank the organization, who desires to stay anonymous, in which the study took place.

D.8 References

[1] I. Gat and C. Ebert, "Technical Debt as a Meaningful Metaphor for Code Quality," IEEE Software, vol. 29, pp. 52-5, 11/ 2012.

[2] O. Dieste, N. Juristo, and M. D. Martinez, "Software industry experiments : A systematic literature review," in 2013 1st International Workshop on Conducting Empirical Studies in Industry, CESI 2013, May 20, 2013 - May 20, 2013, San Francisco, CA, United states, 2013, pp. 2-8.

[3] C. Wohlin, "Empirical software engineering research with industry : Top 10 challenges," in 2013 1st International Workshop on Conducting Empirical Studies in Industry, CESI 2013, May 20, 2013 - May 20, 2013, San Francisco, CA, United states, 2013, pp. 43-46.

[4] P. Grunbacher and R. Rabiser, "Success factors for empirical studies in industry-academia collaboration : A reflection," in 2013 1st International Workshop on Conducting Empirical Studies in Industry (CESI), 20 May 2013, Piscataway, NJ, USA, 2013, pp. 27-32.

[5] A. L. Strauss, Qualitative Analysis for Social Scientists : Cambridge University Press, 1987.

[6] M. Lavallée and P. N. Robillard, "Why Good Developers Write Bad Code : An Observational Case Study of the Impacts of Organizational Factors on Software Quality," in International Conference on Software Engineering, Florence, Italy, 2015.

[7] S. Sherman and I. Hadar, "Conducting a long-term case study in a software firm : an experience report," in 2013 1st International Workshop on Conducting Empirical Studies in Industry (CESI), 20 May 2013, Piscataway, NJ, USA, 2013, pp. 47-50.

[8] S. Patil, A. Kobsa, A. John, and D. Seligmann, "Methodological reflections on a field study of a globally distributed software project," Information and Software Technology, vol. 53, pp. 969-80, 2011.

[9] S. Jain, M. A. Babar, and J. Fernandez, "Conducting empirical studies in industry : Balancing rigor and relevance," in 2013 1st International Workshop on Conducting Empirical Studies in Industry, CESI 2013, May 20, 2013 - May 20, 2013, San Francisco, CA, United states, 2013, pp. 9-14.

[10] N. Unkelos-Shpigel and I. Hadar, "Mind the gap and find common ground : Empirical research in multiple firms," in 2013 1st International Workshop on Conducting Empirical Studies in Industry, CESI 2013, May 20, 2013 - May 20, 2013, San Francisco, CA, United states, 2013, pp. 33-36.

[11] J. Witschey, E. Murphy-Hill, and X. Shundan, "Conducting interview studies : Challenges, lessons learned, and open questions," in 2013 1st International Workshop on Conducting

Empirical Studies in Industry (CESI), 20 May 2013, Piscataway, NJ, USA, 2013, pp. 51-4.

[12] T. C. Lethbridge, S. E. Sim, and J. Singer, "Studying software engineers : data collection techniques for software field studies," Empirical Software Engineering, vol. 10, pp. 311-41, 07/ 2005.

[13] Y. Cerqueira Cavalcanti, P. A. de Mota Silveira Neto, D. Lucredio, T. Vale, E. Santana de Almeida, and S. Romero de Lemos Meira, "The bug report duplication problem : an exploratory study," Software Quality Journal, vol. 21, pp. 39-66, 03/ 2013.

[14] L. Jingyue, F. O. Bjoernson, R. Conradi, and V. B. Kampenes, "An empirical study of variations in COTS-based software development processes in the Norwegian IT industry," Empirical Software Engineering, vol. 11, pp. 433-61, / 2006.

[15] H. A. Landsberger, Hawthorne Revisited : Cornell University, 1958.

[16] C. O. De Melo, D. S. Cruzes, F. Kon, and R. Conradi, "Interpretative case studies on agile team productivity and management," P.O. Box 211, Amsterdam, 1000 AE, Netherlands, 2013, pp. 412-427.

[17] S. Eldh, "Some researcher considerations when conducting empirical studies in industry," in 2013 1st International Workshop on Conducting Empirical Studies in Industry, CESI 2013, May 20, 2013 - May 20, 2013, San Francisco, CA, United states, 2013, pp. 69-70.

ANNEXE E ARTICLE 4 : PERFORMING SYSTEMATIC LITERATURE REVIEWS WITH NOVICES : AN ITERATIVE APPROACH

Type de publication	Référence complète
Journal	*Performing Systematic Literature Reviews With Novices : An Iterative Approach*, **Mathieu Lavallée**, Pierre N. Robillard et Reza Mirsalari, IEEE Transactions on Education, vol.57, no.3, pages 175-181, 2013, doi : 10.1109/TE.2013.2292570

E.1 Abstract

Reviewers performing systematic literature reviews require understanding of the review process and of the knowledge domain. This paper presents an iterative approach for conducting systematic literature reviews that addresses the problems faced by reviewers who are novices at one or both levels of understanding. This approach is derived from traditional systematic literature reviews and based on observations from four systematic reviews performed in an academic setting. These reviews demonstrated the importance of defining iterations for the eight tasks of the review process. The iteration approach enables experiential learning from the two levels of understanding : the process level and the domain level.

E.2 Introduction

Performing a systematic literature review (SLR) requires an expert to find the relevant studies, compile the important conclusions, analyze the key data, and synthesize the state of knowledge. Even with expert support, recent evidence shows that the reviews of novices are not repeatable [1], unlike the reviews made entirely by experts [2]. This therefore raises the following research questions :

— RQ1 : What can be done to ensure that the two main qualities of systematic reviews, completeness and repeatability [3], are optimal when novices are involved ?

— RQ2 : Can novices successfully perform a systematic literature review, where success is defined as a review that is both complete and repeatable ?

This paper presents a new iterative systematic review (iSR) approach designed for novices

in the domain and/or on the review process itself. This approach is based on the current state of the literature and our own experience guiding students in performing SLR. It was built and tested on four systematic reviews : The two first reviews were used to build the approach, while the two last were used to test it. Although the method presented here has been developed within a software engineering program it is not specific to this domain and could be applied to other areas.

E.3 Related Work

The two main qualities of SLR are completeness and repeatability [3]. Completeness implies that it must cover the whole field under study, and not be limited to subjectively selected papers. Repeatability implies that an independent team following the same process will find the same conclusions.

Repeatability of the systematic literature review process have been confirmed ; but only with teams of domain and systematic review experts [2]. The results of novices are much less repeatable, casting doubt on the capability of novices to perform publication-grade quality literature reviews [1]. Repeatability ensures that independent teams following the same systematic literature review process in the same field would reach the same conclusion. Poor repeatability can throw serious doubt on the scientific value of the review.

Completeness has not been clearly confirmed : The lack of reporting standards in the domain makes it difficult to define an adequate search strategy [4]. Different authors use different definitions for the same concept, making the creation of an appropriate search string nearly impossible for many topics. Completeness ensures that the systematic literature review covers the entire area and is not a reflection of some handpicked literature. Poor completeness may indicate that the selection of the study was biased to draw specific conclusions.

To achieve both completeness and repeatability, many authors recommend the reiterations of certain tasks [5]. But in all the reported cases, the iterations remain limited to a set of specific tasks, mostly related to question definition and search strategy. As will be shown in this paper, iterations are beneficial for other tasks, like data extraction, analysis and synthesis, especially when novices are involved. This is because repeatability and completeness are related to the entire review process, from the identification of the problem to the writing of the conclusions.

The iSR process is based on the theory of experiential learning, where procedural knowledge is acquired as the tasks are performed, or as Kolb writes :

> [Learners] must be able to involve themselves fully, openly, and without bias in new experiences ; reflect on and observe these experiences from many pers-

pectives ; create concepts that integrate their observations into logically sound theories ; and use these theories to make decisions and solve problems. [6]

The iSR approach defines these experiences in eight tasks :

1. Review planning : Plan the review effort and training activities,
2. Question formulation : Define the research questions,
3. Search strategy : Define the review scope and search strings,
4. Selection process : Define inclusion and exclusion criteria,
5. Strength of the evidence : Define what makes a high quality paper,
6. Analysis : Extract the evidence from the selected papers,
7. Synthesis : Structure the evidence in order to draw conclusions,
8. Process monitoring : Ensure the process is repeatable and complete.

E.3.1 Review Planning in the Literature

The literature recommends that the following topics be presented to novice students at some point during the review process : SLR, along with an explanation of its use, depending on the focus of the research [7]–[12] ; Literature review planning [7], including a planning tutorial [7]–[12] ; An introduction to the scientific method [10], to help students understand what makes a high quality paper ; Approaches to empirical studies [10], [12] ; A brief overview of common statistical calculations [10], [13] ; An introduction to common research biases [10].

On the teaching of statistics, Brereton et al. [13] say that *"some statistical knowledge is needed and we found that training in basic statistical techniques, such as proportions, confidence intervals, and significance levels, was required."*

E.3.2 Question Formulation in the Literature

The formulation of the research question can be initiated with a very generic question, such as *"What has already been written on subject X ?"* [5]. More targeted questions can be introduced as the domain becomes better understood [5], [14]. MacDonell et al. proposes that question writing should follow the PICO approach of Population, Intervention, Comparison intervention and Outcome [2]. Question formulation has been found by many authors to be a very difficult a task [5], [7], [9], [12]. They suggest a four-step procedure for writing good research questions [15] : Identify the problem ; Write down the relevant definitions ; Write down the assumptions made ; Identify your own preconceived ideas (hypotheses).

E.3.3 Search Strategy in the Literature

The construction of a search string often uses the "population AND intervention AND outcome" structure [13], [16]. Building a good search string involves the same paradox as building a good research question. In Brereton's words [7] :

> It is quite a challenge, especially when you are not an expert in the topic of study. Of course the data should come from the studies, but it is hard to establish the best search strings until you are familiar with the topic.

The challenge of building a good search string is compounded by the fact that database search engines do not use a standardized language. To alleviate these problems, Rainer and Beecham present a short iterative procedure for the construction of a search string [15] ; an initiative also supported by Oates and Capper, who state that *"the search strategy is likely to be adjusted as the results are inspected and the research question evolves"* [5].

A good search string should retrieve all the papers found in a SLR of the same domain (per the recall metric), as well as retrieve as little irrelevant papers as possible (per the precision metric). This "quasi-gold standard" string evaluation approach is however only possible in domains where a quality SLR exists [17].

An alternative to search strings, the snowballing approach, starts with a body of high-quality relevant papers. It then searches which papers are in the reference list of these starting papers (backward snowballing), and which papers cite these starting papers (forward snowballing) [18]. However, the snowballing approach can be very sensitive to the starting papers chosen [19]. Therefore, a mix of database searches and snowballing seems to be preferable.

E.3.4 Selection Process in the Literature

The selection process typically uses the following steps, each with their own inclusion and exclusion criteria : (1) Selection based on the paper title [13], [16] ; (2) Selection based on the paper keywords [13] ; (3) Selection based on the paper abstract [8], [13], [16].

Brereton et al. [13] give an example of a case where an exclusion criterion became evident only after the review was well under way. Inclusion and exclusion criteria should therefore be periodically revised. Another problem is that one study can be presented in multiple papers, and similarly, one paper can present multiple studies [13]. This can cause a study to be over-represented, as they are used for multiple analyses across different papers.

The literature reports that novices can either be too restrictive [9] or not restrictive enough [20] in their choice of inclusion and exclusion criteria. Instructors should therefore keep a

close eye on the progress of the selection process, in order to ensure that the criteria are clearly and fully defined. Inter-rater agreement metrics like the Cronbach Alpha [21] and the Fleiss' Kappa [22] can be used to assess the quality of a selection process.

E.3.5 Strength of the Evidence in the Literature

Many authors [5], [9], [13] consider this task to be the most difficult for novices to perform. According to Kitchenham [8], *"Students found evaluating the quality of studies found in a systematic review particularly problematic."* As a result, Brereton et al. decided to skip this activity during their 2011 review [7]. As they describe it :

> "Students were not expected to assess study quality, an activity considered especially challenging, even for experienced researchers."

They affirm that novice reviewers are not accustomed to the way empirical papers are written. Even the use of a simple checklist does not improve the situation. Rainer et al. report that some *"students simply did not use the checklist, a number of students used it poorly, whilst some students used it well. The varied use of the checklist is surprising, as it had already been used in some tutorials and lectures"* [12].

This means that research quality evaluation is often based on the student's perception, rather than on the evidence in the paper. The quality evaluation thus becomes a matter of personal opinion.

E.3.6 Analysis in the Literature

Strauss [23, p. 19] writes that qualitative data collection and codification should follow an iterative and incremental approach. The data collected, the code used to qualify the data, and the memos used to organize the codes must be constantly refined as the selected articles become better understood.

Petticrew and Roberts [14, p. 165] recommend focusing on the studies having the highest research quality. The main synthesis work (and conclusions) should be based on the data extracted from these top quality papers, with the lower quality ones used for either confirmation or to support minor conclusions.

They also recommend not transforming qualitative data into quantitative data [14, p. 191]. Practices like tally counting can be useful, but they result in a large amount of important contextual information being neglected, and should be used with care. The works of Hannay

et al. show how one can adequately pool data from contextually different studies into a single meta-analysis [24].

Brereton et al. [7] maintain that a "key problem" in data extraction is that students have to know which data are relevant, even though they are domain novices. The extraction must therefore be adjusted as the need arises and as the students' comprehension of the field improves.

E.3.7 Synthesis in the Literature

The activity of producing a synthesis has been reported as difficult by many authors [8], [16]. One of the reasons stems from the use of data extracted by other reviewers. Baldassarre et al. describe filling out the data aggregation forms as *"a demanding task, as it requires combining the individual work of a number of students"* [16].

The poor quality of syntheses in literature reviews reported by Cruzes and Dyba [25] can be traced back to the existing review processes, which provide very few details on how to perform a synthesis. Consequently, there is a real need for a defined synthesis procedure.

Janzen and Ryoo [26] present a successful synthesis process when working with students. They asked each of their students to read 17 articles and summarize them. The synthesis approach was to write one global summary based on the 17 individual article summaries. The quality of the summaries produced proved to be equal to or better than that of the summaries produced by experts. This successive summarization approach also proved useful to initiate systematic review syntheses.

Another synthesis approach seen in the literature is to use a validated model, such as a recognized domain taxonomy, as a basis for the categorization of the evidence [6].

E.3.8 Process Monitoring in the Literature

This task is not defined in the traditional SLR processes. The 'big upfront' approach to traditional literature reviews implies that all resources are assigned at the beginning and cannot be easily moved during process execution. This task evaluates the repeatability and completeness of the review results and produce recommendations for the next iterations.

E.4 Contexts of the Performed Reviews

To build and validate the iSR approach, four systematic reviews in four different domains were performed, as described in Table I. All reviews were performed with students within one

semester of a software engineering graduate course. The goal was to introduce the students to the state of the art in a specific domain : the whole class therefore worked as a single team.

Students attended a weekly three hour lecture followed by a home assignment to be realized individually or in pairs. The lecture was split in three parts : the first hour the instructor provided a feedback on the previous week's assignment ; the second hour the instructor introduced new concepts ; and the third hour students and instructor discuss issues seen in the results and plan the next assignment.

The first review, performed during the Fall 2010 semester, involved domain and process novices. The domain targeted was the impact of software process improvement on developers. The review was successful and the results were published [27]. Nevertheless, a number of problems emerged, mostly at the analysis and synthesis stages, which required some rework of the analysis and synthesis tasks.

The second review was performed during the Fall 2011 semester, and involved domain and process novices. The domain targeted was the use of Web-based tools during software development. The review failed, mostly because the size of the body of research on distributed (or global) software development had been underestimated. The results did not provide an accurate synthesis of the domain, and were found to be too biased to be pursued.

It was the problems found in the Fall 2010 and Fall 2011 reviews that motivated the development of the iSR approach. To validate this new approach, a third review was conducted during the summer of 2012. The objective of this review was to perform a synthesis of the various definitions of Pair Programming practices reported in software engineering studies. The literature review was conducted by two graduate students familiar with SLR processes, but not with the iSR approach. They were domain novices. The review was successful, and, through multiple iterations, they managed to produce a good summary of the targeted domain. The review results led to a paper submission.

The fourth review was performed during the Fall 2012 semester by three graduate students, all of whom were domain novices and process novices. The domain targeted was the research of information on the Web by software developers. The review demonstrated the usefulness of an iterative approach, as the research questions had to be adjusted multiple times during the review to account for the state of the domain. The review results have since been integrated into a paper.

Table E.1 Contexts of the SLR Performed to Build and Test the iSR Process.

Fall 2010	Software process improvement	5	Novices	Novices	Biolchini et al. [28]	Partial success (repeatable but incomplete) : Analysis and synthesis had to be completely redone.
Fall 2011	Web tool use	14	Novices	Novices	Biolchini et al. [28]	Failure (non-repeatable and incomplete) : Bad research question was not corrected, resulting in poor inclusion and exclusion criteria.
Summer 2012	Pair programming	2	Proficient and expert	Novices	iSR	Success (repeatable and complete).
Fall 2012	Web information lookup	3	Novices	Novices	iSR	Success (repeatable and complete).

E.4.1 Evaluation of the Students

Students were graded on the fulfillment of the assignments planned for the week : Gradation was based on the process activities instead of the content. Disagreements on the content were discussed in class during plenary sessions, which typically resulted in changes in the process and the task assignments for the following week. Relevant interventions during plenary sessions require that students master self-evaluation capabilities. Self-evaluation is most efficient when the students execute the process tasks over and over, hence the importance of iterations.

E.5 Lessons learned of the iSR Approach

This section presents the lessons learned from the eight tasks comprising the iSR approach. Figure 1 presents an artistic view of the effort expanded on the eight tasks of an iSR process composed of ten iterations. The curves for each task are not to scale.

E.5.1 Planning and Training

This task has two objectives : First, to plan the review work, and second, to plan training sessions and tutorials, to provide novice students with the technical know-how required to perform a literature review.

All documentation submitted to the students should be carefully written, as the repeatability of the process requires clarity and understandability. Any ambiguous statement can result in divergent results, which are detrimental to repeatability. The use of an iterative approach resolves part of this problem, by offering opportunities to correct any misunderstanding.

The seminar presentations, tutorial, and exercises should follow the first steps of Bloom's taxonomy of educational objectives [29], in order to progressively introduce the concepts. The following three levels should be considered :

— **Remember** : Ask students to follow instructions by rote ;
— **Understand** : Ask students to provide a feedback on the appropriateness of the instructions given ;
— **Apply** : Ask students to tailor the instructions to the context at hand.

During the systematic review process, the students will need to differentiate strong search strings from weak ones, high-quality studies from biased ones, etc. It is important to teach students how to perform self-evaluations of their work. These quality evaluations require a high level of understanding, something which cannot be achieved with the rote application

of a method.

All the required knowledge should not be transmitted to the novice students directly at the beginning. For example, the importance of distinguishing high quality from low quality papers does not emerge until a number of relevant papers have been selected. Tutorials on the scientific method can therefore be delayed until the concepts are needed.

The reviews showed the importance of adapting the work plan. Some domain idiosyncrasies only become known during the execution of the review, which requires reiterating on the work performed. The review planning therefore needs to be regularly revised.

Another problem is the pressure to meet a deadline. It is tempting to cut corners in order to complete the review process. It is however better to have an uncompleted review containing good quality data that can be completed later on by another team, than a botched review that is useless since it needs to be redone.

E.5.2 Question Formulation

The aim of this task is to define research questions and review objectives. Calibrating the research question is essential, in order to obtain a manageable, yet significant, body of literature. An SLR based on hundreds of papers is not pragmatic, since it could take years to analyze and synthesize.

The question formulation task suffers from a paradox : The domain must be well understood in order to produce good research questions. But, to understand the domain, the research must first be performed. The main cause of the failed review in the Fall 2011 semester was the fact that the question was not suited to the current state of the domain. Redefining the research questions is useful when the current state of the domain does not match initial expectations, which is a frequent issue when domain novices are involved. Consequently, research questions must be iteratively readjusted.

E.5.3 Search Strategy

This task's objective is to define the journals and conferences that will be targeted, the databases that will be searched, whether or not snowballing will be applied, whether or not "grey" literature [1] will be searched, etc.

The 'term impact analysis' string validation method consists of testing the search string with and without each term to evaluate how they affect the results. A term with no impact on the

1. Grey literature comprises unpublished papers, or papers published in non-peer reviewed journals. These items mostly consist of technical reports, the quality of which can vary from excellent to mediocre.

results can be safely discarded. A term responsible for a large part of the results might be too generic and could be refined. This methodology could resolve the problem that *"students provided poor explanations in their reports of how their searches were conducted"* [11].

Another problem with the search strategy is that reviewers tend to perform a typical Google-type search. They are looking for the one exact answer to the research question, instead of looking for all the relevant answers.

There were also some technical problems inherent in building search strings. The most common problems found were : inappropriate use of wildcard characters (?, *) ; orthographical errors ; subsumed terms ("software" OR "software engineering") ; and unbalanced strings ("software" OR "test-driven development"). Most of these problems can be detected by a validation technique like term impact analysis.

Iterations based on a pilot approach also works well for building the search string. A small sample of the papers found with a search string can be transferred to the selection, analysis, and synthesis tasks. This can help the reviewer spot relevant and irrelevant keywords which were not included in the initial search string.

Finally, some of the important keywords are not immediately obvious, and only become clear as they are seen in the papers. The search string must therefore be adjusted each time a new keyword emerges as either relevant or irrelevant. The major risk is therefore to find new, relevant keywords late in the review process.

E.5.4 Selection Process

Existing literature review processes often mix the quality in terms of the relevance of the paper for the purposes of the review, and quality in terms of the paper's research methodology. "Relevance quality" is measured during the Selection Process task. "Research quality" is measured during the Strength of Evidence task.

Relevance quality is determined by the inclusion and exclusion criteria, which defines which papers are relevant and should be kept, and which papers are off topic and should be rejected. Two guidelines are proposed for this task :

— The selection-by-title stage rejects only duplicates titles and proceedings introductions since there is often little information in a title.
— The selection-by-abstract is based on the relevance of the content.

An "overview step" is needed and consists of looking at the tables, figures and conclusions of the paper. The main purpose is to perform a final relevance quality evaluation, along with a preliminary research quality evaluation. [7].

Most review processes suggest the use of a spreadsheet application like Excel to briefly document the inclusion or exclusion rationale of each paper. These justifications will help in the revision of the inclusion and exclusion criteria.

Misinterpretation of inclusion and exclusion criteria explains some of the differences found in reviews performed by novices [1]. When misinterpretations occur in the selection process, it must be discussed freely and openly with the students, to establish the reasons for the poor results. These discussions enable the instructors and students to synchronize their views on the value of the papers. Students were rarely to blame for poor inter-rater agreement ; the culprit being poorly worded assignments and ambiguous tasks.

However, Excel is not entirely appropriate for collaborative work, and synchronization proved problematical when two students worked in parallel.

The third literature review, on Pair Programming (PP), generated some discussions on what constitutes a PP paper and what does not. Since the definition of the practice proved to be very fluid, many papers initially thought to be irrelevant were reconsidered. The importance of this selection criterion did not emerge until synthesis began. Thus, one benefit of the iterative approach is that not all the resources were expended at the start of the synthesis task. A common issue of this task is the definition of overly restrictive inclusion and exclusion criteria, typical of a Google-type search.

E.5.5 Strength of the Evidence

This task is aimed at evaluating the research quality of the selected papers. The process uses a two-step approach for the research quality evaluations. The first step is to produce the overview which consists of the verification of the presence of three critical context details : A description of the sample and its population ; A description of the study context ; Clearly written study conclusions.

The second step is a thorough quality evaluation, using a checklist based on the works of Dyba and Dingosyr [30]. Filling out this form requires a thorough reading of the full text, however, and this task is typically performed in parallel with the Analysis task. Each element of the checklist uses a three level evaluation : (0) the element is absent or very poor ; (1) the element is present but could be more detailed ; (2) the element is present and sufficiently detailed.

While previous studies reported problems with the use of a checklist [12], it was found that multiple iterations on research quality evaluation resolved these issues.

E.5.6 Analysis

In this task, the papers are analyzed in order to extract the relevant data. This analysis is typically performed using a provided extraction form. The data extracted is not always the data needed for the synthesis. This is because it is not known beforehand which data will be required for the synthesis. In many cases, the extraction had to be redone once the synthesis approach had been clarified.

Providing novice reviewers with a sample completed form helped them understand what is required in each field. The form needs to be sufficiently detailed in order to ensure that the results are comparable from one reviewer to another. Students should also focus on the conclusions of the studies, as they often provide the 'meat' of the synthesis.

According to qualitative feedback from the reviewers, this step is the most time-consuming activity of the process, but not the most mentally strenuous. In light of this, extraction work should start as soon as possible, in order to ensure that reviewers are familiar with the extraction procedures and the form, and what is required of them. Early extraction can also avoid a sudden drop in extraction quality toward the end of the review, when deadline pressures loom.

In addition, some extractions tend to be too succinct and others too verbose. Constant evaluation and feedback on the results is therefore required, to ensure that reviewers understand what is expected of them.

A learning iteration is recommended for novice reviewers, where they are assigned a short synthesis exercise based on good and bad extractions performed by other reviewers. The goal of such an exercise is to clarify what differentiates a good extraction from a bad one.

Another approach used was to perform the analysis in stages. During the first analysis iteration, a small batch of papers is analyzed and an attempt is made to synthesize the extracted data. The extraction form is then modified based on the synthesis results. During subsequent iterations, the data extraction process is corrected, and the analysis is then performed on a larger batch of papers. This approach proved fruitful, as in most cases the final extraction form was very different from the initial one.

E.5.7 Synthesis

The reviews show that synthesis benefits greatly from an iterative approach. To find some structure in a large amount of qualitative data requires many attempts. Each attempt might also call for a different view of the data, which might in turn require a revision of the extraction form. In some cases the data extracted did not support some interesting synthesis

that had emerged. This required the reviewers to revisit the source papers to extract the missing information.

Students are typically expecting Google-type search responses. As a result, they discard many articles as irrelevant, even though they provide interesting answer fragments.

The synthesis approach which provided the best results in the reviews involves successive summarization, much like Janzen and Ryoo's approach [26]. The following are the four steps in the approach :

1. Each selected paper is summarized in a single paragraph, based on its extraction form.
2. Each summary paragraph is further summarized into a single phrase.
3. The summary phrases from all the selected papers are put together to see how they support or contradict one another.
4. A single paragraph is built based on the phrases summarizing all the selected papers.

At every stage of the approach, the cohesiveness of the results is monitored. The phrases and paragraphs built must make a coherent whole. The resulting paragraph does not consider the context of the studies, and should not be used as-is in the review synthesis. The objective of this exercise is limited to finding a potential structure for the evidence found in the selected papers. Feedback from the students underlines the fact that the synthesis activity is not especially time-consuming, but it is mentally strenuous. The successive summarization approach helps the students find the relationships between the pieces of evidence, but building a conceptual map of the evidence still involves a significant mental strain.

E.5.8 Process Monitoring

This task is related to the collection of data during the systematic review. Its goal is to improve the process for future reviews and to plan the resources needed for the forthcoming iterations. The results obtained are not always good enough to warrant a transition to the next task. Mistakes made in the previous tasks must be corrected before more effort can be invested in the next ones, to ensure that results reflects the state of the literature (completeness) and can be redone with similar results (repeatability).

This paper is the result of multiple improvements to the systematic review processes currently available. These improvements were made possible by gathering data during every task of the process. Reviewers were required to provide the time spent on the tasks, and to provide feedback on the difficulties encountered. These reports enabled us to find problems and propose solutions to common issues arising in systematic review processes.

Solutions like the term impact analysis for search string validation and the successive summarization approach for the synthesis of the evidence were introduced because of reported issues with these tasks.

E.6 Conclusions

Lessons learned from multiple systematic reviews demonstrate that an iterative approach can be beneficial when working with domain and process novices. As the review progresses, the perception of the domain by novices changes, and the design of the review should evolve accordingly. The research questions may have to be rewritten, the selection procedure may need some adjustment, the extraction forms and analysis tables may require revision, and the synthesis conclusions may need to be redesigned. This approach should produce better and more accurate results iteration after iteration, reflecting the progressive gain in expertise and understanding of the novices. It should also enable instructors to calibrate the effort output required through the addition or removal of iterations.

E.7 References

[1] B. Kitchenham, P. Brereton, Z. Li, D. Budgen, and A. Burn, "Repeatability of systematic literature reviews," in 15th Int. Conf. on Evaluation and Assessment in Software Engineering (EASE 2011), Durham, UK, 2011, pp. 46–55.

[2] S. MacDonell, M. Shepperd, B. Kitchenham, and E. Mendes, "How Reliable Are Systematic Reviews in Empirical Software Engineering ?," IEEE Trans. Softw. Eng., vol. 36, no. 5, pp. 676-687, Sept.-Oct. 2010.

[3] B. Kitchenham, P. Brereton, and D. Budgen, "Mapping study completeness and reliability - A case study," in IET Seminar Digest, vol. 2012, pp. 126-135, 2012.

[4] O. Dieste, A. Griman, and N. Juristo, "Developing search strategies for detecting relevant experiments," Empir. Softw. Eng., vol. 14, no. 5, pp. 513-539, Oct. 2009.

[5] B. J. Oates and G. Capper, "Using systematic reviews and evidence-based software engineering with masters students," in 13th Int. Conf. on Evaluation and Assessment in Software Engineering (EASE 2009), Durham, UK, 2009, pp. 79-87.

[6] F. O. Bjørnson and T. Dingsøyr, "Knowledge management in software engineering : A systematic review of studied concepts, findings and research methods used," Inf. Softw. Technol., vol. 50, no. 11, pp. 1055-1068, Oct. 2008.

[7] P. Brereton, "A Study of Computing Undergraduates Undertaking a Systematic Literature Review," IEEE Trans. Educ., vol. 54, no. 4, pp. 558-563, Nov. 2011.

[8] B. Kitchenham, P. Brereton, and D. Budgen, "The Educational Value of Mapping Studies of Software Engineering Literature," in 32nd Int. Conf. on Software Engineering (ICSE 2010), Cape Town, South Africa, 2010, pp. 589-598.

[9] A. Rainer and S. Beecham, "A follow-up empirical evaluation of evidence based software engineering by undergraduate students," in 12th Int. Conf. on Evaluation and Assessment in Software Engineering (EASE 2008), Bari, Italy, 2008, pp. 78-87.

[10] M. Jorgensen, T. Dybå, and B. Kitchenham, "Teaching Evidence-Based Software Engineering to University Students," in 11th Int. Software Metrics Symp. (METRICS 2005), Como, Italy, 2005, pp. 213-220.

[11] A. Rainer, S. Beecham, and C. Sanderson, "An assessment of published evaluations of requirements management tools," in 13th Int. Conf. on Evaluation and Assessment in Software Engineering (EASE 2009), Durham, UK, 2009, pp. 98-107.

[12] A. Rainer, T. Hall, and N. Baddoo, "A preliminary empirical investigation of the use of evidence based software engineering by under-graduate students," in 10th Int. Conf. on Evaluation and Assessment in Software Engineering (EASE 2006), Keele University, UK, 2006, pp. 91-100.

[13] P. Brereton, M. Turner, and R. Kaur, "Pair programming as a teaching tool : a student review of empirical studies," in 22nd Conf. on Software Engineering Education and Training (CSEET 2009), Hyderabad, India, 2009, pp. 240-247.

[14] M. Petticrew and H. Roberts, Systematic Reviews in the Social Sciences. Malden, MA : Blackwell Publishing Ltd, 2008.

[15] A. Rainer and S. Beecham, "Supplementary Guidelines and Assessment Scheme for the use of Evidence Based Software Engineering," University of Hertfordshire, Hertfordshire, UK, Tech. Rep. CS-TR-469, 2008.

[16] M. T. Baldassarre, N. Boffoli, D. Caivano, and G. Visaggio, "A Hands-On Approach for Teaching Systematic Review," in 9th Int. Conf. on Product-Focused Software Process Improvement (PROFES 2008), Rome, Italy, 2008, pp. 415-426.

[17] H. Zhang, M. A. Babar, and P. Tell, "Identifying relevant studies in software engineering," Inf. Softw. Technol., vol. 53, no. 6, pp. 625-637, Jun. 2011.

[18] S. Jalali and C. Wohlin, "Systematic literature studies : database searches vs. backward snowballing," in Proc. of the 6th ACM-IEEE Int. Symp. on Empirical Software Engineering and Measurement (ESEM '12), Lund University, Sweden, 2012, pp. 29-38.

[19] M. Skoglund and P. Runeson, "Reference-based search strategies in systematic reviews," in 13th Int. Conf. on Evaluation and Assessment in Software Engineering (EASE 2009), Durham, UK, 2009, pp. 31-40.

[20] P. Brereton, M. Turner, and R. Kaur, "Pair programming as a teaching tool : a student review of empirical studies," in 22nd Conf. on Software Engineering Education and Training (CSEET 2009), Hyderabad, India, 2009, pp. 240-247.

[21] L. J. Cronbach, "Coefficient alpha and the internal structure of tests," Psychometrika, vol. 16, no. 3, pp. 297-334, Sept. 1951.

[22] J. L. Fleiss, "Measuring nominal scale agreement among many raters," Psychol. Bull., vol. 76, no. 5, pp. 378-382, Nov. 1971.

[23] A. L. Strauss, Qualitative Analysis for Social Scientists. Cambridge, United kingdom : Cambridge University Press, 1987.

[24] J. E. Hannay, T. Dyba, E. Arisholm, and D. I. K. Sjoberg, "The effectiveness of pair programming : a meta-analysis," Inf. Softw. Technol., vol. 51, no. 7, pp. 1110-1122, Jul. 2009.

[25] D. S. Cruzes and T. Dybå, "Research synthesis in software engineering : A tertiary study," Inf. Softw. Technol., vol. 53, no. 5, pp. 440–455, May 2011.

[26] D. S. Janzen and J. Ryoo, "Seeds of Evidence : Integrating Evidence-Based Software Engineering," in 21st Conf. on Software Engineering Education and Training (CSEET 2008), Charleston, USA, 2008, pp. 223-230.

[27] M. Lavallée and P. N. Robillard, "The Impacts of Software Process Improvement on Developers : A Systematic Review," in Int. Conf. on Software Engineering (ICSE 2012), Zurich, Switzerland, 2012, pp. 113-122.

[28] J. Biolchini, P. Gomes Mian, A. Candida Cruz Natali, and G. Horta Travassos, "Systematic Review in Software Engineering," PESC, Rio de Janeiro, Brazil, Tech. Rep. RT-ES 679/05, 2005.

[29] B. S. Bloom, M. D. Engelhart, E. J. Furst, W. H. Hill, and D. R. Krathwolh, Taxonomy of educational objectives : the classification of educational goals. United Kingdom : Longman Group, 1956.

[30] T. Dyba and T. Dingsoyr, "Empirical studies of agile software development : A systematic review," Inf. Softw. Technol., vol. 50, no. 9-10, pp. 833-859, Aug. 2008.

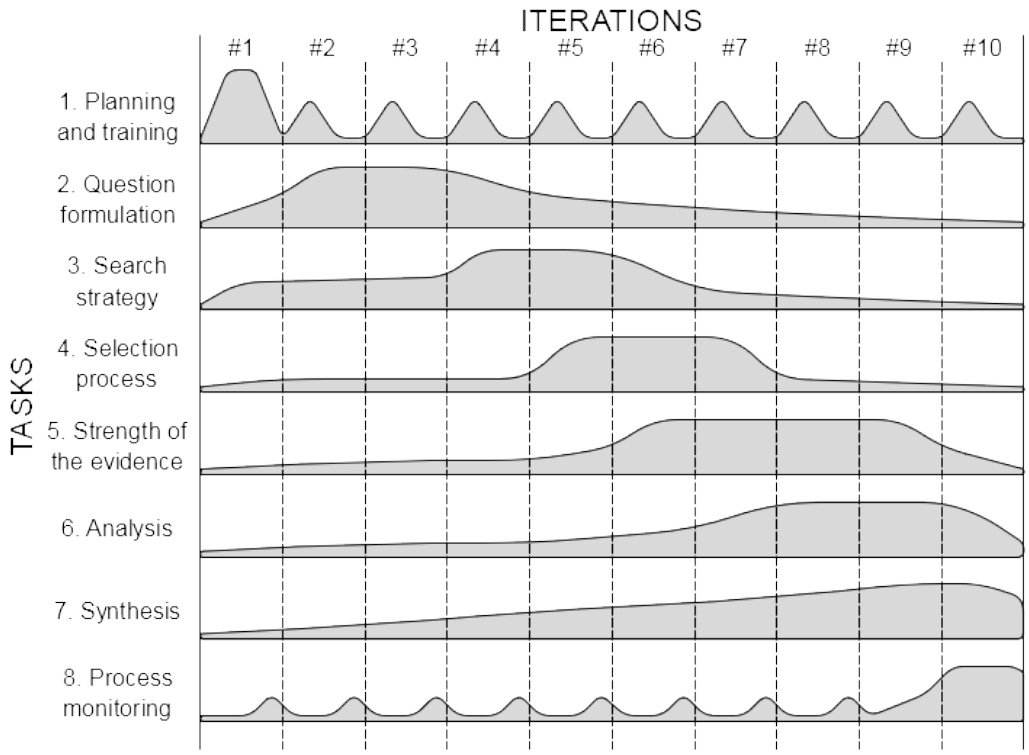

Figure E.1 Example of the iSR process in action. The shaded area represents an artistic view of the effort expanded on each task for a given iteration.

ANNEXE F ARTICLE 5 : THE IMPACTS OF SOFTWARE PROCESS IMPROVEMENT ON DEVELOPERS : A SYSTEMATIC REVIEW

Type de publication	Référence complète
Conférence	*The Impacts of Software Process Improvement on Developers : A Systematic Review*, **Mathieu Lavallée** et Pierre N. Robillard, International Conference on Software Engineering (ICSE), 2012, doi :10.1109/ICSE.2012.6227201

F.1 Abstract

This paper presents the results of a systematic review on the impacts of Software Process Improvement (SPI) on developers. This review selected 26 studies from the highest quality journals, conferences, and workshop in the field. The results were compiled and organized following the grounded theory approach. Results from the grounded theory were further categorized using the Ishikawa (or fishbone) diagram. The Ishikawa Diagram models all the factors potentially impacting software developers, and shows both the positive and negative impacts. Positive impacts include a reduction in the number of crises, and an increase in team communications and morale, as well as better requirements and documentation. Negative impacts include increased overhead on developers through the need to collect data and compile documentation, an undue focus on technical approaches, and the fact that SPI is oriented toward management and process quality, and not towards developers and product quality. This systematic review should support future practice through the identification of important obstacles and opportunities for achieving SPI success. Future research should also benefit from the problems and advantages of SPI identified by developers.

F.2 Introduction

A company's software development process describes how its software products come to life, and, more specifically, how developers bring them to life. Development processes are rooted in the developers' work, and so improving those processes is very often dependent on acceptance on the part of developers of the need for improvement and their support for it. Success factors in the Software Process Improvements (SPI) field have been the subject of research for many

years [1]. Of the factors confirmed by previous research, adoption of the improved process by developers has, not surprisingly, been deemed significant [2]. However, research on the developers themselves is limited to the narrow field under study. So, what about the big picture ? How are developers treated by SPI initiatives as a whole ? What are the impacts of these initiatives on the software developer ? The goal of this systematic review is to answer this last question.

This paper reviews the literature using the Systematic Review approach developed by Kitchenham [3] and refined by Biolchini et al. [4]. Systematic Literature Reviews (SLR) are different from common literature reviews by their formalism. The goal of a SLR is to provide a reproducible summary of the state of a problem or field of study. The conclusions obtained by a SLR should therefore represent an accurate and objective assessment of the situation, given the body of literature at the time of the review. This SLR was performed by the two authors, with the assistance of four graduate students.

Section II details the methodology used. Section III presents a quality overview of the selected studies. Section IV describes the Ishikawa model used, including the SPI factors identified as potentially having either positive or negative impacts on developers. Section V presents the supporting evidence from the selected studies. Section VI presents other systematic reviews in the SPI field, and how we position our work in this field. Section VII presents our conclusions, and outlines the main findings of this review.

F.3 Methodology

The following subsections describe the process used for this formal systematic review, which was based on the technical report written by Biolchini et al. [4].

F.3.1 Data Sources

Study selection was restricted to a number of journals, conferences, and a workshop which are arguably the most important resources for SPI research. Restriction of the research in this way proved rewarding in terms of quality, as very few studies were rejected for quality reasons. The selected journals are listed in Table I. The "#" column shows the number of studies selected in the corresponding source.

F.3.2 Search String

The search string used in the various data sources was adapted from a common concept plan, shown in Table II. Ideally, all the concepts were applied to each data source. However, the application of all concepts proved too restrictive for certain data sources. In these cases, combinations of concepts from the plan were tested in the data source, until a specific combination returned a sufficient number of studies ($\geq$20).

F.3.3 Selection Process

A three-stage approach was used in the selection process. The first stage involved the selection of studies based on their titles. At the title level, obvious irrelevant studies were removed. If there was any doubt about selection based on the title alone, the study was kept.

The second stage involved the selection of studies based on their abstracts. At the abstract level, the focus was on selecting empirical studies. Purely theoretical models and historical recapitulation were excluded at this stage, as our goal with this review is to provide conclusions based on empirical data.

The last stage involved the selection of studies following a summary overview. The overview focused on the main elements of the studies : introductions, conclusions, tables, and figures. The goal at this stage was to perform a preliminary quality evaluation to determine whether or not the studies' conclusions were justified.

F.3.4 Extraction Process

The studies remaining at this stage were carefully read. Any conclusion pertaining to developers was copied into a repository for further analysis. The evidence was extracted by the individual reviewers, using a standardized extraction process based on previous work by Dyba and Dingsoyr [5], and then reviewed formally by one of the authors.

F.3.5 Grounded Theory Approach

Once all the selected studies had been read, the copied conclusions were associated with a set of codes, following the grounded theory approach. This approach was chosen because it had already been used successfully in the software engineering field (S12, S18). Practitioners seeking more information on the approach are referred to the works of Strauss [6] and Glaser [7]. The codes were further grouped into concepts, which formed the seven factors presented in section III. These seven factors were matched with different models, but were found to be

Table F.1 Data Sources and Number of Selected Studies

Source	#
Journals	
- Empirical Software Engineering	1
- IEEE Software	3
- IEEE Transactions in Software Engineering	2
- Information and Software Technology	4
- Journal of Systems and Software	3
- ACM Transactions on Software Engineering and Methodology	1
- Software Process : Improvement and Practice[a]	7
Conferences	
- International Conference on Software Engineering	2
- International Conference on Software Maintenance	1
- EuroSPI[b]	2
- International Conference on Software and Systems Process	0
Workshop	
- International Software Process Workshop	0
	26

[a] Now included in the journal of Software Maintenance and Evolution.
[b] Also known as European System & Software Process Improvement and Innovation.

Table F.2 Concept Plan

Concept	Search String
Process	"software process" OR SPI OR "software development"
Standard Model	"capability maturity model" OR CMM* OR SPICE OR 15504 OR 900*
Developers	programmer* OR developer* OR coder* OR engineer* OR practitioner*
Impact	practice* OR impact* OR benefit* OR ROI OR strateg*

closer to the basic six categories of the Ishikawa ("Fishbone") Diagram. The choice of the Ishikawa categories to represent the results was made after the codification of the conclusions.

The goal of the six categories of Ishikawa Diagrams is to cover all the possible root causes of a specific problem [8]. The Resources, Process, People, Materials, Environment, and Management categories are typical of Ishikawa Diagrams and were kept by the authors.

F.3.6 Validity of the Process

The entire review process was planned and executed by the authors and four graduate students. An initial iteration on the journal Software Process : Improvement and Practice showed unacceptable discrepancies between the reviewers' selections, mainly as a result of ambiguities in the process. A second iteration on the improved process presented herein was performed by the same reviewers and showed much improved results. Extraction was first performed by the reviewers, and the evidences obtained were then confirmed by the authors.

Coding and grouping into factors was performed by the authors. However, this review presents the raw conclusions, with no global meta-ethnographic inferences. The grounded theory groupings and the Ishikawa diagram were only used to give structure to the evidence, and to help us understand the data extracted. In fact, the Ishikawa categories must remain the same, even though another reviewer may create his own codes and groupings.

The evidence in this review is presented as objectively as possible. We do not present meta-ethnographic conclusions, but let the conclusions of the selected studies speak for themselves. There is therefore no publication bias, nor is there a hindsight bias. While some fringe evidence could be moved from one factor to another, there is still a significant body of knowledge supporting each factor. Therefore, our belief is that subjectivity in this review was kept to a minimum.

F.4 Quality of the Selected Studies

Almost all the studies selected for extraction were of high quality. Quality evaluation was performed using a standard form adapted from the work of Dyba and Dingsoyr [5]. Their original quality evaluation form was based on eleven elements worth one point each. Four of these elements did not provide relevant information in our case (e.g. use of a control group). Our adapted quality evaluation was thus limited to seven elements, with each worth zero (null), one (poor) or two (adequate) points. To obtain the two points, the element must answer the main question (showed in roman type in Table III), as well as the sub questions (showed in italics in Table III). Therefore, the maximum score for our quality evaluation

form, which is presented in Table III, is fourteen.

The acceptability threshold was set at 10/14. Studies below this threshold showed distinct flaws in either their methodology or their conclusions. Few studies were rejected because of their quality, since the research was limited to reputed data sources. Additionally, the worst studies were removed during the overview stage (described in section II.C), which focused on elements 2 (context description), 3 (sampling) and 7 (findings) described in Table III. Two studies below the threshold were nevertheless included, but the evidence extracted was limited to the conclusions supported by the data and the methodology of the research. All the evidence presented in this review is therefore supported by the data in the related studies. Results of the quality evaluation are presented in Figure 1.

The main flaw found in the SPI field is in the reflexivity quality element (element 6 in table III). Bias consideration is often weak, or totally absent, from many studies. This flaw has been examined in greater detail in our previous work [9].

F.5 Factors Influencing Developer Support

From the evidence obtained in the selected studies, seven factors emerged that were found to fit loosely with the standard causes of problems in the Ishikawa Diagram, better known as the Fishbone Diagram, as shown in Figure 2. Use of the Ishikawa Diagram was justified because the factors identified sometimes had positive impacts and sometimes negative ones.

The six main categories of the Ishikawa Diagram : "People", "Management", "Process", Environment", "Resources", and "Materials", represent the potential causes of the problem under study. The seven factors found fall into those six main categories surrounding the problem. The content of the Ishikawa diagram, shown in Figure 2, is as follows :

- Impact of SPI on Developers : The root of the Ishikawa Diagram describes the problem under study. What are the factors that have a positive or a negative impact on software developers ?
- Motivation and Resistance : The people, represented by the developers, constitute the first potential problem factor. SPI implies an adjustment in their normal habits, which may, in turn, lead to resistance to change. What causes this resistance ? How can developers be motivated ?
- Technical Focus : Management's involvement with SPI essentially favors technical approaches. Is this focus on developers' practices justified ?
- Documentation : Process implementation and assessment require a fair amount of documentation. How much overhead do standard processes like the CMMI generate ?

Table F.3 Quality Evaluation Form

Element	Questions
1 Aims of the Research	Is there a clear statement of the aims of the research ? *Does the study present empirical data ?* *Is there a clear statement of the study's primary outcome ?*
2 Context Description	Is there an adequate description of the context in which the research was carried out ? *What are the skills and experience of the staff under study ?* *What are the software processes being used ?*
3 Sampling	Was the recruitment strategy appropriate to the aims of the research ? *Were the cases representative of a defined population ?* *Was the sample size sufficiently large ?*
4 Data Collection	Were the data collected in a way that addressed the research issue ? *Is it clear how the data were collected ?* *Has the researcher justified the methods chosen ?*
5 Data Analysis	Was the data analysis sufficiently rigorous ? *Was there an in-depth description of the analysis process ?* *Has sufficient data been presented to support the findings ?*
6 Reflexivity	Has the relationship between researcher and participants been adequately considered ? *Did the researchers critically examine their own role, potential bias, and influence during : research question formulation, sample recruitment, data collection, and analysis and selection of data for presentation ?*
7 Findings	Is there a clear statement of the findings ? *Are the findings explicit (e.g. magnitude of effect) ?* *Are the limitations of the study discussed explicitly ?*

How do developers react to these documentation needs ?

— Knowledge Sharing : The environment of the developer is his or her team. Good team dynamics, communication, and knowledge sharing are essential for successful software development. What are the advantages of good team communications ? What happens when knowledge does not reach the developer ?

— Effort and Morale : These two causes are always presented as intertwined in the evidence found. Effort and morale are seen as a resource ; excessive effort in the form of overtime and late-night rushes depletes it, as surely as low budget reserves. What is the effect of SPI on these factors ?

— Skills : Another resource provided by developers, a skilled team, can mean the difference between success and failure in a software project. What skills are critical to SPI success ?

— Requirements and Clients : Of the materials provided to developers, requirements are certainly among the most important. How are requirements and client relations perceived by software developers ?

F.6 Results

This section presents the evidence found for each factor identified. The selected studies are presented in Appendix A, at the end of this review. They are identified from S1 to S26, to distinguish them from the references. This scheme is taken from Dyba and Dingosyr's systematic review [5].

F.6.1 Motivation and Resistance

The evidence on developer resistance, shown in Table IV, demonstrates the importance of awareness training during SPI execution. Damian et al. (S1) and Basili et al. (S2) report the fear that the SPI initiatives will result in an increased workload for software developers. Basili et al. also show the need to address developers' concerns when introducing data collection processes. Batista and Dias de Figueiredo (S3), by contrast, report initial resistance in an environment not accustomed to processes :

> Human resources have been shown to be a particularly sensitive critical factor, because the level of maturity was initially very low ; and the small dimension of the [team] did not facilitate a global cultural change, unless it was attempted directly through each member.

Kuilboer and Ashrafi (S4) reveal another source of resistance ; the discrepancy between developers' needs and the effects of the SPI initiatives. They report that :

> Correctness, reliability and intra-operability were the most important factors to the software developers. The impacts of SPI on these three factors do not measure up to their perceived importance.

They blame this discrepancy on the initial focus of the standard, which is to define and control internal processes to the detriment of the external properties of the product.

Basili et al. also report the following : *"There will always be tension between the need to rapidly feed back information to developers and the need to devote sufficient time to do an analysis of the collected data."* (S2).

According to Green et al. (S5), developers are motivated by the perceived usefulness of the process changes. Without perceived results for developers' productivity and/or the quality of the product, support for the SPI cannot be assured. Unfortunately, as Basili et al. (S2) report, it takes time to collect, analyze, and confirm data supporting positive results. This delay erodes developers' support of the SPI initiative.

F.6.2 Technical Focus

The evidence for the Technical Focus factor, presented in Table V, shows excessive emphasis by management on technical approaches to SPI. An evaluation of popular topics by Daily and Dresner (S6) on a process improvement assessment website shows that development stages garnered by far the most interest. Similarly, Basili et al. (S2) report that, at the NASA Software Engineering Laboratory, "many of the studies impacted only the developers." This

Table F.4 Evidence for Motivation and Resistance.

Studies	Evidence
(S1, S2)	Resistance to SPI stems from a perceived increase in overhead.
(S2)	Developers fear that data collection will be used for personnel evaluation.
(S3)	Resistance to change is stronger than expected in a very small team with a history of low maturity.
(S4)	Factors important to developers are not improved by SPI.
(S2)	Good data feedback takes time, but taking time to report results erodes support.
(S5)	SPI must show positive impacts on quality/productivity, otherwise it is perceived as useless.

shows that developers are very often the target of SPI initiatives.

Additionally, a study by Hall et al. (S7) on the causes of development problems shows that there is agreement that technical causes are not to blame :

> Practitioners at all levels (senior managers, project managers and developers) in the three companies consider that organisational issues contribute almost half of all problems in software development. Development problems only contribute a quarter of all problems experienced in the software process.

Focus on technical issues in SPI is therefore unwarranted, since they are only marginally responsible for development problems.

Also, the focus on technical issues does not correlate with business results. A questionnaire-based survey performed by Blanco et al. (S8) shows that the achievement of business goals lags behind the achievement of technical objectives. Process Improvement Experiments (PIE) are begun for technical reasons, without regard to potential business benefits.

Dangle et al. (S9) and Sommerville and Ransom (S10) show that this detachment from business goals is problematic, because long-term support of SPI initiatives depends on them. As Sommerville and Ransom write :

> "To engage industry in technology transfer, we cannot just present technical solutions but must also relate these to the real needs of the business."

Both studies maintain that the link with business goals is essential if the approval required to begin the SPI initiatives is to be obtained.

In terms of standards, the situation is less clear. Catteneo et al. (S11) extend the SW-CMM to encompass business goals. In this 2001 study, they report :

> CMM is a good starting point for conducting an assessment since it embeds a great deal of knowledge on good software engineering practice. However, it is necessary to integrate it with other models. Even the scope of CMMI appears to be still too narrow.

Nine years later, in 2010, Basri and O'Connor (S12) reported an opposite situation from interviews conducted in very small enterprises :

> Current software quality standard objective such as encapsulated in standards such as ISO 9000 are more toward on the management and services of the software development process rather than a software technical issues and product.

While both studies disagree on the shortcomings of common standards, they both agree about their narrowness. Taken together, the two studies show that the technical focus of management should be evaluated, and that the technical importance of SPI initiatives should be balanced with potential business benefits.

F.6.3 Documentation

For the proponents of the Agile Manifesto [10], excessive documentation and processes are a thorn in the side of developers. Examination of the evidence the reviewers found shows a mixed picture, as seen in Table VI : Niazi and Babar (S13) show that developers and managers agree that the documentation of requirements is a key practice of the CMMI. Baddoo et al. (S14) show that developers expect their coworkers to write extensive documentation, but they are not so keen about writing it themselves. Their survey, conducted among software teams, shows that "good developers fully document their work" (rated 7/7). However, they perceive their teammates to be good, but not ideal, documenters (rated 5/7).

Chen and Huang (S15) report that the highest-severity maintenance problems are mostly related to poor documentation. The top five problems reported are :

— *Inadequacy of source code comments* ;
— *Documentation is obscure or untrustworthy* ;
— *Changes are not properly documented* ;
— *Lack of traceability* ;
— *Lack of adherence to programming standards.*

These problems fall squarely in the lap of developers. Limited documentation, therefore, while comforting to developers, can have very negative impacts at the maintenance level.

However, according to Chen and Huang, there is a problem : SPI improves process manage-

Table F.5 Evidence for Technical Focus.

Studies	Evidence
(S2, S6)	Interest in SPI is mostly centered on development stages and developers.
(S7)	Development problems are organizational, not technical.
(S8)	More technical objectives are achieved than business goals.
(S9, S10)	Process should not be implemented for the sake of process, but should focus on business goals.
(S11)	CMM is too focused on engineering, and not focused enough on business goals.
(S12)	Standards focus on management, and not on technical and product issues.

ment and documentation quality, but has no effect on code quality, requirement problems, or personnel problems. The increase in documentation imposed by SPI has no perceivable positive impacts for software developers. This conclusion is similar to the study by Kuilboer and Ashrafi (S4) presented previously : The factors that are important to developers are not improved by early stage SPI initiatives, which are concerned with process documentation and control.

There is no doubt that SPI imposes a documentation overhead on developers. Leung and Yuen (S16) evaluate this overhead at 35

> We found a perception of increased overhead and increased demand on the developers' time. This is somewhat a natural reaction from software engineers [...], who would be accustomed to a less time-consuming but less thorough analysis of the required functionality.

Similarly, Subramanian et al. (S17) note that simpler documentation, as when the "Keep It Simple and Straightforward" principle (KISS) is applied to management, leads to earlier completion times. They note no immediate effect on quality, although maintenance impacts were not considered. Basri and O'Connor (S12), along with Coleman and O'Connor (S18) show that small organizations are openly hostile to this documentation overhead. Coleman and O'Connor report that large enterprises view documentation as a *"necessary evil"*, and *"the smaller the company the greater the hostility towards Documentation"*. According to statements reported by Basri and O'Connor :

> Too much documentation and you need somebody to just work on the software process alone. Because our developers are busy with coding, documentation is the last thing they do.

> If you want to get done quickly then what you need is focusing to the output not the process.

Developers in the study by Basri and O'Connor in very small enterprises consider that the qualities of a lightweight process are the following :

— *Minimum documentation requirement*,
— *Easy to administer*,
— *Less change from current development process*,
— *Minimum overhead in terms of cost and resources.*

There is, therefore, a very strong aversion to extensive documentation in smaller enterprises, and thus toward SPI at the initial levels.

Basili et al. (S2), from their experience at the NASA Software Engineering Laboratory, argue that the need for data must be balanced with the capacities of the developers :

> You must compromise in asking only as much information as is feasible to obtain.

This means that, in general, documentation needs must be tailored to the software teams, especially in the early stages of SPI, and especially in smaller settings.

F.6.4 Knowledge Sharing

The environment of the developer is defined by his team, as seen in Table VII. Proper communication and knowledge sharing within the team is therefore critical for a good work environment. Evidence from the selected studies shows that documentation problems can be mitigated with better communications. As Coleman and O'Connor (S18) report :

> Many companies substituted verbal Communication for Documentation, and co-located their development teams in an effort to reduce process cost. A benefit of doing this was an increase in the sharing of Tacit Knowledge.

Damian et al. (S1) report that documentation shortcomings in requirements can be mitigated by asking developers to join requirements elicitation meetings. This enables developers to share the rationale behind requirements directly with the stakeholders, rather than documenting them explicitly.

In a similar vein, Ramasubbu and Balan (S19) report that increased agility in a formally structured software team increases the *"opportunities to help each other and conduct community-based learning programs"*. This approach has led to improved performance for the more agile teams, because of increased knowledge sharing.

Hyde and Wilson (S20) report that one intangible benefit of SPI initiatives is an improvement in team communications. They cite a study by Herbsleb et al. [11], which concludes that *"communication, both between departments within an engineering centre and between centres, has improved"*. Hyde and Wilson also report that the standardization of information across the company also makes the developers more mobile :

> It is easier for software professionals to move across projects without any significant learning curve.

Baddoo et al. (S14) report that one trait of good software developers is that they *"share knowledge within the team"*. According to the developers in their study, knowledge sharing is as important as documenting their work.

Table F.6 Evidence for Documentation.

Studies	Evidence
(S13)	Documenting requirements is seen as a high value practice by developers and managers alike.
(S14)	Developers expect their coworkers to fully document their work, but are not as thorough themselves.
(S15)	High-severity maintenance problems are mostly related to documentation.
(S4, S15)	SPI improves process documentation, but with no perceived value to the developers.
(S1, S16, S17)	SPI forces a documentation overhead on developers. Simpler documentation improves completion time.
(S12, S18)	Small organizations view process models as oriented too much toward documentation bureaucracy.
(S2)	Ask for only as much documentation as it is feasible to obtain.

Table F.7 Evidence for Knowledge Sharing.

Studies	Evidence
(S1, S18, S19)	Documentation can be replaced with better communication within co-located teams.
(S20)	SPI improves communications within the team. It also increases the mobility of developers among projects.
(S14)	Good developers share knowledge within the team.

F.6.5 Effort and Morale

Process maturity, achieved through continuous SPI initiatives, leads to better mastery of a company's development processes. This mastery translates into an increased stability in the conduct of development processes, which in turn reduces the need for heroic efforts by personnel, as seen in Table VIII. As Agrawal and Chari (S21) see it :

> The adoption of highly mature software development processes during software development reduced the significance of many factors such as personnel capability.

Kandt (S22) reports similar results, with a reduction in uncounted overtime by at least fifteen percent. Hyde and Wilson (S20) cite other authors who report a reduction in the required heroic efforts :

> Software engineers are spending fewer late nights and weekends on the job, and there is less turnover. [12].

> [With] improvement in work life and decreases in crises, overtime hours were reduced and turnover among the professional software personnel has remained relatively low. [11].

In this last example, the reduction in the demand for heroic efforts on the part of developers resulted in an improvement in work life quality, and of the team's morale. Hyde and Wilson (S20) report similar results in his SPI experiment :

> Overall, respondents agreed that the quality of [their] work life/working conditions had improved. They believed they were facing fewer crises/problems and that the increased stability in their working environment was giving them more confidence.

Dangle et al. (S9) report a similar increase in morale when an SPI initiative improved responsibility sharing at its target company. A frustrating process bottleneck was streamlined. They write :

> Employee morale is higher. Some have been able to get more responsibility, while others, such as the chief programmer, were able to offload some of theirs.

However, in a less mature environment where management does not listen to developers, some problems arise. Hall et al. (S7) report that :

> The following quote from a software developer in one company sums up the frustration that many of our study participants voiced to us : ...software is being released with known faults as time usually takes precedence over quality.

Without the resources to ensure quality, the developers become frustrated, and morale dips. Niazi et al. (S23) report that lack of resources is the greatest obstacle to SPI success, with eleven out of eighteen developers describing it as a threat. In short, SPI initiatives without dedicated resources are doomed to fail, as developers are asked to juggle with increased documentation without the proper support. Unfortunately, the beneficial reduction in effort is not immediate, while the costs of the SPI initiatives are [9].

F.6.6 Skills

As seen in Table IX, two kinds of software development skills are related to SPI : technical skills and process skills. Technical skills are related to the task at hand. Process skills reflect knowledge of the process and on how to use its capabilities. Success with software projects depends on technical skills, but success with SPI depends on process skills.

Very similar conclusions are drawn in the selected studies in terms of skills : A good set of process skills is essential for SPI success. Dangle et al. (S9) report as follows :

> Assess staff process improvement experience. The individual assigned to support the process improvement effort didn't have much process improvement experience. We had to do much more training and hand-holding for the process improvement staff than we expected.

In the domain of reuse-oriented SPI, Morisio et al. (S24) report :

> We find that the programming language, experience [technical skills], rewards, repository and reuse measurement are not decisive factors, while reuse education is.

As these two studies report, process awareness education is essential for SPI success. Studies

Table F.8 Evidence for Effort and Morale.

Studies	Evidence
(S20, S21, S22)	High maturity reduces the need for personnel heroics, by reducing crises.
(S20)	The reduction in crises improves stability, job satisfaction, and morale.
(S9)	SPI leads to better responsibility sharing, which leads to better morale.
(S7)	Releasing software with known faults is bad for morale.
(S23)	Lack of resources is seen by the developers as the greatest barrier to SPI success.

by Niazi et al. (S23), Rainer and Hall (S25), and Sommerville and Ransom (S10) also stress the importance of experience and education, although the difference between process skills and technical skills is unclear in their case. They all mention that inexperienced staff is an obstacle to success, although it is not clear if they are talking about technical inexperience or inexperience with SPI.

By contrast, Agrawal and Chari (S21) demonstrate that, in highly mature companies, staff experience has no impact on effort. Process-aware developers within a mature process are able to find solutions to skill shortcomings and to prevent variations in effort.

Subramanian et al. (S17) show that technical training does not, in fact, impact completion time. They note that trained staff produces higher quality artifacts, however.

Wilkie et al. (S26) conclude that technical skills are essential in small settings :

> Small software companies tend to focus on product quality assurance rather than process quality assurance – relying more heavily on individual developer competence as opposed to process.

F.6.7 Requirements and Clients

Requirements documents are some of the main input materials provided to software developers. Requirements are regularly studied in the selected studies, as seen in Table X. Niazi and Babar (S13) report that both developers and managers see the requirement practices of the CMMI as the most important for software development :

> These findings indicate that our respondents are aware of the importance of "requirements management" process area as most of the practices in these process

Table F.9 Evidence for Skills.

Studies	Evidence
(S9, S10, S23, S24, S25)	Successful SPI requires skills with processes. SPI education is critical for success.
(S21)	The skills of personnel have no impact in highly mature companies.
(S17)	Training does not affect completion time, but it does affect quality.
(S26)	Small companies rely on individual competencies, with respect to product quality, rather than process quality.

> areas were perceived as high value practices by more than 50% of our respondents.

Hall et al. (S7) bear this out with their findings on the impact of bad requirements on maintenance :

> Developers report major problems in companies with faults delivered to customers coupled with major problems with low user satisfaction levels. Indeed we found that poor requirements capture contributes many problems to maintenance.

This shows that poor quality requirements have a direct impact on developers, through an increased maintenance overhead. The good news is that SPI can improve the quality of the requirements, as shown by Kandt (S22) :

> I attribute the reduced number of defects found in the code partially to the low number of defects introduced in requirements. Only 1.8 percent of requirements had findings written against them, which is an order of magnitude fewer than other projects.

Damian et al. (S1) found similar results when developers were included in the requirement development process.

F.7 Related Works

A number of systematic reviews have already been conducted in the SPI domain. However, none of them involved direct observation of the developers. The objective of this systematic review is to fill this gap.

The review by Staples and Niazi [13] looks at organizations and their motivations in adopting SPI. They found that the primary motivation of organizations is the potential for business benefits. This conclusion is in line with those of the selected studies in this review, although it is a nuanced one. We find that business benefits should indeed be linked to SPI initiatives prior to their introduction, but that this is not always the case. Staples and Niazi mention, however,

Table F.10 Evidence for Requirements and Clients.

Studies	Evidence
(S13)	Developers and managers alike view requirements as having a significant value.
(S7)	The main maintenance problems are linked to requirements.
(S1, S22)	SPI improves the quality of requirements

that *"people-related reasons [are] rarely given by organizations"*. Developers are therefore rarely considered by organizations when discussing SPI, which might explain the undue focus on technical approaches.

The review by Sulayman and Mendes [2] looks at SPI in Web development companies. These authors show that *"development team satisfaction"* is an important factor for SPI success. Impacts on developers are therefore important, at least in the Web development sector.

The review by Beecham et al. [14] looks at the factors that motivate and demotivate developers. Their review does not consider SPI, but is nevertheless superficially similar to ours. Their main conclusion is the following :

> There is no clear understanding of the Software Engineers' job, what motivates Software Engineers, how they are motivated, or the outcome and benefits of motivating Software Engineers.

Our review shows a number of factors which could potentially explain the motivational and demotivational elements identified by Beecham et al. By studying generic impacts, rather than limiting ourselves to motivation, we found more potential impact factors. However, our review focuses solely on SPI, while theirs observes motivation in general.

Previous reviews had mainly looked at motivational factors, which is only a minor subset of all the impacts of SPI. By concentrating on generic impacts in this review, we were able to find more evidence, and to pinpoint some new factors with potential indirect impacts on developers' motivations.

F.8 Summary

In this systematic review, twenty-six studies were selected from the field of SPI research. Evidence from these studies was then extracted and organized using the grounded theory approach. This resulted in seven factors having positive and negative impacts on software developers.

The impact of the Motivation and Resistance factor is that developers worry about the overhead of the SPI initiative on their work and its effect on their performance. Justifying SPI is difficult, because early stage improvements are not oriented toward developers' needs, but toward those of management. Without rapidly visible results, there is certain to be resistance to SPI from developers.

The impact of the Technical Focus factor is that management is mostly concerned with technical approaches to SPI, even though research shows the problem to be mostly organizational.

There is also a discrepancy between SPI approaches and business goals, which undermines management support. The technical approaches proposed are not linked to potential business benefits, which makes it difficult to obtain sustained commitment from senior management.

The impact of the Documentation factor is that SPI initiatives increase the developers' overhead. Early stage SPI is mostly concerned with process documentation, which has little perceived value to the developers. However, requirement documentation is viewed positively by the developers, and is critically important in maintenance activities. The extent of the documentation should therefore be calibrated to the needs of the project, and to the capacity of the developers to handle it.

The impact of the Knowledge Sharing factor is that some documentation can be substituted with informal communications. The good news is that SPI improves team communications, and thus knowledge sharing.

The impact of the Effort and Morale factor is that SPI improves morale by reducing the need for heroic efforts by team members. Process maturity reduces crises and improves stability, which is also positive for morale.

The impact of the Skill factor is that SPI success requires process skills, not just technical skills. Developers must be trained to understand and use the process. Technical skills are important in low-maturity companies, but shortcomings can be mitigated with a high-maturity process. Technical skills only affect quality, and not time-to-market.

The impact of the Requirements and Clients factor is that requirements are seen as critically important by developers. Poor requirements lead to maintenance problems. Thankfully, SPI as a whole improves the quality of the requirements.

In summary, SPI can have positive impacts on developers, such as an increase in morale through a reduction in the number of crises occurring during development. But it can also have negative impacts, such as a need to produce documentation that is of no perceived value to the developers. By shedding light on the impacts of SPI initiatives on software developers, this review should help future SPI practices to succeed by identifying obstacles to success, but also what promotes success, from the developers' standpoint. Future research should investigate the impacts identified for SPI initiatives, and ensure that future practices obtain the continuous support of the software developers.

F.9 Acknowledgment

This work was made possible through the support of the Natural Sciences and Engineering Research Council of Canada, through grant 361161 and grant A0141.

Preliminary testing of the methodology was performed by four reviewers. They are, in alphabetical order, André Breton, Frédéric Doll, Hamza Kechih, and Walid Ben Jemia.

F.10 References

[1] T. Dyba, "An Instrument for Measuring the Key Factors of Success in Software Process Improvement," Empirical Software Engineering, vol. 5, pp. 357-390, 2000.

[2] M. Sulayman and E. Mendes, "A Systematic Literature Review of Software Process Improvement in Small and Medium Web Companies," Advances in Software Engineering, vol. 59, pp. 1-8, 2009.

[3] B. Kitchenham, "Procedures for performing systematic reviews," Keele University and NICTA, Technical Report, 2004.

[4] J. Biolchini, P. Gomes Mian, A. Candida Cruz Natali, and G. Horta Travassos, "Systematic Review in Software Engineering," PESC / COPPE / UFRJ, Rio, Technical Report, 2005.

[5] T. Dybå and T. Dingsøyr, "Empirical studies of agile software development : A systematic review," Information and Software Technology, vol. 50, nos. 9-10, pp. 833-859, 2008.

[6] A. L. Strauss, Qualitative analysis for social scientists, Cambridge Univsersity Press, 1987.

[7] B. G. Glaser, Basics of grounded theory analysis, Sociology Press, 1992.

[8] K. Ishikawa, Introduction to quality control : 3A Corporation, 1990.

[9] M. Lavallée and P. N. Robillard, "Do software process improvements lead to ISO 9126 architectural quality factor improvement," Proc. International Workshop on Software Quality (WoSQ'11), ACM, 2011, pp. 11-17, doi : 10.1145/2024587.2024592.

[10] K. Beck, M. Beedle, A. van Bennekum, A. Cockburn, W. Cunningham et al., "Principles behind the Agile Manifesto," Available at : http ://agilemanifesto.org/principles.html, 2001.

[11] J. Herbsleb, A. Carleton, J. Rozum, J. Siegel, and D. Zubrow, "Benefits of CMM-Based Software Process Improvement : Initial Results," Software Engineering Institute, Carnegie Mellon University, 1994.

[12] R. Dion, "Process improvement and the corporate balance sheet," IEEE Software, vol. 10, no. 4, pp. 28-35, 1993.

[13] M. Staples and M. Niazi, "Systematic review of organizational motivations for adopting CMM-based SPI," Information and Software Technology, vol. 50, nos. 7-8, pp. 605-620, 2008.

[14] S. Beecham, N. Baddoo, T. Hall, H. Robinson, and H. Sharp, "Motivation in Software Engineering : A systematic literature review," Information and Software Technology, vol. 50, nos. 9-10, pp. 860-878, 2008.

F.11 Appendix A. The Selected Studies

(S1) D. Damian, D. Zowghi, L. Vaidyanathasamy, and Y. Pal, "An Industrial Case Study of Immediate Benefits of Requirements Engineering Process Improvement at the Australian Center for Unisys Software," Empirical Software Engineering, vol. 9, nos. 1/2, pp. 45-75, 2004.

(S2) V. R. Basili, F. E. McGarry, R. Pajerski, and M. V. Zelkowitz, "Lessons learned from 25 years of process improvement : the rise and fall of the NASA software engineering laboratory," Proc. International Conference on Software Engineering (ICSE 2002), IEEE Computer Society, May 2002, pp. 69-72, doi :10.1145/581339.581351.

(S3) J. Batista and A. Dias de Figueiredo, "SPI in a very small team : a case with CMM," Software Process : Improvement & Practice, vol. 5, no. 4, pp. 243-250, 2000.

(S4) J. P. Kuilboer and N. Ashrafi, "Software process and product improvement : an empirical assessment," Information and Software Technology, vol. 42, no. 1, pp. 27-34, 2000.

(S5) G. C. Green, A. R. Hevner, and R. Webb Collins, "The impacts of quality and productivity perceptions on the use of software process improvement innovations," Information and Software Technology, vol. 47, no. 8, pp. 543-553, 2005.

(S6) K. Daily and D. Dresner, "Towards software excellence—informal self-assessment for software developers," Software Process : Improvement & Practice, vol. 8, no. 3, pp. 157-168, 2003.

(S7) T. Hall, A. Rainer, N. Baddoo, and S. Beecham, "An empirical study of maintenance issues within process improvement programmes in the software industry," Proc. IEEE International Conference on Software Maintenance (ICSM 2001), IEEE Computer Society, Nov. 2001, pp. 422-430, doi : 10.1109/ICSM.2001.972755.

(S8) M. Blanco, P. Gutierrez, and G. Satriani, "SPI patterns : learning from experience," IEEE Software, vol. 18, no. 3, pp. 28-35, 2001.

(S9) K. C. Dangle, P. Larsen, M. Shaw, and M. V. Zelkowitz, "Software process improvement in small organizations : a case study," IEEE Software, vol. 22, no. 6, pp. 68-75, 2005.

(S10) I. Sommerville and J. Ransom, "An empirical study of industrial requirements engineering process assessment and improvement," ACM Trans. Softw. Eng. Methodol., vol. 14, no. 1, pp. 85-117, 2005.

(S11) F. Cattaneo, A. Fuggetta, and D. Sciuto, "Pursuing coherence in software process assessment and improvement," Software Process : Improvement & Practice, vol. 6, no. 1, pp. 3-22, 2001.

(S12) S. Basri and R. V. O'Connor, "Understanding the Perception of Very Small Software Companies towards the Adoption of Process Standards," in Systems, Software and Services Process Improvement, Communications in Computer and Information Science, vol. 99, A. Riel, R. O'Connor, S. Tichkiewitch, and R. Messnarz, Éds. Berlin : Berlin Heidelberg, 2010, pp. 153-164.

(S13) M. Niazi and M. A. Babar, "Identifying high perceived value practices of CMMI level 2 : An empirical study," Information and Software Technology, vol. 51, no. 8, pp. 1231-1243, 2009.

(S14) N. Baddoo, T. Hall, and D. Jagielska, "Software developer motivation in a high maturity company : a case study," Software Process : Improvement & Practice, vol. 11, no. 3, pp. 219-228, 2006.

(S15) J.-C. Chen and S.-J. Huang, "An empirical analysis of the impact of software development problem factors on software maintainability," Journal of Systems and Software, vol. 82, no. 6, pp. 981-992, 2009.

(S16) H. K. N. Leung and T. C. F. Yuen, "A process framework for small projects," Software Process : Improvement & Practice, vol. 6, no. 2, pp. 67-83, 2001.

(S17) G. H. Subramanian, J. J. Jiang, and G. Klein, "Software quality and IS project performance improvements from software development process maturity and IS implementation strategies," Journal of Systems and Software, vol. 80, no. 4, pp. 616-627, 2007.

(S18) G. Coleman and R. O'Connor, "Software Process in Practice : A Grounded Theory of the Irish Software Industry," in Software Process Improvement, Lecture Notes in Computer Science, vol. 4257, I. Richardson, P. Runeson, and R. Messnarz, Éds. Berlin : Springer Berlin / Heidelberg, 2006, pp. 28-39.

(S19) N. Ramasubbu and R. K. Balan, "The impact of process choice in high maturity environments : An empirical analysis," Proc. International Conference on Software Engineering (ICSE 2009), IEEE Computer Society May 2009, pp. 529-539, doi :10.1109/ICSE.2009.5070551.

(S20) K. Hyde and D. Wilson, "Intangible benefits of CMM-based software process improvement," Software Process : Improvement & Practice, vol. 9, no. 4, pp. 217-228, 2004.

(S21) M. Agrawal and K. Chari, "Software Effort, Quality, and Cycle Time : A Study of CMM Level 5 Projects," Software Engineering, IEEE Transactions on, vol. 33, no. 3, pp. 145-156, 2007.

(S22) R. K. Kandt, "Experiences in Improving Flight Software Development Processes," Software, IEEE, vol. 26, no. 3, pp. 58-64, 2009.

(S23) M. Niazi, M. A. Babar, and J. M. Verner, "Software Process Improvement barriers : A cross-cultural comparison," Information and Software Technology, vol. 52, no. 11, pp. 1204-1216, 2010.

(S24) M. Morisio, M. Ezran, and C. Tully, "Success and failure factors in software reuse," Software Engineering, IEEE Transactions on, vol. 28, no. 4, pp. 340-357, 2002.

(S25) A. Rainer and T. Hall, "A quantitative and qualitative analysis of factors affecting software processes," Journal of Systems and Software, vol. 66, no. 1, pp. 7-21, 2003.

(S26) F. G. Wilkie, D. McFall, and F. McCaffery, "An evaluation of CMMI process areas for small- to medium-sized software development organisations," Software Process : Improvement & Practice, vol. 10, no. 2, pp. 189-201, 2005.

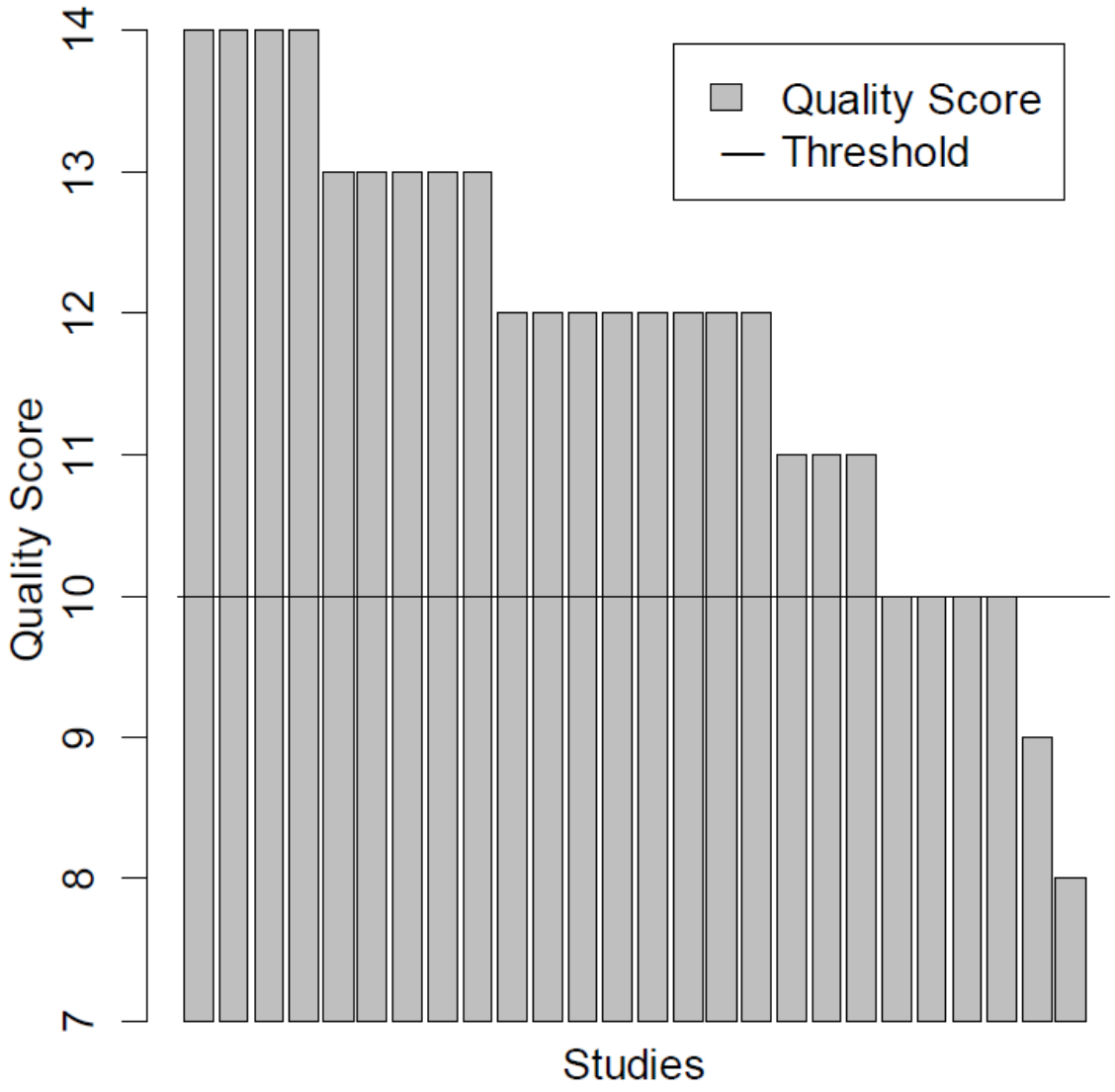

Figure F.1 Quality Results

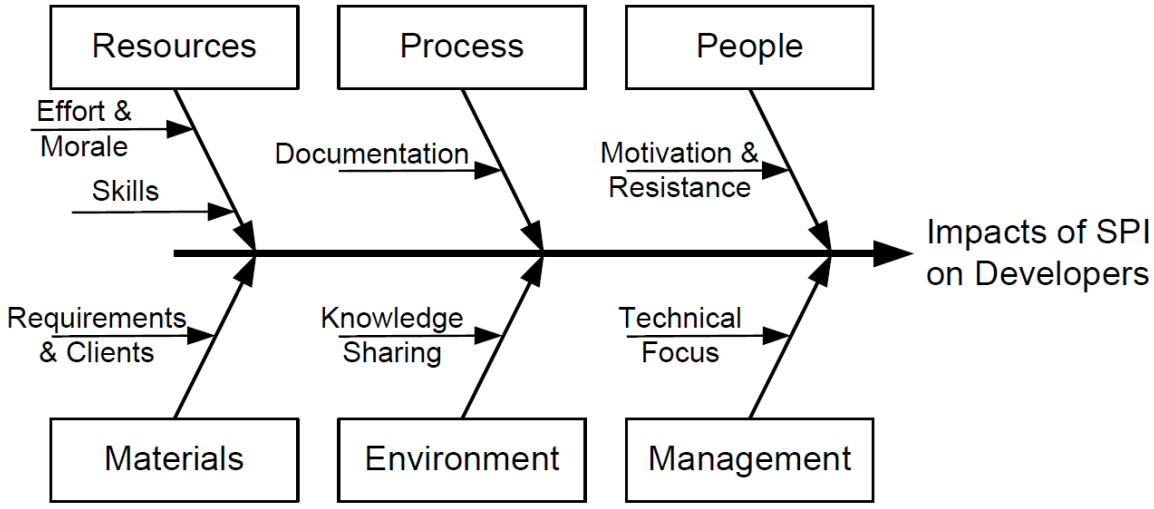

Figure F.2 Ishikawa diagram for the seven categories of evidence.

ANNEXE G ARTICLE 6 : CAUSES OF PREMATURE AGING DURING SOFTWARE DEVELOPMENT : AN OBSERVATIONAL STUDY

Type de publication	Référence complète
Workshop	*Causes of Premature Aging During Software Development : An Observational Study*, **Mathieu Lavallée** et Pierre N. Robillard, International Workshop on the Principles of Software Evolution and ERCIM Workshop on Software Evolution (IWPSE-EVOL), 2011, doi :10.1145/2024445.2024458

G.1 Abstract

Much work has been done on the subject of what happens to software architecture during maintenance activities. There seems to be a consensus that it degrades during the evolution of the software. More recent work shows that this degradation occurs even during development activities : design decisions are either adjusted or forgotten. Some studies have looked into the causes of this degradation, but these have mostly done so at a very high level. This study examines three projects at code level. Three architectural pre-implementation designs are compared with their post-implementation design counterparts, with special attention paid to the causes of the changes. We found many negative changes causing anti-patterns, at the package, class, and method levels. After analysis of the code, we were able to find the specific reasons for the poor design decisions. Although the underlying causes are varied, they can be grouped into three basic categories : knowledge problems, artifact problems, and management problems. This categorization shows that anti-pattern causes are varied and are not all due to the developers. The main conclusion is that promoting awareness of anti-patterns to developers is insufficient to prevent them since some of the causes escape their grasp.

G.2 Introduction

It is generally admitted that the architecture of a software component degrades during maintenance activities [1-4]. Software maintenance introduces new requirements and technological advances [5] which contort existing design structures in unplanned ways. This in turn creates

structures that are even more difficult to understand and maintain, which contributes to the increased complexity of the source code. Unfortunately, research also shows that this vicious circle of increased complexity is almost inevitable [5-7].

The post-implementation design (post-ID) derived from the source code reflects this increased complexity, which is often identified through common design pitfalls known as "smells" [8] or "anti-patterns"[9]. From the software aging perspective, the intensity of these smells will depend on the quality of the pre-implementation design (pre-ID). A good pre-ID should be flexible enough to minimize the impact of future changes [6, 10], thereby minimizing the adjustments required in the post-ID, which minimizes the increase in complexity.

G.2.1 Design Erosion

The increased complexity phenomenon has been referred to in a number of ways. Parnas [6] calls it "software aging" and attributes the condition to the introduction of new requirements. Perry and

Wolf [11] present two definitions of it : "architectural erosion" and "architectural drift". Architectural erosion refers to changes that are in conflict with previous design decisions, while architectural drift refers to design decisions forgotten or unused by the development team. Van Gurp and Bosch [5] call it "design erosion", because its effects are not limited to the architecture, but can influence all the software artifacts. We prefer this latter definition, although our analysis shows signs of both erosion and drift.

Design erosion is usually associated with maintenance. New research seems to show that even the initial development phase is not immune to transformed design decisions [4, 12, 13]. This indicates that premature software aging is possible if poor design decisions are made during development. There is a need, therefore, to prevent these bad decisions at the development stage, since a good post-ID will be resistant to change and will not age the software as quickly [6]. But, to prevent bad decisions, we must first determine their causes, which is the purpose of this article.

G.2.2 Evolution, Erosion and Development

The study of software evolution focuses on the mitigation and control of changes during the lifetime of software systems [3].While erosion is typically concerned with post-development deterioration, evolution is concerned with all changes, whether detrimental or beneficial, throughout the lifecycle of the software.

This study focuses on development activities, and is thus limited on software evolution during

development. However, results show that change mitigation and control is as important during development as it is during maintenance. This study shows that an important number of problems were introduced because of poorly managed changes made between pre-ID and post-ID, which falls square into the field of software evolution.

G.2.3 Code Smells

Many studies have performed high-level analysis of the design of various projects, but this study takes a lower-level approach through code reviews. Problems at the code level have often been identified through a metrics approach, referred to as "code smells". Smells are common pitfalls found in code and are indicative of design problems. Common classes of smells include [8] :

— Blob : Also known as a "god class", a blob is a large class doing most of the control and decision making in a software project. It is characterized by low cohesion and the presence of multiple unrelated concerns.
— Spaghetti Code : This class is a reflection of procedural programming in an object-oriented environment. It contains many long methods which process data through global variables and forgo inheritance and polymorphism.

However, metrics-based approaches do not capture all the available information. While we retain the concepts of the smell domain, our study focuses on a qualitative approach. Smell metrics are used and are provided when pertinent, but a qualitative analysis is performed to confirm or refute the presence of a problem. This approach proved fruitful, as subtle errors not caught by metrics caused far-reaching problems (see, for example, problem 8-4 in section 3.2). Manual context evaluation has been deemed important by other authors as well [8, 13, 14].

G.2.4 Related Works

This study complements Cook and Lehman's description of software evolution [3]. Chaki's approach [15] defines the "how" of software evolution through planned processes, Cook and Lehman try to explain the "what". Our study focuses instead on the "why", that is, the causes behind the problems observed. Specifically, we study Paradigm-type systems, where a pre-ID serves as a reference for the construction phase, but is not strictly enforced.

In addition, this study parallels Eklund's classification of design decisions [16]. However, Eklund's work focuses more on the lifecycle of documented high level design decisions, while ours focuses on design decisions at the code level, from a comparison of pre-ID artifacts and a

reverse-engineering of the post-ID code. Eklund's work examines decisions at the management level, while ours examines decisions at the developer level.

Analysis at the code level is in line with problem correction, as described by Fowler in his Refactoring book [17]. We found however some problems more complex than those presented in Fowler's book and which could not have been found using purely automated means.

G.2.5 Focus of this Article

The objective of this article is to compile a list of causes behind negative discrepancies between pre-ID and post-ID. We follow the tenet that design erosion is inevitable, but we think that this erosion can be mitigated with careful design decisions. We hope that, by identifying and categorizing the causes behind harmful design decisions taken during development, we will help software engineers prevent premature aging. We present our identification and categorization results, obtained through a careful design and code analysis of three selected projects.

As an observational study, the causes identified herein remain limited to the context of the projects. However, the objective is not to provide an exhaustive list of categorized causes, but to demonstrate that some causes are beyond the reach of the developers. This observation is coherent with Deming's definition of quality, where quality is not the sole responsibility of the worker, but is also the responsibility of management [18].

Brown et al. reference on anti-patterns [9] has already compiled a list of potential "root causes" but remains at a very high level without a link to direct observations and specific anti-patterns. This observational study aims to fill this gap between high-level abstract causes and low-level observed problems.

G.3 Methodology

This study is based on three projects. The EPM6 and EPM8 projects are capstone projects, while the Dioscuri project is from Sourceforge. The EPM6 and EPM8 projects were realized by teams of five undergraduate students in their final semester. These students were specially selected for these projects, based on superior grades and teacher references. These two projects were realized with the same private partner company, and their objective was to fulfill genuine needs within the company. Both projects were considered a success, and the partner company declared both the pre-ID and the post-ID to be coherent with similar documentation at their company. The projects are still in use and maintained within the company. The pre-ID artifacts were reviewed and approved by a software professional at the partner company

before any implementation activity took place.

The Dioscuri project was initiated by the National Library of The Netherlands [1]. It was developed by a private company, Tessella Support Services. Once an initial version had been completed by the company (version 0.0.9), it was subsequently released on Sourceforge in February 2007. The project is still active (as of February 2011) and is used in a variety of settings, even though it is still graded as an alpha release (version 0.7.0). The choice of this project was based on the fact that it was initially developed by software professionals, and that much of its pre-ID documentation is available, most notably the Architectural Design Document (ADD).

Analysis is performed through a pre-ID review, followed by a post-ID review performed from the source code. Other artifacts were analyzed to confirm the problem classes, among them :

— Initial and final requirement analysis documents,
— Issue tracking software (Bugzilla),
— Tokens for actual work performed,
— Iteration plans.

These documents helped us find the portions of the code that caused the most problems. These portions were subsequently analyzed for problematic design decisions, common code smells, and recognized anti-patterns.

G.3.1 Problem Identification Methodology

The problem identification methodology used a qualitative approach based on manual artifact and code reviews. The aim of the reviews was to identify significant problematic anti-patterns within the projects. Table 1 describes the target anti-patterns of the manual code review. The artifact review was limited to the anti-patterns relevant to the artifact content. For example, concern mixture can be identified in class diagrams, while dead code cannot. The definitions of the anti-patterns are based on Mäntylä's Master Thesis [19] and Microsoft's Application Architecture Guide [20]. They are respectively coded as MMT and AAG in Table 1.

This methodology was validated in a single-blind study done on the EPM8 project. Six graduate students split into three teams performed the same work as the original reviewer. The three teams were given the project and the methodology, with no indications on what could be found. Little variation was found with the minor problems, and all the major problems found were identical. All the EPM8 problems presented in this study were also found by the three validating teams.

1. http ://dioscuri.sourceforge.net/

Table G.1 Anti-patterns targeted by the reviews.

Anti-pattern	Description
Lack of Polymorphism	Presence of switch statements and/or type properties (MMT, page 37).
Long Method	Complex method which could be broken up into steps (MMT, page 36).
Duplicated Code	Same snippet of code at multiple places (MMT, page 37).
Concern Mixture	Multiple concerns (aspects) in the same class (AAG, pages 11-13).
Dead Code	Never executed code or empty methods (MMT, page 37).
Lazy Class	Class with minimal particular functionality which defer to other classes for its inner working (MMT, page 37).
Data Class	Class with only data and no functionality (MMT, page 37).
Encapsulation Violation	Exposition of the inner working of a class to the rest of the project (MMT, page 37 and AAG, page 13).

G.4 Results

G.4.1 The EPM6 Project

The objective of the EPM6 project was to add a graphical user interface (GUI) to an application previously available only through a command line. The command line application is a code and documentation generation tool. The project reused a significant portion of its code from the previous command line version. There are, however, enough new elements in the pre-ID and the post-ID to justify an analysis (27 new classes with 2300 executable statements). The project has been accepted as is by the partner company, and has been used in their daily operations.

Problem 6-1 – Implementation of the visitor pattern

The post-ID of the EPM6 project possesses a complex class with a very complex method that transforms the GUI model into code, documentation, and a functional Visual Studio project. The pre-ID, however, had proposed a different approach : that each generation would be handled by a different class (genCode, genDoc, genProject). The fusion of the three classes into one method created a behemoth containing 574 executable statements and 557 function calls, and featured an overabundance of code duplication.

The pre-ID approach to the three generations was close to a visitor pattern. The generating class would iterate over a list of elements and ask them to generate their code, documentation,

and project details themselves. The most probable cause of the loss of the visitor pattern is that certain details could not be generated by means of this pattern. Instead of trying to adjust the pattern to their specific needs, the developers preferred to scrap it entirely. This non visitor pattern approach forced the developers to add type attributes to their elements, which short-circuited all its polymorphic capabilities.

The destruction of the other two classes was probably opportunistic. The genCode class still exists in the post-ID, but is actually dead code : all the actual generation occurs in the genDoc class. It seems that the three pre-ID classes were created, but actual implementation only occurred in one of them.

There are three possible causes for this problem. One is incomprehension of the visitor pattern, the use of which requires a mastery of object-oriented concepts, especially polymorphism. It is possible that the developers' knowledge in this field was limited, and therefore they favored another approach with which they were more comfortable.

Another is that the pattern was known, but the developers did not know how to adapt it to their current needs. Certain parts of the generation could not be performed with a visitor pattern. There was a need, therefore, to modify the pattern to fit their requirements. The developers might have been unable or reluctant to modify a validated pattern, and preferred a home-brewed approach.

The third is a lack of project monitoring. At some point, a decision was made to forgo the visitor pattern. The negative impacts of this decision probably went unnoticed for some time, until it became too complex and too costly to correct. Either the decision's impact was not properly measured, or was not properly monitored.

Problem 6-2 – API functionalities

The P6 project used the Qt API for the graphical user interface (GUI). The API version used defines both toolbar and toolbox elements. In the Qt API, a toolbar contains buttons linked to other windows. A toolbox contains icons that can be dragged into the modeling scene. The pre-ID called for the construction of a toolbox, mistakenly referring to it as a toolbar. Unfortunately, the developers created their own toolbox, instead of reusing the Qt one. This adds a complex class to the system, one that does not feature the optimization of the Qt class.

The first possible cause of this problem would be in the pre-ID. Requirements mention the need for a toolbox, but mistakenly identify it as a toolbar. The developers would therefore analyze the Qt toolbar, and, when finding that it did not fit their needs, create their own toolbox which they named GraphicComponents.

A complementary cause to the first one is the fact that the Qt API had never been used before by the developers. An API is a complex software component and takes time and effort to master. It is probable, therefore, that some parts of the API were not explored by the developers, which might explain the presence of this redundant toolbox class.

Problem 6-3 – Implementation of an inheritance structure

The pre-ID defined an inheritance hierarchy for the graphical components used in the modeling. This inheritance hierarchy can be found in the post-ID. The use of inheritance enables polymorphic capabilities, but, unfortunately, the post-ID makes no use of polymorphism. The cause can probably be traced to the need to separate view concerns from control and model concerns. In the pre-ID inheritance model, these concerns are not separated. It seems that the necessity to separate the two concerns emerged during implementation. But this separation was not propagated to all the classes concerned, resulting in a bastardized design, as shown in Figure 1.

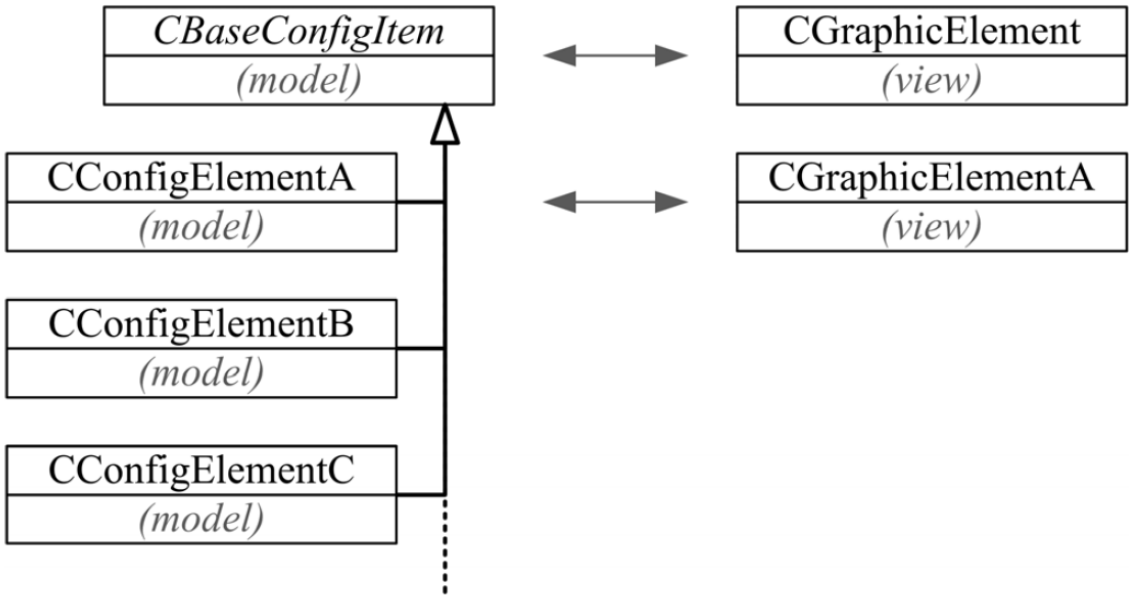

Figure G.1 Post-ID inheritance model for the graphic components.

The base abstract class CBaseConfigItem was appropriately split, sending its view concern to the CGraphicElement class. But this split was limited to the base class and one derived class : the view code for the other classes is all shoved into the CGraphicElement class. This incorrect split forces each element to define its type, so as to be appropriately managed by the view class. This in turn explains why polymorphism is completely absent from the post-ID, even though it was accounted for in the pre-ID.

The most probable cause is an inappropriately propagated design decision. The reason behind this design decision is not clear, however : Did the developers evaluate the impact of this change on the design ? Was this decision consensual, or the work of a single developer ? Improper planning and improper decision making could be at the root of this problem.

Another possible cause is the inability to split an inheritance hierarchy into control and view elements. Creating a structure that is coherent for both concerns, while keeping polymorphism possible, is not a trivial task, especially with many classes involved (nine, in this case). A lack of experience with object-oriented structures could have played a role in the decision to eliminate the whole hierarchy altogether.

Problem 6-4 – Quality of the pre-ID

The P6 project also outlines ambiguity errors in the pre-ID artifacts. In the pre-ID, two classes were shown with an implicit view/control separation. This separation was not explicit, however, and probably went unnoticed by the partner company's review. The pre-ID only contained interface elements, and the classes were shown interacting with the control classes. However, nowhere in the pre-ID was it made explicit that the classes CGraphicDesign and hierarchyDialog were to be specific view classes.

Unfortunately, the developers did not respect this implicit separation, and placed both the view code and the control code in the same classes, resulting in increased complexity in both cases.

The cause of this problem is the presence of ambiguity errors within the pre-ID. The pre-ID was made to follow the MVC (Model-View-Controller) architecture, and thus principles of MVC were expected to be found in the design artifacts. The reviewer saw MVC-appropriate classes, but the developers did not, because they were not labeled as such.

G.4.1.1 Impacts on maintenance

Poor decisions made during development can lead to maintenance challenges. Van Gurp notes that development itself can suffer from poor decisions made earlier, when their first case study project stalled between two development phases. They write that "the code was not as generic as it should have been" and that "some adaptive maintenance was needed to fix this" [12]. If this is the case during development, where developers know all the details of the code, what can happen when maintainers are seeing the code for the first time ? In our case, it is certain that this huge method (6-1) would have to be rewritten, and keeping the redundant toolbox up to date (6-2) would add extra overhead to the maintenance activities. In fact, it took us many hours to discover the reason for the aberrant inheritance structure

(6-3). Finally, the loss of the MVC architecture (6-4) means that the code is now harder to browse and understand.

G.4.2 The EPM8 Project

The objective of the EPM8 project was to limit the involvement of a human operator in the analysis of a video, with the use of optical recognition (OCR and OSR) through an open source library called Tesseract [2]. The project's software did not manage to remove all human interactions, but did achieve its objective of limiting the necessary post-analysis by human operators. The software has been maintained since its delivery, and is still in use at the partner company. Despite its success, a number of pre-ID to post-ID discrepancies have been identified :

Problem 8-1 – Implementation of the MVC architecture

The pre-ID defined a package structure in line with the MVC architecture. However, this structure is entirely absent from the post-ID. While a transformation into a new structure would have been acceptable, the post-ID is mostly unstructured : 29 of the 41 classes (70%) are in the generic root package, which accounts for about 80% of the project's executable statements. This lack of structure makes browsing the project's code a daunting task.

One possible cause of this forgotten design decision is the developers' lack of understanding of the MVC architecture. Classes identified as the Model in the pre-ID are mostly concerned with save/load features, while the real Model classes were mistakenly placed in the Controller pre-ID package. A lack of knowledge of what the MVC architecture implies could have led to this structural problem.

Another possible cause is that the MVC architecture design decision was forgotten during development. The decision was written down in the pre-ID, but was not read during the implementation phase. Therefore, it is not visible in the final structure.

A third possible cause is the inappropriateness of using the basic UML language to describe a structure in C#. The C# language does not have packages, but enables other structural elements, like namespaces, projects and directories. While the developers used the project elements minimally, they did not use the other elements at all. The absence of these elements from diagram artifacts means that these design decisions cannot be made when the diagram is sketched out. In this particular case, it is likely that these decisions were not made at all. This problem has already been identified by van Gurp's team [5] by the term expressiveness of representations.

2. http ://code.google.com/p/tesseract-ocr/

Problem 8-2 – Modularity violation

One of the project classes contains no less than 30% of all the executable statements. In smell lingo, such an anti-pattern would be named a "blob" or "god class" [8]. This class controls the main interface, and, while it is not present in the pre-ID, it would have belonged in the View package. Unfortunately, it also performs Control tasks, and even some Model tasks. While a certain amount of complexity is to be expected in a main interface class, this class has grown into a tangle of duplicated code which is not maintainable. A snippet of code used to manage an XML file appears in no fewer than six identical iterations throughout the class. This violation of modularity had been previously identified by van Gurp's team [12], who found that it had a severe impact on configuration management.

The cause behind this blob can be traced back to the vaporization of the MVC architecture. The post-ID contains two classes for the main interface, one assigned to View and the other assigned to Control. Unfortunately, the Control class is empty ; it is present in the project, but serves simply as a useless layer of indirection, since it does not possess any features of its own. Therefore, all the Control code is in the View class. Table 2 describes the two classes, where the features count all the functions and methods except object management ones (constructors, destructors, get/set methods). Note that the two attributes of Control-Class are the object itself (singleton pattern) and the InterfaceClass object it is supposed to encapsulate.

The elimination of the planned design architecture has an impact, even outside the package level. The absence of a clear line between architectural levels means that all of them were shoved into the same components, resulting in bloated blobs and god classes. While the MVC architecture is not a cure-all design solution, the absence of any architecture at all is certainly a problem.

A possible cause beyond the misunderstanding of the MVC architecture is the inability of

Table G.2 Bad split between View and Control classes.

Class	Executable Statements	Features
InterfaceClass	1032	23 attributes 6 public methods 47 private functions
ControlClass	18	2 attributes 0 public methods 0 private functions

the developers to implement this architecture in the project. Errors in the pre-ID seem to confirm that the developers do not know how to label View, Control, and Model elements appropriately. Lack of experience with the MVC architecture would explain why the separation was not implemented in the code.

Another possible cause is a communication problem within the team. At some point during development, a decision was made to have two separate classes, InterfaceClass and ControlClass. But either this decision was not communicated to the rest of the team, or it was not enforced during development, as the InterfaceClass was not in the pre-ID. Van Gurp shows a similar problem with a Data Access Layer (DAL) which was created but not enforced, resulting in a non-uniform system [12].

Problem 8-3 – Encapsulation violation

The developers understood the necessity of encapsulating complex code. The Tesseract library, being written in C, had to be encapsulated before being used by the C# system. The pre-ID took care to separate the complex C code from the C# interface, using a multi-layered approach. This approach was respected in principle in the post-ID, but not in practice. Figure 2 describes the situation.

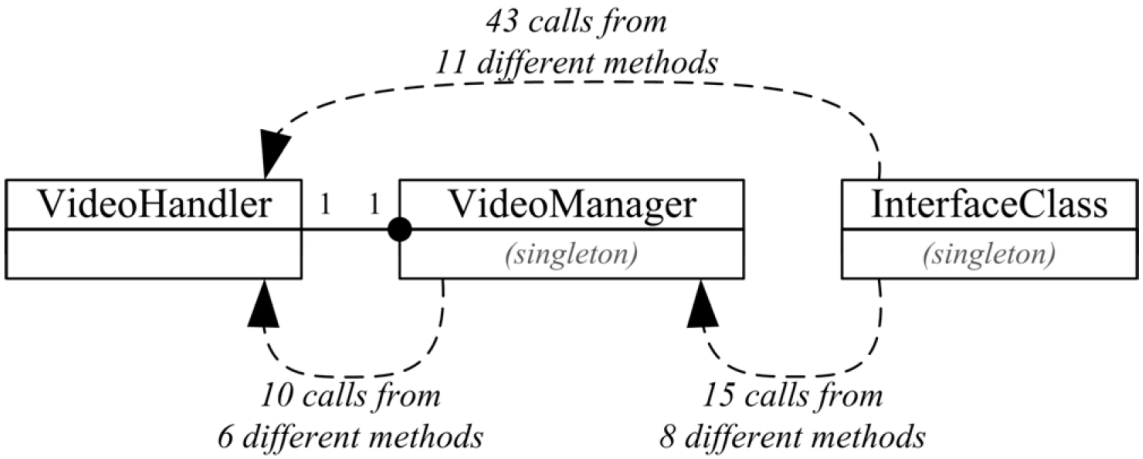

Figure G.2 Call bypass caused by a public data member.

The VideoManager class is a singleton class, and can therefore be used by any other class in the project (more on this in problem 8-4). It is mainly used by the main interface class of the project. The VideoManager class offers some middle level methods, while the encapsulated VideoHandler class does the low level video transformations. Unfortunately, at some point during the implementation phase, the VideoHandler encapsulated object was made public. This seems to have opened the floodgates, and the final post-ID can bear witness to this :

the vast majority of calls to VideoHandler are encapsulation violations. This results in a significant portion of the VideoManager class being found in the main interface class, further aggravating the blob problem found previously.

A possible cause of this problem is the inability to code the middle level functions. The developers knew the high level functionalities the InterfaceClass needed and the low level manipulation the VideoHandler class could deal with. Building the intermediate level methods in the encapsulating class is not a trivial task, however. For developers with time constraints, it is a superfluous step. This encapsulation only has value for the maintainers, since it offers a staged approach to the video manipulation that is easier to understand. Therefore, it is possible that the encapsulation was broken because the developers did not know how to implement it, and they did not want to spend resources on finding out how to do it.

This leads us to the most probable cause : time constraints. The developers acknowledge that they underestimated the time required to perform the middle level video manipulation in the VideoManager class. They admit that the solution implemented was rushed and is suboptimal. The problem, therefore, was essentially a project management one. Insufficient resources were assigned to the implementation of the class, resulting in basic object-oriented violations. In their final presentation, the developers themselves admit that this class will be a maintenance challenge and should be rewritten.

Problem 8-4 – Encapsulation and the "static" keyword

There is an abuse of static elements throughout the projects, but especially of the singleton design pattern [21]. While a critique of Gamma's singleton pattern is outside the scope of this article, we certainly see some of the less desirable traits in this project. The problem is that static elements in C# can be accessed by any object of the project. This can break the pre-ID's planned encapsulation by permitting the bypassing of calls. Figure 3 presents such an example, as seen in the EMP8 project.

The problem class is ImportedControl. The main interface class possesses the control cache, but unfortunately it is also possible to obtain a reference to this cache. This enables the SymbolicFieldVariable, a class deep within the data model, to bypass the entire project's structure, because the main interface class is a singleton and is thus accessible to anyone. This implies that any change to a singleton class can have an impact throughout the project, because any object could be calling it.

Worse, the ControlImporter class has defined a static method to build an ImportedControl object. The control cache, which should be the only class able to manipulate an imported control, uses this foreign static method to create its own objects ! This creates a three-way

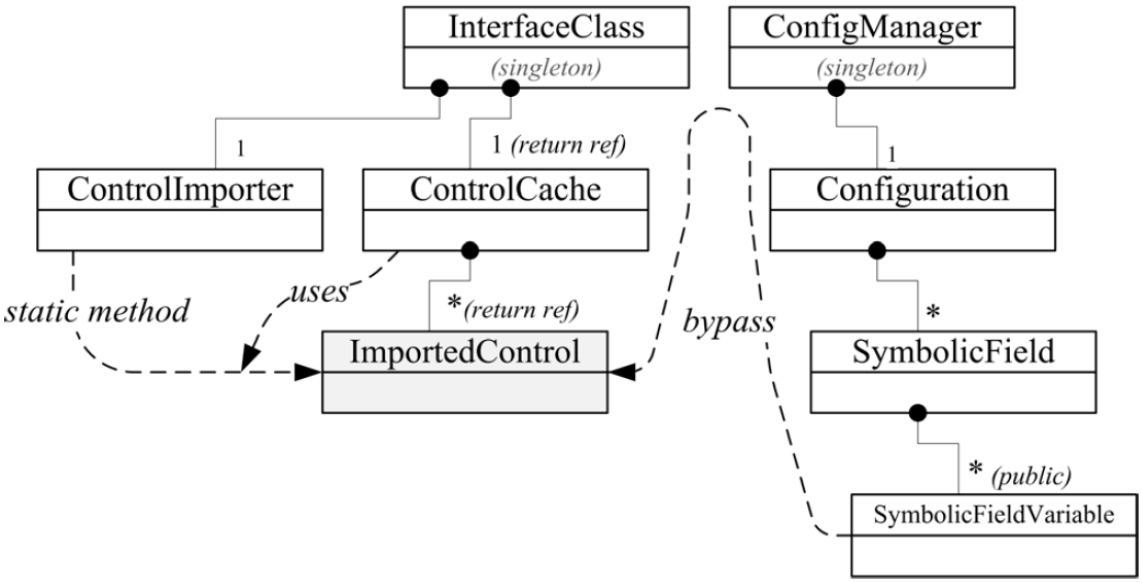

Figure G.3 Encapsulation breaking through static abuse.

mess which will be a maintenance nightmare.

The principle of good object-oriented design is that a class interface should be "easy to use correctly and hard to use incorrectly" [14]. This is not the case here, mostly through the abuse of static elements.

The main cause is the inability of developers to understand the impacts of the "static" keyword. The static keyword created global elements and significant coupling problems throughout the project, but which went unnoticed – and why should the developers notice it ? In most cases, the coupling is limited to closely related classes. But some aberrant cases do exist, and this design opens the door to more of these cases in the future.

Another complementary cause is a lack of planned review tasks. The developers enforced a coding standard, which demonstrated an interest in code quality. However, no resources were reserved for coding and design reviews. Even a superficial review would have shown the abnormal construction of the ImportedControl class. This project management omission is glaring, considering that design degradation is inevitable. Without degradation mitigation initiatives, whether through reviews or other means, the post-ID can age prematurely. This in turn will cause the software to be phased out more quickly than anticipated.

G.4.2.1 Impacts on maintenance

All four of the problems identified above will lead to maintenance challenges. The purpose of the MVC architecture is to isolate the concerns and make modification of one layer independent from that of the other. Its loss between pre-ID and post-ID means that maintainers will have a harder time when performing adjustments to the project (8-1). Similarly, modularity violations (8-2) indicate that a class is much more complex than planned, and thus harder to maintain. Encapsulation violations (8-3) are issues of procedural programming, which makes object-oriented code harder to understand. The abuse of the static keyword (8-4) adds unplanned coupling across the project, making future modification difficult.

Interview with one of the developers stresses that the decision to use MVC architecture (8-1) was unknown to him. He knew what MVC was, but was unaware that following this architecture was expected. This confirms that the decision to follow MVC was not appropriately communicated across the team. The pre-ID documents explicitly mention only twice the use of MVC architecture.

The developers also mention that they were aware of the introduced encapsulation violations (8-2, 8-3) but were forced to do so because of deadline pressures. The resources required to complete the task was poorly estimated. The developer concerned mentions that he was unaware of problems with the singleton pattern (8-4). Transformation of classes into singletons was done so the classes could be accessed anywhere within the code, in order to accommodate cross-cutting concerns. The developer did not realize that this approach would cause cohesion and coupling problems.

G.4.3 The Dioscuri Project

The objective of the Dioscuri project is to create a generic emulator for obsolete computers, implemented in the portable Java language. The rationale behind this project is to prevent the need to redigitalize documents recorded with older machines. A java emulator would be able to access the disk images of obsolete computers and thus ensure digital preservation, even when new technologies are introduced.

The project is currently able to emulate 8086, 186, 286, and 386 microprocessors with a wide variety of peripherals (floppy disks, CD-ROM drives, etc.). A number of operating system disk images are also available, most notably FreeDos and ELKS (an embedded Linux kernel). The project meets the end-user requirements, but has some maintenance challenges. Sourceforge download statistics seem to indicate that about 500 users are downloading the software after each update.

Problem D-1 – Architectural ambiguities

Dioscuri's pre-ID document is called the "Architectural Design Document", or ADD. However, it is limited to the presentation of the main structural objects and their interactions, and does not present an overall architecture of the future system. The Dioscuri team states that the objects "will be encapsulated in an emulation environment," which is not shown in the pre-ID. The overall architecture is thus deliberately left fuzzy and at the discretion of the implementers.

Unfortunately, this fuzzy pre-ID architecture created a very succinct package structure with an unclear purpose. For a project with 140k executable statements, this simple structure is inefficient. Packages represent the emulated physical modules of the system, with heavy coupling between each. The generic package contains the user interaction classes, which themselves contain high level GUI views and low level input transformations.

The configuration package mixes model, view, and control concerns. The ATA package, which defines parallel ATA ribboncable controllers, contains its own definitions, as well as unrelated objects simply using the controller, like CD-ROMs and disk images.

The main reason for this confusing structure is the ambiguous pre-ID. The lack of an overall architecture in the pre-ID indicates that no high level decision was made on the subject. Unfortunately, it seems that no high level thinking took place during the implementation either. Ambiguities in the pre-ID were not resolved in the post-ID, resulting in a suboptimal package structure.

Another possible reason is a lack of monitoring. By leaving the architecture open ended, the pre-implementation designers should have been aware of the risks behind that decision. Review meetings should have been planned to ensure that the design decision taken during implementation was the right one.

There is also a potential misunderstanding about the purpose of packages. In this project, packages are used simply as generic folders of classes. A package should be fairly self-contained, and coupling outside the package should be kept to a minimum. As Martin Fowler puts it "The biggest problem comes from uncontrolled coupling at the upper levels" [22]. Unregulated packages create many dependencies which are not evident to the maintainer [13]. The Dioscuri packages follow a hardware approach to group by physical module, and do not follow the software approach, which is to group by concern.

Problem D-2 – Misuse of the defined inheritance structure

The post-ID of the Dioscuri project defines an elaborate inheritance structure for its modules. However, the elements still possess type attributes, which are used in complex if-else-if struc-

tures. This implies that each change to a module must be propagated to all the appropriate if-else-if structures. For example, the attribute for floppy disk controllers, ModuleType.FDC, appears in 23 places in the project, spanning nine different classes in two packages.

This problem appears throughout the project. Instead of creating derived classes for specific hardware, type attributes are embedded in generic classes with elaborate control structures for each type of hardware implemented. This creates some serious Spaghetti issues.

The possible main cause is a lack of experience with object-oriented programming, most importantly with polymorphism. The approach used by the developers is reminiscent of procedural programming, and could stem from a lack of experience with object-oriented languages.

Problem D-3 – Overly complex method

The ATA.java class is one of the most complex classes in the project. Its objective is to emulate the working of the parallel ATA ribbon-cable controller. One of its methods, setPacketA0, is very complex. It contains 400 executable statements and 500 function calls, mostly included in a switch-case statement 770 lines long. This switch-case also contains internal switch-cases, further augmenting the complexity of the algorithm.

This huge switch-case is there to manage the many instructions of the ATA controller. However, the project already has classes to manage CPUs. Reusing these classes to manage the processor of an ATA controller would not have been very complex, as all the functionalities were already there.

The most probable cause is a lack of communication between team members. The ATA developers do not seem to be aware of the CPU structure implemented. This creates redundant code, which, in one case, the setPacketA0 method, will be very difficult to maintain.

Another potential problem is the difficulty of adapting the generic CPU to the limited instruction set of an ATA controller. It is possible that the option to reuse the CPU classes for the ATA controller was considered, but rejected. The developers might have been unable to adapt the existing CPU class to their needs.

G.4.3.1 Impacts on maintenance

The post-ID studied was based on the professional version, as released to the open source community, and marked as version 0.0.9. The project has been continuously improved by the community ever since, and has now reached version 0.7.0. This enables us to study how the problems found have matured between the release and the maintained versions. It shows that some problems were not corrected and remain maintenance challenges to this day.

The lack of an architectural structure of the project (D-1) remains in the current version. The main impact of this problem is that the maintainer cannot rely on the packages to determine where a modification must be made. Any modification is likely to affect multiple packages, due to heavy coupling and inter-package dependencies.

The inheritance problem (D-2) was seemingly discovered between version 0.0.9 and version 0.7.0, as the culprit attribute was eliminated. Unfortunately, it was replaced by a similar attribute in the root abstract class. This offers the advantage of centralizing the initialization of the type variable, instead of relying on derived classes. However, this does not solve the problem that a single modification of this attribute must be propagated across the project.

The complex setPacketA0 method (D-3) has kept the same design, while the problematic switch-case has swelled by 25% to reach almost a thousand lines of code. A significant portion of the additions are new comments, commented code, or "TODO" markers. It shows that this method is hard to maintain, as new additions are difficult to code, test, and approve.

G.5 Analysis

This section presents the categorization of the problems found in the three selected projects. It is an attempt to reuse patterns already identified by previous authors. What we are attempting to do is to present a more comprehensive list of categories. However, there are certainly other categorization approaches possible, such as Mäntylä's smell-oriented classification [4] or Brown et al. development, architecture and management anti-pattern categories [9]. Also, the categories among the approaches are certainly not exhaustive. The objective here is to show that a categorization of causes and symptoms is possible, and that it helps to explain how a pre-ID to post-ID discrepancy occurs, and possibly how to manage it.

G.5.1 Structural Categorization of Symptoms

The symptoms identified can be divided according to their structural level in the software component. From our results, three structural levels are proposed :

— Packaging problems
— Class problems
— Method problems

The package level is the main entrance to the software's source code. Packages (Java), namespaces (C++), and projects (C#) are all means to organize the code into cohesive structures. A good packaging structure can help the maintainer to accurately pinpoint the appropriate

package to adjust. An incorrect packaging structure can either be useless to the maintainer or worse, mislead him.

Problems 8-1 and 8-2 are examples of forgotten design decisions (or architectural drift [11]) leading to unstructured source code. Packaging design decisions made in the pre-ID were forgotten during implementation and are totally absent from the post-ID.

The class level is the encapsulation and relation level. Classes define a public interface and encapsulate its private inner workings. Class relations define external interactions with other classes (aggregation, composition) and the position of the class in the project (inheritance).

Problem 8-3 is an example of a lost encapsulation decision. Placing an inner object as public broke the pre-ID's planned encapsulation to the point that, in the post-ID, the majority of calls are in violation of the encapsulation principles. Problem 6-3, in contrast, is an example of a broken inheritance structure. The pre-ID model was appropriately implemented, but a development design decision broke it. The broken structure had a ripple effect across the project, preventing polymorphism use.

The method level is the algorithmic and access level. Methods define how a class works, but also who can and cannot access its inner workings. A single erroneous method can have a drastic effect on the project if it exposes an element which should remain hidden.

Problem 8-4 is an example of an access violation decision. An abuse of the static keyword in the post-ID causes a class to be visible to all, even though it was planned to be hidden from sight in the pre-ID. Problem 6-1 is an example of an erroneous algorithmic decision. Shoving the content of three classes into one method created an unmanageable mess.

G.5.2 Categorization of Causes

Causes found in the three selected projects can be naturally grouped into basic categories. The categories are aligned with the causes found, and different causes could yield a different categorization scheme. The intention here is to show that a categorization is both possible and meaningful, and not to define a final cause categorization scheme. Our approach yielded three base categories :

— Knowledge problems
— Artifact problems
— Management problems

These categories can be further split into sub-categories. The Knowledge base category can thus be split into two subcategories :

— Ignorance problems
— Misuse problems

Ignorance problems are linked to a lack of basic knowledge in the field. It is the knowledge related to facts and factual information, sometimes called "declarative" knowledge [23]. For example, not knowing the content of an API would be an ignorance problem, as shown in problem 6-2. This problem was also identified by van Gurp [12] : a lack of process awareness resulted in undocumented changes, which in turn led to inconsistency problems. We recall that a developer of the EMP8 project confirmed that they were not aware of potential problems with the singleton design pattern (8-4).

Misuse problems are linked to a lack of experience in the field. This is the knowledge linking performed actions with desired goals, sometimes called "procedural" knowledge [23] or knowhow. For example, the abuse of the static keyword would be a misuse problem, as the developers are not conscious of the link between making an object static and its impact on encapsulation, as seen in problem 8-4. This problem was also identified by van Gurp [12] : poor programming techniques led to poor code quality in a highly flexible 4GL language. The distinction with ignorance is that developers are aware of the basic concepts manipulated, but are unaware of how to integrate them into their current context. For example, problem 6-1 outlines the gap between knowing the visitor design pattern and performing an adaptation of it in a real project.

The artifact base category can be split into three sub-categories :

— Correctness problems
— Continuity problems
— Language problems

Correctness problems are linked to errors found in the pre-ID or incomplete pre-ID artifacts. The most common source or design erosion, according to the research, is the addition of new requirements [5, 6]. In a case study, van Gurp shows the impact of the arrival of new requirements during development, especially on a pre-ID architecture which is unable to accept them [12]. In the same study, he also shows the impact of an error in the pre-ID with a significant quality impact on the finished product ; the lack of normalization of the data model leading to data duplication in the database. Ambiguity errors in the pre-ID, as seen in problem 6-4, can also lead to problems in the post-ID. Problem 6-2 also shows the error between the toolbar and toolbox concepts.

Continuity problems are linked to forgotten design decisions also called architectural drift [11]. These causes are also commonly known as write-only artifacts, and involve design decisions

made in the pre-ID, but which were lost in the post-ID. Problems 6-1 and 8-1 are examples of where a decision to keep certain structures was completely lost in the post-ID. We recall that a developer of the EPM8 project confirmed that he was unaware that he had to follow the MVC architecture, even though it was explicitly written in the pre-ID artifacts (8-1).

Language problems are linked to incompatibility between the pre-ID languages used and the capacity of the coding language. Problem 8-1 describes the impact of this problem, where the basic UML packaging diagram proved insufficient to cover the structural capacity of the C# language. This problem was also identified by van Gurp [12] in their 4GL case study.

The management base category can be split into two sub-categories :

— Decision making problems
— Project management problems

Decision making problems are linked to team dynamics. These problems arise when design decisions are not sufficiently evaluated and debated. Major decisions made in an unconcerted way or without accounting for a ripple effect can lead to a broken structure, as shown in problem 6-3. A lack of impact analysis or impact monitoring can also lead to harmful decisions, as seen in problem 6-1. Adequate communication of the decisions made can also be a problem, as confirmed by the developer of the EPM8 project with the MVC architecture (8-1).

Project management problems are linked to estimation and planning errors. These problems arise when insufficient time and resources are allocated to a project. Deadline pressures can result in sub optimal design corrections, resulting in a low quality post-ID, as seen in van Gurp's PSS case study [12]. This cause seems to be the source of some encapsulation violation problems, as confirmed by one of the developers (8-2, 8-3).

G.6 Discussion

A great deal of work has been done on the causes, symptoms, and impacts of design erosion. However, these works largely failed to present case-specific evidence. We think the field is mature enough now to permit a meaningful reorganization of common causes and symptoms based on case study data. Categorization is a means to reach a higher level of abstraction and forgo case-specific details. The added value of our categorization is that it permits a more targeted approach to design management. Since it seems that the research has concluded that design erosion is inevitable, it is only sensible to manage that erosion as well as possible.

The categories and sub categories identified can serve as a baseline to implement a manage-

Table G.3 Sub-categories of causes and associated symptoms.

Sub categories	Symptoms
Ignorance	Visitor pattern eliminated (6-1) API class redundantly rewritten (6-2) Implication of MVC ignored (8-1) Impacts of static objects ignored (8-4) Company processes ignored [12]
Misuse	Inability to adapt the visitor pattern(6-1) Inheritance structure adaptation failure (6-3) Inability to implement MVC (8-2) Middle level function coding inability (8-3) Impacts of static elements unmanaged (8-4) Improper use of packages (D-1) Improper use of inheritance (D-2) Inability to adapt a class to a similar use (D-3) Poor programming techniques [12]
Correctness	Incorrect term used in specifications (6-2) Ambiguities in the design document (6-4) Ambiguities in the design document (D-1) New requirements during development [12]
Continuity	Visitor pattern forgotten (6-1) MVC architecture forgotten (8-1)
Language	Design language incompatible w/code (8-1) Proprietary language not accounted for [12]
Decision making	Lack of decision impact monitoring (6-1) Improper decision impact estimation (6-3) MVC architecture not communicated (8-1) New class not communicated (8-2) Architecture decision not monitored (D-1) Potential internal reuse not considered (D-3)
Project management	Task estimation error (8-2) Task estimation error (8-3) No resources for reviews (8-4) Unrealistic time constraints [12]

ment approach. Identification of knowledge, artifact, and management risks before the project is launched can lead to a better transformation from pre-ID to post-ID and to more durable software. As van Gurp writes about Parnas : "Aging of software [is] the result of bad design decisions" [12]. The impact of such decisions is presented in Table 3.

To be successful, a categorization effort presupposes that a meaningful abstraction from causes to categories of causes can be made. It is possible that only case-specific anecdotes can be identified. However, our limited study covered widely different projects with variable languages and approaches, along with previous studies conducted in the field. This article shows that problem causes found elsewhere can be linked to these categories.

G.7 Conclusion

This article reviews how the pre-implementation design (pre-ID) artifacts of three software projects were transformed into a post-implementation design (post-ID) extracted from the code. Our analysis focused on code-level discrepancies between the pre-ID and the post-ID, which reflect negative changes in design decisions. Previous higher level studies have shown that this design erosion is inevitable, and have already identified a number of causes for it. Our code-level approach enabled us to find new causes and to generalize those causes to a categorization system.

This categorization groups the discrepancy problems into three categories ; knowledge problems, artifact problems, and management problems. Knowledge problems outline harmful decisions made out of ignorance or misuse of design and language concepts. Artifact problems outline changes in pre-ID elements, the problems of write-only decisions and incompatibilities between the artifact's language and the possibilities of the code. Management problems outline poor team dynamics, leading to decision making shortfalls, along with project management mistakes leading to inappropriate resource and time constraints. Table 4 presents the categorization of this project and the problems found within the study. The problem column refers to sections 3.1, 3.2 and 3.3 of this study and external studies on design erosion.

This categorization enables software engineers to better target design management risks, and thus to propose software practices that will enable a final design that is more resistant to aging. Resistance to aging is crucial, if we want to prevent new software components from being phased out too quickly [6]. Given the escalating costs of software maintenance [1], any aging mitigation initiative can yield significant returns on investment.

This categorization is not exhaustive, and will need to be validated on a greater sample. It shows however that problem causes are varied and are not all within control of the developers.

Promoting anti-pattern awareness to developers is therefore insufficient as long as harmful causes outside of their reach are not dealt with.

Our analysis is limited to the identification and categorization of causes, and does not propose solutions. The rationale behind this limitation is that a single cause can be resolved using different solutions. For example, misuse causes can be mitigated through appropriate training, expert reviews, or by adjusting a team's composition [12]. However, knowing the causes can help a team find the solutions that are most appropriate to their context. Future research could focus on finding successful solutions to the causes of the problem identified here. The results shown herein demonstrate that many causes of the problems are outside the reach of developers. Software evolution control during development must therefore reach managers, and not be limited to developers. Poor change control and mitigation, the staple of software evolution, is evident in many of the identified problems, such as forgotten design decisions (6-1, 8-1) or poor decision impact monitoring (6-1, D-1).

G.8 Acknowledgments

We are grateful to Olivier Gendreau for his rigorous supervision of the EPM projects which was essential to the success of this study.

This study was made possible through the support of the Natural Sciences and Engineering Research Council of Canada, through grant 361163 and grant A0141.

G.9 References

[1] T. Yong and V.S. Mookerjee. Comparing Uniform and Flexible Policies for Software Maintenance and Replacement. IEEE Transactions on Software Engineering, 31(3) :238-

Table G.4 Categorization results.

Categories	Sub-categories	Problems found
Knowledge	Ignorance	6-1, 6-2, 8-1, 8-4, [12].
	Misuse	6-1, 6-3, 8-2, 8-3, 8-4, D-1, D-2, D-3, [12].
Artifact	Correctness	6-2, 6-4, D-1, [12].
	Continuity	6-1, 8-1.
	Language	8-1, [12].
Management	Decision-making	6-1, 6-3, 8-1, 8-2, D-1, D-2.
	Project	8-2, 8-3, 8-4, [12].

255, 2005.

[2] T. Mens and T. Tourwe. A survey of software refactoring. IEEE Transactions on Software Engineering, 30(2) :126-139, 2004.

[3] S. Cook, R. Harrison, M.M. Lehman and P. Wernick. Evolution in software systems : foundations of the SPE classification scheme. Journal of Software Maintenance and Evolution : Research and Practice, 18(1) :1-35, 2006.

[4] M.V. Mantyla. Empirical software evolvability – code smells and human evaluations. In Proceedings of the 26th IEEE International Conference on Software Maintenance (ICSM 2010), IEEE Computer Society, pages 1-6, 2010.

[5] J. Van Gurp and J. Bosch. Design erosion : Problems and causes. Journal of Systems and Software, 61(2) :105-119, 2002.

[6] D.L. Parnas. Software aging. In Proceedings of the 16th International Conference on Software Engineering (ICSE 1994), IEEE Computer Society, pages 279-287, 1994.

[7] W. Li, L. Etzkorn, C. Davis and J. Talburt. Empirical study of object-oriented system evolution. Information and Software Technology, 42(6) :373-381, 2000.

[8] N. Moha, Y.-G. Gueheneuc, L. Duchien and A.-F. Le Meur. DECOR : A method for the specification and detection of code and design smells. IEEE Transactions on Software Engineering, 36(1) :20-36, 2010.

[9] W.J. Brown, R.C. Malveau, H.W. McCormick and T.J. Mowbray. AntiPatterns : Refactoring Software, Architectures, and Projects in Crisis. Wiley, 1998.

[10] J. Bosch and L. Lundberg. Software architecture - Engineering quality attributes. Journal of Systems and Software, 66(3) :183-186, 2003.

[11] D.E. Perry and A.L. Wolf. Foundations for the study of software architecture. ACM SIGSOFT Software Engineering Notes, 17 :40-52, 1992.

[12] J. Van Gurp, S. Brinkkemper and J. Bosch. Design preservation over subsequent releases of a software product : A case study of Baan ERP. Journal of Software Maintenance and Evolution, 17(4) :277-306, 2005.

[13] S. Wong, Y. Cai and M. Dalton. Detecting Design Defects Caused by Design Rule Violations (Report DU-CS-09-04). Drexel University, 2009.

[14] S. Meyers. Effective C++ : 55 Specific Ways to Improve Your Programs and Designs. Addison-Wesley Professional, 2005.

[15] S. Chaki, A. Diaz-Pace, D. Garlan, A. Gurfinkel and I. Ozkaya. Towards engineered architecture evolution. In Proceedings of the 2009 ICSE Workshop on Modeling Software Engineering (MiSE 2009), IEEE Computer Society, pages 1-6, 2009.

[16] U. Eklund and T. Arts. A Classification of Value for Software Architecture Decisions. In Proceedings of the 4th European Conference on Software Architecture (ECSA 2010), Springer-Verlag, 2010.

[17] M. Fowler, K. Beck, J. Brant, W. Opdyke and D. Roberts. Refactoring : Improving the Design of Existing Code. Addison-Wesley Professional, 1999.

[18] E.W. Deming. Out of the Crisis. Massachusetts Institute of Technology, 1986.

[19] M. Mäntylä. Bad Smells in Software – a Taxonomy and an Empirical Study. Helsinki University of Technology, 2003.

[20] Microsoft Patterns & Practices Team. Microsoft Application Architecture Guide, 2nd Edition. Microsoft Press, 2009.

[21] E. Gamma, R. Helm, R. Johnson and J.M. Vlissides. Design Patterns : Elements of Reusable Object-Oriented Software. Addison-Wesley Professional, 1994.

[22] M. Fowler. Reducing coupling. IEEE Software, 18(4) :102-104, 2001.

[23] S.K. Reed. Cognition : Theory and Applications. Wadsworth Publishing, 2009.