Analyse multi-facettes et opérationnelle pour la transformation des systèmes d'information

Mahugnon Honore Houekpetodji

▶ **To cite this version:**

Mahugnon Honore Houekpetodji. Analyse multi-facettes et opérationnelle pour la transformation des systèmes d'information. Informatique et langage [cs.CL]. Université de Lille, 2022. Français. NNT : . tel-04402943v1

HAL Id: tel-04402943
https://hal.science/tel-04402943v1

Submitted on 18 Jan 2024 (v1), last revised 18 Jan 2024 (v2)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Université de Lille, Faculté des sciences et technologies

Département de formation doctorale en informatique **École doctorale MADIS Lille**
UFR IEEA

Analyse multi-facettes et opérationnelle pour la transformation des systèmes d'information

THÈSE

présentée et soutenue publiquement le 24 Juin 2022

pour l'obtention du

Doctorat de l'Université de Lille
(spécialité informatique)

par

HOUEKPETODJI Mahugnon Honoré

Composition du jury

Président :	Mireille BLAY-FORNARINO	(Professeur – Université de Nice)
Rapporteurs :	Olivier BARAIS	(Professor – Université de Rennes 1)
	Chouki TIBERMACINE	(Professeur associé HDR – Université de Montpellier)
Examinateur :		
Directeur de thèse :	Nicolas Anquetil	(Professeur associé HDR – Université de Lille 1)
Co-Encadreur de thèse :	Stéphane Ducasse	(Directeur de recherche – INRIA Lille Nord-Europe)

Centre de Recherche en Informatique, Signal et Automatique de Lille — UMR 9189
INRIA Lille - Nord Europe

Remerciements

Je tiens à remercier en général tous nos enseignants, l'Institut Supérieur des Sciences Appliquées et Technologies de Sousse qui m'ont dispensé des cours durant notre deuxième cycle d'études universitaires, ainsi que tout le personnel administratif qui m'a apporté son aide.

Très sincèrement, je remercie mes encadrants **Mr. Nicolas Anquetil, Mr. Stéphane Ducasse, Mme. Fatiha Djareddir et Mr. Jérôme Sudich** qui m'ont offert toutes ses gratitudes et m'ont assisté tout au long de cette thèse. Je passe aussi mes profonds remerciements à toute l'Équipe RMod INRIA qui m'a bien accueilli et m'a fourni un environnement favorable pour ma thèse de doctorat.

J'adresse également mes remerciements à tous ceux qui, de près ou de loin, ont participé à la mise en œuvre de cette thèse de thèse, en particulier **Mr. Denis Houékpétodji**, **Mr. Roger Houékpétodji**, **Mme. Levine Lemvo**, **Mr. Oleksandr Zaitsev**, **Mr. Mario Sanchez**, **les parents**, sans oublier les collègues de la société CIM et de l'équipe RMoD qui m'ont soutenus.

Résumé

L'un des facteurs de réussite des éditeurs de logiciels est leur capacité à livrer rapidement des produits de bonne qualité. Cette capacité de livraison est souvent influencée par le modèle de développement logiciel adopté par la société. Pour cela, elles doivent améliorer leurs pratiques de développement logiciel pour s'adapter aux demandes du marché. Dans cette thèse, nous avons travaillé un partenaire industriel qui est une société française de taille moyenne appelée CIM qui change ces pratiques de développement.

La société CIM est une éditrice, intégratrice, hébergeur et infogérant de solutions pour l'assurance de personnes en santé et prévoyance. Elle offre une expertise en santé et prévoyance acquise après plus de 30 ans auprès de ses clients. La société a effectué une analyse de risque pour son évolution et sa croissance en 2017, d'où il ressort qu'elle a des problèmes de qualité et de retard de livraison, notamment sur son logiciel principal Izy Protect écrit en PowerBuilder. Alors la société a décidé de moderniser ses pratiques de développement logiciel vers les pratiques du modèle de développement agile, SCRUM en particulier. Durant la modernisation des pratiques, la société a fait face aux conditions exceptionnelles liées à la crise de la COVID-19 avec la généralisation du travail à distance.

Si les avantages de tels processus de développement sont bien documentés, la transition présente des défis. L'objectif de cette thèse est d'évaluer et suivre le processus de modernisation dans la société CIM et de documenter les bénéfices et les défis de la modernisation perçus par les pratiquants.

Dans cette thèse, nous proposons des actions concrètes pour améliorer les pratiques de la société. De plus, afin de pouvoir vérifier la pertinence de ses actions (et d'autres menées par ailleurs par la société), nous avons choisi une méthode d'évaluation qualitative et quantitative. Après la modernisation des pratiques par la société CIM, nous avons mené une étude exploratoire en nous appuyant sur des entretiens approfondis et semi-structurés avec des participants au processus de développement afin de comprendre comment ces changements ont impacté leur travail. Dès que possible, nous validons les conclusions des entretiens par des données empiriques.

Mots-clés : Modèle de développement logiciel, COVID-19, Entretiens semi-structurés, Industriel, *SCRUM*

Abstract

One of the key factors of software companies is their ability to deliver good quality products quickly. This delivery capability is often influenced by the software development model adopted by the company. To do so, they must improve their software development practices to adapt to market demands. In this thesis, we have worked with an industrial partner which is a medium-sized French company called CIM that is changing these development practices.

The company CIM is a publisher, integrator, host and outsourcer of software for personal insurance in health and welfare. It offers expertise in health and welfare acquired after more than 30 years with its customers. The company conducted a risk analysis for its evolution and growth in 2017, from which it emerged that it had quality and delivery delay issues, particularly on its main software Izy Protect written in PowerBuilder. So the company decided to modernize its software development practices towards the agile development model practices, SCRUM in particular. During the modernization of practices, the company faced exceptional conditions related to the COVID-19 crisis with the generalization of remote work.

While the benefits of such development processes are well documented, the transition presents challenges. The purpose of this thesis is to evaluate and monitor the modernization process in the company CIM and to document the benefits and challenges of modernization as perceived by practitioners.

In this thesis, we propose concrete actions to improve the company's practices. Furthermore, in order to verify the relevance of these actions (and others carried out by the company), we have chosen a qualitative and quantitative evaluation method. After CIM modernized its practices, we conducted an exploratory study based on in-depth, semi-structured interviews with participants in the development process to understand how these changes impacted their work. Wherever possible, we validate the interview results with empirical data.

Keywords : Software development model, COVID-19, Semi-structured interviews, Industrial, *SCRUM*

Table des matières

1 Introduction **1**
- 1.1 Contexte . . . 1
- 1.2 Problématique . . . 2
- 1.3 Notre solution en bref . . . 2
- 1.4 Notre contribution . . . 3
- 1.5 Structure de la thèse . . . 3
- 1.6 Liste des publications . . . 4

2 État de l'Art **5**
- 2.1 Introduction . . . 5
- 2.2 Processus de développement logiciel . . . 5
  - 2.2.1 Processus de développement logiciel . . . 6
  - 2.2.2 Modèles de développement logiciel . . . 7
  - 2.2.3 *SCRUM* . . . 9
- 2.3 Qualité logicielle en *SCRUM* . . . 10
- 2.4 Problèmes liés à l'adoption des méthodes de *SCRUM* . . . 11
- 2.5 Contexte de recherche selon Kitchenham . . . 13
  - 2.5.1 Le logiciel maintenu . . . 13
  - 2.5.2 Les activités de maintenance . . . 14
  - 2.5.3 Procédé de maintenance logicielle . . . 15
  - 2.5.4 Ressources humaines . . . 15
- 2.6 Évaluation de processus de développement . . . 16
  - 2.6.1 Méthodes de collecte de données . . . 16
  - 2.6.2 Analyse des données . . . 20
- 2.7 Résumé du chapitre . . . 21

3 Description du contexte **23**
- 3.1 Introduction . . . 23
- 3.2 Le langage PowerBuilder . . . 24
- 3.3 Présentation du contexte de la société CIM . . . 27
  - 3.3.1 Le logiciel maintenu . . . 28
  - 3.3.2 Les activités de maintenance . . . 28
  - 3.3.3 Procédé de maintenance logiciel . . . 30
  - 3.3.4 Ressources humaines . . . 32
- 3.4 Résumé du chapitre . . . 32

4 Situation avant les changements de pratiques **35**
4.1 Introduction 35
4.2 Etude quantitative 36
4.2.1 Collecte des données 36
4.2.2 Nettoyage des données 37
4.2.3 Indicateurs à mesurer 38
4.2.4 Analyse des données 39
4.2.5 Résultat de mesure quantitative 40
4.3 Étude qualitative du processus de développement dans la société CIM 45
4.3.1 Collecte des données 45
4.3.2 Analyse des données 47
4.3.3 *Stand-up meeting* 49
4.3.4 Organisation d'équipe 49
4.3.5 Cycle de développement du logiciel 50
4.3.6 Gestion du code source 51
4.3.7 Qualité du code 52
4.4 Résumé du chapitre 53

5 Modernisation des pratiques de développement **55**
5.1 Introduction 55
5.2 Pratiques modernisées 56
5.3 Étude qualitative des pratiques modernisées 59
5.3.1 Les *stand-up meetings* 59
5.3.2 Changement organisationnel en équipes 61
5.3.3 Cycle de développement du logiciel 65
5.3.4 Gestion de code source 66
5.3.5 Qualité de code 67
5.4 Résumé du chapitre 68

6 Tests et exécution automatique **71**
6.1 Introduction 71
6.2 Solutions existantes 72
6.2.1 Test fonctionnel sur Izy Protect 72
6.2.2 Cadriciel (*framework*) de test unitaire 73
6.3 Problèmes avec la pratique de tests 75
6.3.1 Problèmes techniques liés à Izy Protect 75
6.3.2 Problèmes techniques liés à PowerBuilder 80
6.3.3 Problèmes liés aux ressources humaines 81
6.4 Piste d'améliorations 81
6.4.1 Résolution des problèmes 82
6.4.2 Serveur d'exécution automatique des tests unitaires 85

6.4.3 Génération des tests unitaires 87
6.5 Résultat des tests unitaires 88
6.6 Conclusions du chapitre 89

7 Conclusion 91
7.1 Synthèse 91
7.2 Contribution 93
7.3 Perspectives et travaux futurs 94

A Mode opératoire de *PBUnit* 95
A.1 Généralités sur les tests Unitaires 95
A.1.1 Tests 95
A.1.2 Tests unitaires 95
A.1.3 Les *smokes tests* ou tests de vérification de santé 97
A.2 PBUnit 97
A.2.1 Présentation 97
A.3 Ecriture de tests avec PBUnit 98
A.3.1 Préparation de l'environnement 98
A.3.2 Création de cas de tests 98
A.3.3 Explication de l'interface graphique de PBUnit 103

Bibliographie 107

Table des figures

3.1 Environnement de développement intégré de PowerBuilder 25

4.1 Temps pour fermer les tickets d'évolution (en bleu) et de défaut (en rouge) 41
4.2 Temps passé par les programmeurs implémenter une solution pour les tickets d'évolution (en bleu) et de défaut (en rouge). 42
4.3 Temps passé par les programmeurs pour tester les tickets d'évolution (en bleu) et de défaut (en rouge) 43
4.4 Temps passé par l'équipe de testeur pour tester les tickets d'évolution (en bleu) et de défaut (en rouge) 43
4.5 Différence entre le temps passé par les programmeurs et le temps estimé pour les tickets d'évolution (en bleu) et de défaut (en rouge) 44

5.1 Temps de développement sur les tickets fermés 61
5.2 Durée de vie des tickets (de la date d'ouverture à la date de fermeture) 62
5.3 Nombre de tickets fermés par programmeur 63
5.4 Retour sur les tickets fermés 63

6.1 Interface de *PBUnit* 74
6.2 Dépendance entre des classes abstraites d'Izy Protect. 76
6.3 Diagramme de séquence d'exécution des tests unitaires 86

A.1 Test case 96
A.2 PBUnit 97
A.3 Création d'une bibliothèque de tests unitaire 99
A.4 Lancer *PBUnit* 104
A.5 *PBUnit* 104
A.6 Choisir une target 104
A.7 Selectionner la *target* à son emplacement 105
A.8 PBUnit après chargement des tests 105
A.9 PBUnit après exécution des tests 106
A.10 *PBUnit* après exécution des tests 106

Liste des tableaux

4.1 Tickets 40
4.2 Liste des développeurs d'Izy Protect. Expérience professionnelle totale et ancienneté dans l'entreprise en années. La dernière colonne indique les participants aux entrevues. 46
4.3 Liste des questions des entrevues 48

6.1 Nombre de méthodes testés 89

CHAPITRE 1
Introduction

Sommaire

1.1 Contexte **1**
1.2 Problématique **2**
1.3 Notre solution en bref **2**
1.4 Notre contribution **3**
1.5 Structure de la thèse **3**
1.6 Liste des publications **4**

1.1 Contexte

Cette thèse s'est déroulée dans le cadre d'un partenariat industriel avec la société CIM [1]. La société *CIM* est une SAS au capital social de 200 000 euros détenu à 100% par *DL Software*. Elle est une éditrice, intégratrice, hébergeur et infogérant de solutions pour l'assurance de personnes en santé et prévoyance. Elle offre une expertise en santé et prévoyance acquise après plus de 30 ans auprès de ses clients. La société *CIM* est hébergeur de ses solutions pour 90% de ses clients et plus de 1000 utilisateurs. Toutes les thématiques d'infrastructure et de surveillance des flux sont intégrées à cette offre. La société est propriétaire de ses serveurs. Tous les éléments actifs des systèmes et tous les éléments de stockage sont achetés par la société, gérés et supervisés par les équipes de la société. Aucun sous-traitant n'intervient dans les opérations quotidiennes d'hébergement, d'exploitation des solutions et des données hébergées.

La société *CIM* est certifiée *Microsoft GOLD Partner*. Elle est l'éditrice des progiciels de la gamme Izy Links et assure l'intégration de l'ensemble des briques de cette gamme ainsi que des briques partenaires nécessaires à la bonne réussite du projet. La société a effectué une analyse de risque pour son évolution et sa croissance en 2017, d'où il ressort qu'elle a des problèmes de qualité et de retard de livraison, notamment sur son logiciel principal Izy Protect écrit en PowerBuilder. Alors la société a décidé de moderniser ses pratiques de développement

1. https ://www.sa-cim.fr/

logiciel vers les pratiques du modèle de développement agile, *SCRUM* en particulier. Durant la modernisation des pratiques, la société a fait face aux conditions exceptionnelles liées à la crise de la COVID-19 avec la généralisation du travail à distance. Si les avantages des processus de développement agiles sont bien documentés, l'impact de la transition est moins connu. De plus, d'après de nombreux agilistes, le travail à distance peut poser des problèmes impactant l'efficacité des pratiques, car la collocation et les interactions en face à face sont importantes pour ces dernières [Smite 2021].

1.2 Problématique

Le contexte industriel dans lequel s'inscrit cette thèse nous permet d'identifier deux problèmes en ce qui concerne la modernisation des pratiques de développement logiciel :

Evaluation et suivi du processus de modernisation : Pour savoir si les pratiques apportent les bénéfices attendus, il est important d'évaluer le processus de développement de la société CIM à travers des indicateurs. Bien que [Singer 2008] détaillent des techniques d'évaluation de processus de développement de logiciels, leurs applications pour identifier des indicateurs d'évaluation dépendent souvent des données relatives au contexte.

Les bénéfices et les défis de la modernisation perçus : Les bénéfices des modèles agiles sont souvent mis en avant par la littérature [Stoica 2013, Vohra 2013, Akinsola 2020]. Par ailleurs, les bénéfices perçus de la transition des anciennes pratiques et les défis auxquels les professionnels sont confrontés face à la modernisation des pratiques sont peu documentés [Chen 2015]. Comme nous l'avons dit ci-dessus, le travail à distance peut affecter le bénéfice des pratiques agiles. Alors, quels sont les impacts durant la transition ?

1.3 Notre solution en bref

Pour atteindre nos objectifs et assurer que nos solutions s'appliquent à la société CIM, nous avons étudié la littérature et analysé le contexte de la société. Cela nous a permis d'identifier des problèmes que nous allons détailler dans la suite de cette thèse. Sur la base de cette étude bibliographique et de la situation actuelle, nous avons proposé des actions concrètes pour améliorer les pratiques de la société : mise en place d'un système de contrôle de version, développement d'analyseur syntaxique, etc. Pour pouvoir vérifier la pertinence de ses actions (et d'autres menées par ailleurs par la société), nous avons choisi une méthode d'évaluation qualitative et quantitative.

Après la modernisation des pratiques par la société CIM, nous avons mené une étude exploratoire en nous appuyant sur des entretiens approfondis et semi-structurés avec 17 participants au processus de développement afin de comprendre comment ces changements ont impacté leur travail. Nous avons adopté l'approche de la *Grounded Theory* [Khadka 2014], une technique qualitative, pour analyser les entretiens. Lorsque cela était possible, les conclusions des entretiens ont été validées par des données empiriques (base de données des tickets de l'entreprise, historique des commits de la gestion du code source, courriels).

1.4 Notre contribution

D'un point de vue recherche, nous apportons les contributions suivantes :

1. Nous documentons la perception industrielle de certaines pratiques de développement agile ;
2. Nous documentons les avantages et les inconvénients perçus de ces pratiques ;
3. Nous identifions l'impact du travail à distance sur certaines pratiques agiles.

Avec cette thèse, nous avons répondu aux attentes de la société. En effet, nous avons proposé de l'outillage permettant d'évaluer le processus de développement de la société. Nous avons aussi proposé des outils qui permettent d'améliorer la qualité logicielle de la société.

1.5 Structure de la thèse

Cette thèse est structurée de la manière suivante :

— Le Chapitre 2 fournit l'état de l'art sur les modèles de développement logiciel ainsi que les techniques d'évaluation de modèle de développement logiciel.

— Le Chapitre 3 décrit le contexte de cette thèse suivant les lignes directrices de [Kitchenham 1999].

— Le Chapitre 4 aborde les mesures quantitatives des pratiques de développement de la société CIM avant la modernisation.

— Le Chapitre 5 refait l'étude qualitative sur les pratiques de développement de la société CIM après la modernisation.

— Le Chapitre 6 présente les problèmes et les solutions de la mise en place des tests à la société CIM.

— Le Chapitre 7 conclut cette thèse et fournit des perspectives pour de futurs travaux.

1.6 Liste des publications

Voici la liste des articles publiés ou soumis à révision dans le cadre de cette thèse par ordre chronologique :

— *Improving practices in a medium french company : First step* [Honore 2020].

— *Modular Moose : A New Generation of Software Reverse Engineering Platform* [Anquetil 2020].

— *Report From The Trenches A Case Study In Modernizing Software Development Practices* [Houekpetodji 2021].

— Soumission à un journal d'un article étendu reprenant les avancées de cette thèse.

CHAPITRE 2

État de l'Art

Sommaire

2.1 Introduction 5
2.2 Processus de développement logiciel 5
2.3 Qualité logicielle en *SCRUM* 10
2.4 Problèmes liés à l'adoption des méthodes de *SCRUM* 11
2.5 Contexte de recherche selon Kitchenham 13
2.6 Évaluation de processus de développement 16
2.7 Résumé du chapitre 21

2.1 Introduction

La société CIM a entamé un projet de modernisation de son processus de développement logiciel afin de répondre à la demande de ses clients. La modernisation des pratiques de développement logiciel relève beaucoup de défis [Nerur 2005]. Pour cela, nous cherchons à aider la société à moderniser ces pratiques de développement et à évaluer les nouvelles pratiques afin de lui faciliter la mise en place de ses pratiques et de s'améliorer. Nous étudions dans ce chapitre les modèles de développement logiciel dans la Section 2.2 ; ensuite, nous étudions la qualité logicielle en modèle agile dans la Section 2.3, puis nous abordons les problèmes liés à la migration vers *SCRUM* dans la Section 2.4. Pour aider la société à évaluer l'impact de la modernisation des pratiques, nous étudions aussi ici comment clarifier le contexte de recherche dans la Section 2.5 puis comment évaluer de processus de développement dans la Section 2.6. Enfin, nous conclurons le chapitre dans la Section 2.7.

2.2 Processus de développement logiciel

Pour développer son logiciel, la société CIM utilise un ensemble de pratiques qui constitue son processus de développement logiciel. Généralement, les processus de développement logiciels sont régis par des modèles de développement logi-

ciel connus [Despa 2014] tels que le modèle en cascade, le modèle en V, le modèle itératif et incrémental, les modèles agiles, etc. La qualité logicielle et le *time to market* d'une société dépendent fortement du processus de développement logiciel qu'elle a adopté [Aitken 2013]. C'est d'ailleurs pour cette raison que beaucoup de sociétés s'orientent vers des modèles de développement logiciel moderne pour répondre aux exigences du marché dynamique de nos jours. Ceci est le cas de la société CIM, qui modernise ses pratiques de développement logiciel basé sur un processus traditionnel (modèle en V) vers des pratiques conseillées par les modèles agiles notamment le *SCRUM*.

Ici, nous allons d'abord expliquer la notion processus de développement logiciel, puis présenter les modèles de développement logiciel avec leurs avantages et leurs inconvénients.

2.2.1 Processus de développement logiciel

Selon [Leau 2012, Stoica 2013], un processus de développement logiciel comporte souvent : l'analyse et la spécification des exigences, la conception de l'architecture, l'implémentation, les tests, le déploiement et maintenance. Chacune de ces étapes ont été expliquées par [Stoica 2013]. Ici, nous allons expliquer ces dernières en nous inspirant de la littérature.

Analyse et Spécification des exigences : Consiste à analyser les exigences du projet. Elle est effectuée par des membres expérimentés de l'équipe, à partir des données fournies par les clients, le service commercial, les analystes et les experts du secteur.

Après l'analyse des exigences, un document détaillant clairement les exigences fonctionnelles et non fonctionnelles du projet est défini. Ces exigences doivent être validées par les analystes et le client.

Conception de l'architecture : Au niveau de la conception de l'architecture du logiciel, les architectes logiciels proposent des architectures logicielles en se basant sur le document spécifiant les exigences du futur logiciel. Les architectures doivent présenter toutes les parties du logiciel, les différentes communications ou échanges de données entre ces dernières ou le monde extérieur ainsi que les moyens de communication ou d'échange.

Implémentation : À ce stade du processus de développement, le code source du futur logiciel est produit. Les programmeurs doivent suivre les exigences pour écrire le code source du logiciel.

Les tests : C'est une étape essentielle avant que le logiciel développé ne soit livré au client. Le rôle principal des tests est de vérifier que les fonctionnalités du logiciel fonctionnent bien comme indiqué dans le document de spécification. Son rôle est aussi de vérifier que les différentes parties du logiciel

fonctionnent bien ensemble. La phase de test permet de réduire le nombre de défauts et de problèmes rencontrés par les utilisateurs. Il en résulte une plus grande satisfaction des utilisateurs et un meilleur taux d'utilisation.

Le déploiement : Une fois que l'étape de tests du logiciel est passée avec succès, le logiciel est prêt pour être déployé sur le marché.

La maintenance : Au cours de son utilisation, le logiciel est maintenu pour les clients à travers la correction des défauts ou des évolutions.

2.2.2 Modèles de développement logiciel

Comme annoncé ci-dessus, il existe plusieurs modèles de développement logiciel. Il existe en particulier les modèles utilisées par la société CIM : le modèle en V qui est une évolution du modèle en cascade, puis les modèles agiles basées sur le modèle itératifs et incrémental. Ici, nous allons présenter les modèles historiquement liées aux pratiques adoptées par la société CIM (le modèle en cascade, le modèle en V, le modèle itératif et incrémental et les modèles agiles) ; ainsi que leurs avantages et inconvénients.

Le modèle en cascade : Le modèle en cascade a été proposé pour la première fois en 1970 par [Royce 1987] comme une méthode pour enseigner le développement logiciel. Il consiste à un enchainement successif des étapes citées dans la Section 2.2.1. Chaque étape doit être achevée avant que la suivante puisse commencer.

Ce modèle présente l'avantage d'avoir une documentation claire avec des étapes claires avant le début d'un projet. Ceci le rend facile à comprendre et à implémenter [Alshamrani 2015]. Dans le modèle en cascade, les étapes sont traitées et achevées une par une. C'est-à-dire que la spécification d'un projet est achevée avant la conception et la conception est achevée avant le codage par exemple. Selon [Bhuvaneswari 2013], il renforce les bonnes habitudes. Le modèle en cascade fonctionne bien sur des projets matures et fournit une structure aux équipes inexpérimentées [Munassar 2010].

Par ailleurs, ce modèle n'est pas flexible aux changements, car il suppose que les exigences sont précises au début du projet de développement. Ce qui ne reflète pas la réalité itérative du développement logiciel [Bhuvaneswari 2013]. Dans un modèle en cascade, de nombreux problèmes concernant une phase particulière peuvent surgir après la fin de ce dernier, ce qui peut engendrer un système mal structuré, rendant ainsi le niveau de risque et d'incertitude élevé [Alshamrani 2015, Munassar 2010]. De plus, les délais de livraison ne sont pas souvent respectés dans ce modèle.

Le modèle en V : Le modèle en V est axé sur la vérification et la validation. Chaque étape doit être achevée, vérifiée et validée avant que la suivante

puisse commencer. Comme le modèle en cascade, la documentation et les étapes sont claires avant le début du projet. Ce qui le rend facile à utiliser [Munassar 2010]. De plus, il y a de test à toutes les étapes. Ces tests permettent de trouver des défauts le plus tôt possible lors de l'avancement d'un projet avec ce modèle. Cela permet une meilleure chance de succès d'un projet dans le modèle en V par rapport au modèle en cascade.

Par contre, le modèle en V est très rigide et peu flexible par rapport aux changements. De plus, avec ce modèle, un projet est fonctionnel vers la fin du cycle avec un faible taux de succès. Aussi, les délais de livraison ne sont généralement pas respectés [Munassar 2010].

Le modèle itératif et incrémental : Dans le modèle incrémental, les exigences sont divisées en modules plus faciles à développer et à livrer. Chaque module est développé dans une itération sous forme d'un miniprojet. Ainsi, il passe par l'analyse des besoins, la conception, la mise en œuvre et les tests. Au cours du premier module, une version de travail du logiciel est créée. Chaque version suivante ajoute de nouvelles caractéristiques et fonctionnalités à la précédente. Le processus se poursuit jusqu'à ce que le système soit terminé.

Le modèle itératif et incrémental facilite la gestion des objectifs et l'identification des risques. Il permet de livrer des prototypes au fur et à mesure des itérations. Cela fait que moins de temps est consacré à la documentation et plus de temps est accordé à la conception et des fonctionnalités importantes sont disponibles rapidement [Alshamrani 2015].

Par ailleurs, chaque itération est une structure rigide qui ressemble au modèle en cascade à petite échelle [Despa 2014]. Aussi, il demande plus de planification par rapport aux modèles en cascade et en V.

Modèles agiles L'objectif des modèles de développement agiles est d'accélérer le développement des logiciels en tenant compte du retour des clients sur l'avancement du développement. Ainsi, ils permettent au client de demander des changements à n'importe quel stade du développement du projet, si nécessaire [Vohra 2013]. Les modèles agiles s'inspirent du modèle itératif et incrémental où le logiciel à construire est divisé en de petits modules. Un module est développé durant une itération ou *Sprint* . Ainsi, le logiciel est construit en plusieurs *sprints* successifs [Larman 2004]. La durée d'un *Sprint* varie entre de 2 à 4 semaines [Schwaber 2004, Upadhyay 2021]. Chaque *Sprint* est considéré comme un miniprojet comprenant l'analyse des exigences, la programmation et les tests. À la fin d'un *Sprint* , une version stable du projet est publiée. Une version complète du projet est publiée et livrée lorsque le dernier *Sprint* est achevé [Larman 2004].

Les principes de base des modèles agiles sont codifiés dans le manifeste

« *Agile Software Development Manifesto* » en 2001 ([Picek 2009, Larman 2004, Mohammad 2017]) :

— Les interactions entre les individus (coéquipiers, client, etc.) sont plus importantes que les outils, les processus.

— Un logiciel fonctionnel est plus important qu'une documentation complète et compréhensible.

— La collaboration avec le client est plus important qu'un contrat formel.

— La réponse aux changements est plus importante qu'un suivi de plans formels.

Ces différents principes permettent aux modèles agiles d'avoir la capacité de répondre aux exigences changeantes d'un projet dans des délais conforment à ceux estimés. De plus, il implique les clients, ce qui réduit le risque de non-respect des exigences.

Par ailleurs, ces modèles coûtent beaucoup de ressources et plus d'effort de planification.

Le « *Agile Software Development Manifesto* » concerne une famille de méthodes appelées 'Agiles' telles que : *Extreme Programming,* SCRUM*, eXtreme testing, Crystal Family of Methodologies*, etc. [Vohra 2013].

2.2.3 *SCRUM*

Comme nous l'avons expliqué dans la section ci-dessus, la société CIM veut moderniser ses pratiques de développement vers les pratiques conseillées par les modèles agiles, en particulier le modèle *SCRUM*. Après avoir présenté les méthodes agiles dans la section précédente, nous allons maintenant présenter le modèle agile *SCRUM* dont les pratiques sont adoptées par la société CIM.

Le modèle *SCRUM* est apparue au milieu des années 90 [Schwaber 1997] et a gagné le plus de popularité parmi les modèles agiles dans l'industrie d'après [Hanslo 2018]. Il comporte trois rôles (*Product owner* , *Scrum master* et les membres de l'équipe) et quatre réunions (la planification du sprint, les *Stand-up meetings*, la revue de *Sprint* et la réunion de rétrospective) selon [Mahalakshmi 2013]. Mahalakshmi et *al.* ont expliqué ces rôles et ces réunions.

Le *Product owner* représente le client au sein de l'équipe *SCRUM* et l'équipe auprès du client. Il travaille avec l'équipe *SCRUM* et est responsable de la planification des tâches du projet. Il définit aussi les fonctionnalités à développer, les ajustements s'il y a besoin, la priorité des exigences, la validation du résultat final, la date d'arrêt du projet pour livraison, etc. Le *Scrum master* est un guide pour l'équipe *SCRUM*. Il a pour rôle d'aider l'équipe à résoudre les obstacles, de veiller au bon déroulement du *Sprint* et de protéger l'équipe des interférences externes. Il

organise aussi les *Stand-up meetings*. L'équipe *SCRUM* a pour rôle de développer et tester le projet à chaque *Sprint*. Elle comporte 5 à 10 personnes. Les membres de l'équipe peuvent être des programmeurs, des testeurs, etc.

La réunion de planification de *Sprint* a pour but principal d'analyser ce qui doit être fait et comment le faire pour le *Sprint* à venir. Les participants à cette réunion sont le *Product owner*, le *Scrum master* et les membres de l'équipe. Le *Product owner* est responsable de la priorisation des sujets qui sont les plus importants durant la réunion.

Durant chaque *Sprint*, il y a des *Stand-up meetings* qui sont des réunions journalières d'une durée de 15 minutes auxquelles participent le *Scrum master* et les membres de l'équipe *SCRUM*. Chaque participant est amené à répondre aux questions : « Qu'est-ce que j'ai fait hier ? », « Qu'est-ce qui m'a retenu ? », « Qu'est-ce que je vais faire aujourd'hui ? ».

À la fin du sprint, l'équipe *SCRUM* tient une réunion de revue du *Sprint* afin de présenter l'incrément réalisé à tous ceux qui sont intéressés, en particulier les parties prenantes externes (clients, autres équipes *SCRUM*, etc.). Le *Product owner* revoit les sujets de la réunion de planification de *Sprint* qui ne sont pas finis, revoit leurs priorités afin qu'ils soient candidats pour le planning du prochain *Sprint* . Le *Scrum master* , aide le *Product owner* et les autres parties prenantes à convertir les commentaires en de nouveaux sujets à planifier par le *Product owner* . La réunion de revue dure deux heures au maximum.

À la suite de la réunion de revue du *Sprint*, une réunion de rétrospective qui dure 15 à 30 minutes est organisée. Toute l'équipe *SCRUM* et les parties prenantes sont conviées à cette réunion, pour analyser les problèmes du *Sprint* qui vient de finir et les approches de solution pour remédier à ces problèmes à l'avenir.

2.3 Qualité logicielle en *SCRUM*

Un des points importants du *SCRUM* est la qualité logicielle. Cela est fait à travers de tests et outils d'analyse statique de code [Marcilio 2019]. Les tests et outils d'analyse de code font partie des pratiques introduites par la société CIM sur son logiciel que nous allons présenter dans le Chapitre 6.

Les tests sont caractérisés par les tests manuels et les tests automatiques. L'automatisation des tests consiste à développer et exécuter des scripts (fonctionnels ou unitaires) de tests, à vérifier des spécifications de test et utiliser des outils d'automatisation de tests [Taipale 2011]. Toutefois, la mise en place des tests automatiques coûte cher au début en termes de temps, formation du personnel, outils, etc. [Rafi 2012].

Les outils d'analyse statique de code (*linter*) analysent le code source d'un programme sans l'exécuter [Nielson 1999]. Ils utilisent souvent des analyseurs

syntaxiques de code (*parser*) pour analyser les instructions de code et des règles de programmation pour détecter des défauts dans le code source, permettant ainsi l'amélioration de la qualité du code source durant le cycle de développement d'un logiciel [Johnson 2013]. Ces outils complémentent les tests automatiques, car ils peuvent détecter des défauts qui ne sont pas forcément détectables par les tests ou l'inspection manuelle du code [Marcilio 2019]. De plus, ils peuvent être automatisés pour être exécutés automatiquement après les modifications de code. Il existe plusieurs outils d'analyse statique de code tels que *CheckStyle* [Novak 2010], *SonarQube*, etc. Ces outils, en particulier *SonarQube* qui est libre, supporte une multitude de langages tels que Java, C#, etc. [Marcilio 2019]. Par ailleurs, il n'existe pas d'outils d'analyse statique de code libre pour certains langages de programmation comme PowerBuilder par exemple qui est le langage utilisé par la société CIM.

Ce qui est positif est que deux analyseurs syntaxiques de code PowerBuilder libre sont en cours de développement. Il s'agit de [Alex 2021] développé par Sashazjukov et [Moo 2021] développé par notre équipe dont nous avons beaucoup contribué. L'analyseur syntaxique de [Alex 2021] permet d'identifier les différents attributs et les méthodes d'une classe PowerBuilder. Mais il n'analyse pas le corps des méthodes et ne donne pas de métadonnée sur les positions des méthodes et attributs trouvés dans le code. Ceci peut poser un problème s'il faut l'utiliser pour détecter et corriger des violations de code par exemple.

L'analyseur syntaxique de [Moo 2021] analyse le code PowerBuilder mais n'analyse pas correctement les accès aux attributs. Cela peut poser une analyse de code qui a besoin des accès aux attributs. Mais il peut permettre de détecter des violations de code par exemple, car il analyse les instructions de code.

2.4 Problèmes liés à l'adoption des méthodes de *SCRUM*

L'adoption des modèles agiles pose quatre problèmes tels que (i) les problèmes organisationnels et de gestions ; (ii) les problèmes liés aux personnels ; (iii) les problèmes liés au processus ; et (iv) les problèmes liés aux outils [Nerur 2005, Almeida 2017].

Durant la modernisation de ses pratiques, la société CIM a dû faire face aux conditions exceptionnelles liées à la crise de la COVID-19 avec la généralisation du travail à distance, ce qui présente des problèmes pour la création et le partage des connaissances en agile, car les éditeurs logiciels agiles s'appuient généralement sur des équipes colocalisées [Mikalsen 2021a]. Ici, nous allons présenter les problèmes ci-dessus.

(i) Problèmes organisationnels et de gestion : Dans une société, la culture (les valeurs, les croyances, les normes) est construite et renforcée au fil du temps et se reflète dans la gestion organisationnelle de la société. En d'autres termes, la culture influence considérablement les prises de décisions, les façons de résoudre les problèmes, l'innovation, la communication, etc. dans la société [Nerur 2005]. Comme il est difficile de changer la culture ou la mentalité des gens, c'est d'autant plus difficile pour de nombreuses sociétés de passer au *SCRUM* [Gandomani 2013].

(ii) Problèmes liés aux personnels : Le succès du *SCRUM* repose sur la communication et la collaboration entre les collaborateurs d'une même équipe multicompétence constituée de programmeurs, des testeurs, etc. qui se valorisent et se font confiance mutuellement. Pour les programmeurs qui sont habitués à travailler en solitaire ou dans un groupe homogène, l'apprentissage partagé, les ateliers de réflexion, la programmation en binôme et la prise de décision en collaboration peuvent être difficiles.

De plus, la diversité de profil dans les équipes *SCRUM* peut créer une pluralité de propositions pendant les prises de décision, les rendant ainsi difficiles contrairement aux modèles de développement non agiles où les chefs d'équipes prennent les décisions.

Les méthodes agiles demandent l'utilisation de nouveaux outils que les développeurs ne maitrisent pas forcément. Par conséquent, il faut les former sur l'utilisation de ces outils.

(iii) Problèmes liés au processus : Les modèles de développement logiciel agiles sont caractérisés par leurs flexibilités par rapport aux exigences, la prise en compte de capacité de production et des compétences des collaborateurs, etc. Le changement d'un modèle de développement logiciel non agile et rigide englobant des activités standardisées vers les modèles agiles demande un investissement important en temps, en efforts et en capital [Nerur 2005].

Les modèles agiles sont basés sur les itérations de courte durée ainsi que des petites équipes, ce qui n'est pas toujours le cas dans les modèles non agiles. Il est difficile de déterminer la bonne composition et la taille de chaque équipe agile en fonction des compétences disponibles et la durée des itérations qui convient.

(iv) Problèmes liés aux outils : Les outils jouent un rôle essentiel dans la mise en œuvre réussie d'une méthodologie de développement logiciel. La technologie existante d'une organisation peut avoir un impact sur les efforts de migration vers les méthodologies agiles, car il faut des outils qui supportent et facilitent le développement itératif rapide, la gestion des versions/configurations, les tests, le *refactoring* et d'autres techniques agiles [Nerur 2005, Gandomani 2013].

(v) Problèmes liés à la COVID-19 : La COVID-19 a entrainé une généralisation du travail à distance. Selon [Dorairaj 2012] les équipes agiles sont des équipes interfonctionnelles qui favorisent le partage des connaissances spécifiques au projet à travers de fréquentes interactions en face à face, une communication efficace et la collaboration avec les clients. Ainsi, les principaux problèmes des sociétés agiles dont des membres d'équipe sont à distance sont dans quatre domaines : communication, coordination, collaboration et culture [Rizvi 2015]. La collaboration entre les membres d'équipes travaillant à distance est faite par une variété d'outils de communication digitaux se substituant aux interactions en face à face [Deshpande 2016, Mancl 2020]. Toutefois, les pratiques agiles sont plus difficiles à réaliser dans un contexte de réunions et des interactions virtuelles. D'après [Mancl 2020], les outils de communication ne sont pas toujours adaptés aux interactions telles que le *Stand-up meeting*, les conversations informelles. De la même façon, la formation de nouvelles équipes et l'intégration du personnel sont difficiles dans un environnement de travail virtuel.

2.5 Contexte de recherche selon Kitchenham

Pour mieux comprendre cette thèse, il est important d'expliquer le contexte. Présenter le contexte d'une recherche pour qu'il soit clair pour tous et que l'expérience soit productible relève d'un défi [Kitchenham 1999]. C'est pour cela que Kitchenham *et al.* ont identifié les facteurs qui influencent un projet de maintenance d'un produit logiciel. À cet effet, ils ont défini une ligne directrice à suivre pour la clarification de contexte de recherche. Elle identifie 4 points qui sont les suivants : le logiciel maintenu, les activités de maintenance, le processus de maintenance logicielle et les ressources humaines.

Dans cette section, nous allons présenter ces points afin de les utiliser pour clarifier le contexte de notre recherche par la suite.

2.5.1 Le logiciel maintenu

Le logiciel maintenu est défini sous plusieurs axes, à savoir : la taille du logiciel, le domaine d'application du logiciel, l'âge du logiciel, la maturité du logiciel, la composition du logiciel et finalement la qualité du logiciel ainsi que celle de ces artefacts.

La taille du logiciel : définit la complexité du logiciel en matière de nombre de lignes de code ou nombre de composants. Cette donnée est importante, car plus le logiciel est complexe, plus il faut une équipe de maintenance large.

Le domaine d'application du logiciel : indique le domaine pour lequel le logiciel est conçu. Par exemple : finance, télécommunication, etc. C'est important, car le domaine d'application peut contraindre la maintenance en matière de coût, fiabilité, etc.

L'âge du logiciel : représente l'âge depuis la première version du logiciel. Il est important, car si la technologie sous-jacente est vieille par exemple, il est difficile de trouver des ressources humaines pour la maintenance. De la même façon, si le logiciel est vieux, il est difficile d'avoir une connaissance globale du logiciel, car les développeurs originaux ne sont plus forcément là.

Maturité du logiciel : différent de l'âge du logiciel, le cycle de vie du logiciel après la première version. Ainsi on distingue 4 niveaux de maturité :

- L'enfance : c'est le stage après la première version quand les utilisateurs commencent à reporter des défaillances dans le logiciel
- L'adolescence : le nombre d'utilisateurs augmente et le nombre de défaillances identifiées aussi. Mais il peut encore y avoir des modifications visant à changer le comportement du système.
- L'âge adule : les utilisateurs sont nombreux plus qu'avant et le système est relativement stable. L'accumulation des modifications peut entraîner un besoin de restructuration du système.
- La sénilité : à ce stage, il y a de nouveaux logiciels sur le marché, et peu d'utilisateurs utilisent encore le logiciel. Les modifications sont souvent correctives ou des solutions de contournement.

La composition du logiciel : définit les différents éléments qui composent le logiciel. Comme un logiciel peut être composé du code généré ou de librairies tierces ou du code spécifique écrit par les développeurs ou encore un ensemble de logiciels standard, *COST (Component Off The Shelf)*; les compétences requises pour sa maintenance dépendent de sa composition.

La qualité du logiciel et ces artefacts : il s'agit de la qualité de la structure et du code source du logiciel, la documentation, etc. C'est important, car par exemple cela peut influer sur le coût de la maintenance.

2.5.2 Les activités de maintenance

En deuxième point, Kitchenham *et al.* ont abordé les activités de maintenance. Ces activités englobent les activités d'investigation, de modification, de gestion ainsi que d'assurance de qualité.

Activité d'investigation : c'est une activité qui évalue l'impact d'une modification lié à une demande ou un rapport de défaillance sur le logiciel existant.

Activité de modification : c'est l'ensemble des actions qui visent à modifier le comportement du logiciel existant.

Activité de gestion : c'est l'ensemble des actions qui visent à gérer le processus de maintenance d'un logiciel, ou bien à configurer son suivi.

Activité d'assurance de qualité : c'est une activité qui vérifie si les modifications ne créent pas de régression dans le système existant.

2.5.3 Procédé de maintenance logicielle

Il y a deux procédés de maintenance logicielle. Il y a le procédé de maintenance utilisé par les ingénieurs pour modifier le code et le procédé utilisé par l'entreprise pour gérer les demandes de maintenance.

Procédé des ingénieurs : résume l'ensemble des outils et techniques utilisées et choix faits par les ingénieurs lors du processus de maintenance du logiciel. Il est généralement composé de :

- Technologie de développement : c'est la technologie utilisée initialement quand le logiciel a été construit. Par exemple, le langage de programmation, le format des données, etc.
- Paradigme : la philosophie utilisée lors de la construction du logiciel maintenu. Par exemple, le paradigme procédural ou orienté objet.

Procédé de l'entreprise : c'est comment l'entreprise orchestre le processus de maintenance du stade de la réception de la demande de maintenance à la modification du code par l'ingénieur.

2.5.4 Ressources humaines

Le quatrième point de [Kitchenham 1999] se focalise sur les caractéristiques des ingénieurs responsables de la maintenance du logiciel. Ainsi on a :

L'attitude des ingénieurs : c'est la motivation des ingénieurs chargés de maintenir le logiciel.

La responsabilité des ingénieurs : permet d'expliquer si l'équipe responsable du développement est la même que celle responsable de la maintenance ou bien il y a une séparation d'équipe pour ces tâches-là.

Compétence des ingénieurs : vise à identifier les différents profiles de compétences dans l'équipe de maintenance.

2.6 Évaluation de processus de développement

Dans la perspective d'identifier les bénéfices de la modernisation des pratiques de développement à la société CIM ainsi que les améliorations possibles pour permettre à la société de mieux s'orienter sur les prochaines étapes de la modernisation, il est important d'évaluer les pratiques de développement avant et après les changements. Dans ce sens, nous étudions ici la littérature sur comment évaluer des pratiques de développements logiciels.

Pour étudier un phénomène en ingénierie logicielle, comme évaluer des développements dans le contexte de cette thèse par exemple, les méthodes de recherche empirique sont souvent utilisées [Sjoberg 2007]. La recherche empirique en ingénierie logicielle étudie les artefacts liés aux logiciels afin de les caractériser, les comprendre, les évaluer, les prédire, les contrôler, les gérer et les améliorer au moyen d'analyses qualitatives et quantitatives [Zhang 2018].

Ceci inclut la collecte des données qui sont soit quantitatives (données numériques) ou qualitatives (mots, descriptions, images, diagramme, etc.). La recherche empirique en ingénierie logicielle est généralement basée sur des données qualitatives, car elles fournissent une description plus riche et plus profonde. Toutefois, une combinaison de données qualitatives et quantitatives permet souvent de mieux comprendre le phénomène étudié [Runeson 2009]. Une liste exhaustive des méthodes de collection et d'analyse de données en ingénierie logicielle a été proposé par [Singer 2008].

2.6.1 Méthodes de collecte de données

Ici, nous allons présenter les méthodes de collecte de données de recherche empirique en ingénierie logicielle.

Remue-méninges et groupes ciblés : Dans un remue-méninges, plusieurs personnes se réunissent et se concentrent sur une question particulière. L'idée est de faire en sorte que la discussion ne se limite pas aux « bonnes » idées ou aux idées qui ont un sens immédiat, mais plutôt de découvrir autant d'idées que possible.

Les groupes ciblés sont similaires au remue-méninges. Cependant, les groupes ciblés sont constitués de personnes respectant les mêmes traits (ou caractéristiques) réunies pour se concentrer sur une question particulière (et non pour générer des idées).

Le remue-méninges et les groupes ciblés sont d'excellentes techniques de collecte de données à utiliser lorsque l'on est novice dans un domaine et que l'on cherche des idées pour une exploration plus approfondie [Singer 2008]. Par exemple, [Zafar 2018] a étudié pourquoi les bonnes pratiques de spécifi-

cation ne sont pas utilisées par les éditeurs de logiciels pakistanais au moyen du remue-méninges. [Dingsøyr 2013] a utilisé la méthode du groupe ciblé pour étudier les facteurs perçus par les développeurs agiles qui influencent l'efficacité du travail d'équipe. De la même façon, [Groeneveld 2021] a utilisé cette méthode pour comprendre le rôle de la créativité en ingénierie logicielle.

Le remue-méninges et groupes ciblés sont de bonnes techniques de collecte de donnée, mais il peut arriver que des participants à l'expérimentation soient timides et ne disent pas ce qu'ils pensent vraiment [Singer 2008]. Par exemple, un programmeur ne dirait pas forcément tout qu'il pense devant son chef d'équipe. Ce qui représente un désavantage.

Les entretiens : Les entretiens sont utilisés dans divers contextes. Ils servent à clarifier des choses qui se sont produites ou ont été dites pendant une observation, à obtenir des impressions sur la réunion ou un autre événement observé, ou à collecter des informations sur des événements pertinents qui n'ont pas été observés [Seaman 1999]. Il y a deux types d'entretiens : les entretiens structurés et les entretiens semi-structurés [Singer 2008].

Dans un entretien structuré, une liste fixe de questions rigoureusement formulées constitue la base de l'entretien. Les questions sont souvent posées exactement comme elles ont été écrites, et il n'y a aucun écart. Les données issues des entretiens structurés sont généralement analysées à l'aide d'analyses statistiques.

Dans le cas d'un entretien semi-structuré, l'entretien suit généralement le cours d'une conversation. De nouvelles questions peuvent être conçues quand de nouvelles informations sont apprises. En général, on pose des questions ouvertes qui permettent une plus grande interaction. En outre, dans certains entretiens semi-structurés, l'entretien sera structuré autour d'un cadre de sujets potentiels plutôt que de questions spécifiques. Les données issues des entretiens semi-structurés sont généralement analysées à l'aide de méthodes d'analyse qualitative. Les entretiens sont utilisés par [Khadka 2014] pour étudier comment les programmeurs perçoivent les logiciels patrimoniaux et la modernisation de logiciel. De même, [Johanssen 2018] a étudié le point de vue des développeurs sur l'intégration continue utilisant les entretiens. Les entretiens structurés sont un moyen efficace de collecter les mêmes données auprès d'un grand nombre de participants [Singer 2008]. De plus, pendant un entretien semi-structuré, le chercheur peut clarifier les questions en vue d'avoir des réponses pertinentes.

Par ailleurs, il faut planifier l'entretien avec le participant en tenant en compte la disponibilité du participant et du chercheur, ce qui peut être difficile si le chercheur ou le participant est chargé dans son emploi du temps. Si l'en-

tretien est enregistré en audio ou vidéo, il faut le retranscrire et analyser la retranscription.

Vu les avantages et les inconvénients de l'entretien, nous pouvons aussi l'utiliser pour collecter les données sur la perception des développeurs de la société CIM sur la modernisation des pratiques dans la société comme [Khadka 2014].

Sondage/ Questionnaires : Il s'agit d'un ensemble de questions présentées sous forme écrite auxquels les personnes représentatives du phénomène à étudier répondent. Ce sont des techniques de collecte de données les plus courantes, car elles peuvent être réalisées rapidement et facilement. Par exemple, [Yang 2020, Zhang 2013] a utilisé ces méthodes de collecte de données dans leur recherche.

Par ailleurs, contrairement à l'entretien, les questions ambiguës peuvent difficiles à répondre correctement, car le chercheur n'a pas la possibilité de mieux expliquer les questions en cas d'incompréhension. De plus, les participants peuvent ne pas avoir le temps de répondre au questionnaire, ce qui affecte le taux de réponse.

Modélisation conceptuelle : Pendant la modélisation conceptuelle, les participants créent un modèle d'un aspect de leur travail. Par exemple, on peut demander aux programmeurs de dessiner un diagramme de flux de données, un diagramme de flux de contrôle ou un diagramme de paquetage montrant les grappes architecturales importantes de leur système. Les résultats de la modélisation conceptuelle sont souvent difficiles à interpréter, surtout si le chercheur n'a pas de connaissance du domaine de modélisation [Singer 2008]. Les développeurs de la société CIM sont de plusieurs domaines que nous allons détailler dans le Chapitre 3. Alors s'il est demandé aux participants de répondre en créant un modèle, l'interprétation serait difficile.

Pensée à voix haute : Comme son nom l'indique, les chercheurs demandent aux participants de penser à voix haute tout en effectuant une tâche. Cette tâche peut se produire naturellement au travail ou être prédéterminée par le chercheur. Comme les programmeurs oublient parfois de verbaliser, les chercheurs peuvent leur rappeler occasionnellement de continuer à penser à voix haute. En général, utiliser pour étudier les différences individuelles dans l'exécution d'une même tâche donnée [Charters 2003].

Les carnets de travail : Les chercheurs demandent aux participants d'enregistrer divers événements qui se produisent au cours de la journée. Il peut s'agir de remplir un formulaire à la fin de la journée, d'enregistrer des activités spécifiques au fur et à mesure qu'elles se produisent, ou de noter la tâche courante à un moment présélectionné. Les carnets de travail peuvent fournir de meilleures données d'étude comparer aux entretiens parce qu'ils enregistrent

les événements sur une base continue plutôt que rétrospectivement comme les entretiens. Un problème des carnets de travail est qu'ils peuvent gêner les participants dans leur travail, car ils n'ont pas forcément envie de noter certains événements de leurs journées de travail. De plus, les participants peuvent oublier de noter des événements.

Suivi/Observation : Pendant le suivi, le chercheur suit le participant et enregistre ses activités. Le suivi peut être pratiqué pendant une période illimitée, tant que le participant est consentant. Dans le cas de l'observation, le chercheur observe des programmeurs dans leur travail ou dans des tâches spécifiques liées à l'expérience, comme des réunions ou la programmation. Contrairement au suivi où le chercheur ne peut que suivre un participant à la fois, pendant l'observation, il peut observer plusieurs participants.

Participant-Observateur : Le chercheur devient essentiellement un membre de l'équipe et participe à des activités clés. Le fait de participer au processus de développement de logiciels permet au chercheur d'acquérir un haut niveau de familiarité avec les membres de l'équipe et les tâches qu'ils accomplissent. Par conséquent, les participants sont plus susceptibles d'être à l'aise avec un membre de leur équipe et d'agir naturellement pendant la collecte des données de recherche. Cette thèse en collaboration avec la société CIM. Il est convenu nous soyons dans la société une demi-journée par jour. Cela nous a permis aux développeurs de la société de la société de se familiariser avec nous. Donc, nous avons utilisé cette approche de collecte de donnée dans cette thèse.

Analyse des bases de données des travaux effectués : Dans la plupart des éditeurs de logiciels, le travail effectué par les développeurs est systématiquement géré à l'aide de systèmes de suivi des anomalies, de signalement des défauts, de demande de modification et de gestion de code source. Ces systèmes exigent que les programmeurs saisissent des données telles que la description d'un problème rencontré ou un commentaire lors de la vérification d'un module de code source. Les nombreux enregistrements générés par ces systèmes constituent une riche source d'informations pour les chercheurs. Par exemple, [Zhang 2010] utilise les données des occurrences de défauts et le temps pour modéliser l'évolution d'un logiciel avec $ccharts$. De la même façon, [Lenarduzzi 2017] propose un système de recommandation d'action au développeur pour un nouveau défaut en se basant sur l'historique des défauts, le code source ainsi qu'un algorithme de prédiction.

Analyse des *logs* d'outils : De nombreux logiciels utilisés par les programmeurs génèrent des *logs*. Par exemple, les outils de construction automatique laissent souvent des enregistrements, tout comme les systèmes de contrôle de version. Ces *logs* d'outils peuvent être analysés par le chercheur.

Analyse de la documentation : Cette technique se concentre sur la documentation générée par les programmeurs, y compris les commentaires dans le code d'un logiciel, ainsi que des documents séparés décrivant le logiciel. D'autres sources de documentation peuvent être analysées, notamment les groupes de discussion locaux, les listes de courriel, les mémos et les documents qui définissent le processus de développement.

Analyse statique et dynamique d'un système : Dans cette technique, on analyse le code (analyse statique) ou les traces générées par l'exécution du code (analyse dynamique) au moyen des critères d'analyse prédéfinie.

2.6.2 Analyse des données

La collecte des données à travers l'une ou la combinaison des méthodes présentées dans la Section 2.6.1 produit souvent une grande quantité de données. Ces données peuvent être sous forme écrite, audio ou vidéo, etc. Pour faciliter l'analyse, les données vidéo ou audio doivent être préalablement transcrites sous forme de texte. Les données quantitatives peuvent être analysées statistiquement. Par contre, il existe plus de 40 méthodes pour analyser les données qualitatives [Kuckartz 2019] en particulier la méthodologie de *Grounded Theory*.

La *Grounded Theory* est une méthode de recherche exploratoire qui vise à découvrir de nouvelles perspectives et idées, plutôt qu'à confirmer celles qui existent déjà. Elle consiste à un travail itératif au bout duquel le chercheur simplifie les données tout en restant le plus proche possible des idées des données initiales. Cette simplification commence par l'« Étiquetage » : un processus qui consiste à diviser les données en unités cohérentes plus petites et à ajouter des étiquettes à ces unités. L'objectif des étiquettes est de nommer des idées qui se dégagent des données. Ainsi, le chercheur peut créer une étiquette pour une donnée ou rajouter la donnée à des étiquettes qui existe déjà. Au fur et à mesure du processus d'étiquetage, le chercheur peut écrire des notes expliquant l'étiquette appelées « Mémos ». Le chercheur répète ce processus jusqu'à la saturation (il n'y a plus de nouvelles idées à dégager des données). Ces étiquettes représentent les caractéristiques clés des données. Par la suite, les étiquettes sont organisées en concepts, qui à leur tours ont regroupés en catégories. Durant cette étape de regroupement en catégorie, les catégories centrales, c'est-à-dire les catégories qui capturent la plus grande variation dans les données et répondent à la principale préoccupation des participants de l'étude, se dessinent [Adolph 2011].

La *Grounded Theory* a été utilisée par [Khadka 2014] pour analyser les données d'entretiens. Pour faciliter cette analyse, ils ont utilisé l'outil *Nvivo 10* [1].

La *Grounded Theory* a permit à [Bragagnolo 2021] d'avoir un esprit ouvert,

1. www.qsrinternational.com/

en réduisant les biais et en laissant les connaissances émerger du texte, plutôt que définir des réponses à des questions strictes préexistantes (ce qui implique un biais sur la façon de lire et d'interpréter le contenu des papiers). Ainsi, pour l'analyse de ses données de revue d'une revue de littérature sur la migration logicielle, [Bragagnolo 2021] a utilisé la *Grounded Theory* avec *MAXQDA2020* [2].

2.7 Résumé du chapitre

Dans ce chapitre, nous avons étudié l'état de l'art sur les modèles de développement logiciels, comment clarifier un contexte de recherche et comment évaluer un processus de développement. Ainsi nous avons vu d'après la littérature que les modèles de développement agiles présente l'avantage tolérant au changement contrairement aux autres modèles. En plus de permettre de livrer les clients à temps, les modèles agiles permettent d'avoir une meilleure qualité de logiciel. Par contre, la COVID-19 par le travail à distance peut impacter sur les bénéfices des modèles agiles et *SCRUM* en l'occurrence. Nous avons vu que pour assurer la qualité dans les modèles agiles notamment *SCRUM*, il faut des tests automatisés et des outils d'analyse statiques de code. Mais ces outils n'existent pas sur tous les langages de programmation, notamment PowerBuilder.

Nous avons aussi vu comment clarifier un contexte de recherche. Ceci nous permettra de décrire le contexte de cette thèse dans le chapitre suivant.

Enfin, nous avons vu comment les méthodes d'évaluation de processus de développement logiciel que nous allons utiliser dans les Chapitre 4 et Chapitre 5 pour évaluer l'impact des changements de la société CIM.

2. https ://www.maxqda.com/

CHAPITRE 3
Description du contexte

Sommaire

3.1 Introduction 23
3.2 Le langage PowerBuilder 24
3.3 Présentation du contexte de la société CIM 27
3.4 Résumé du chapitre 32

3.1 Introduction

Les sociétés d'édition de logicielle sont confrontées à la nécessité de s'adapter à un environnement commercial complexe, en constante évolution et transformation. Dans ces circonstances, leur agilité est la clé de voûte du maintien de leur place sur le marché. Comme vu au chapitre précédant, les anciennes méthodes de développement atteignent leurs limites. Cela fait que de nombreuses sociétés d'édition de logiciel ont du mal à honorer les délais de livraison des clients. Il en résulte la frustration des clients.

Dans ces sociétés, le retard et la frustration des clients entraînent des urgences dans les différentes équipes du cycle de développement. Ainsi, les sociétés d'édition de logiciel ont souvent peu de ressources disponibles à allouer à l'amélioration de la qualité des logiciels ou à l'élimination des défauts (*bugs*) du code. Au fur et à mesure que leur logiciel évolue, il devient de plus en plus difficile à maintenir.

Pour assurer leurs futurs, de nombreuses sociétés changent leurs pratiques de développement pour adopter des pratiques plus adaptées au marché. Ceci crée une ruée des sociétés vers les méthodes *SCRUM*. C'est le cas de la société CIM qui, afin de remédier aux problèmes de sa prestation auprès de ses clients et améliorer la qualité de son logiciel principal écrit en PowerBuilder, et ainsi rester compétitive sur le marché, a entreprise des actions de modernisation de ces pratiques de développement.

Durant cette thèse, nous avons accompagné la société CIM durant ces changements sur son logiciel principal. Dans ce chapitre, nous allons décrire la société CIM ainsi que son processus de développement logiciel qui représente notre contexte de recherche.

3.2 Le langage PowerBuilder

Comme introduit dans le Chapitre 1 à la page 1, le logiciel principal de la société CIM, Izy Protect, est écrit en PowerBuilder. Cela fait que nous allons utiliser PowerBuilder dans la suite. Bien que PowerBuilder ait été populaire dans les années 90, il est devenu moins populaire de nos jours. Cela fait que les différents concepts du langage sont moins connus. De plus, PowerBuilder présente des caractéristiques particulières (gestion du code) qui le rendent difficile à intégrer avec des outils modernes de génie logiciel. Ici, nous allons présenter l'environnement et langage de programmation.

PowerBuilder est un langage de programmation et un environnement de développement intégré initialement développé par PowerSoft en 1991. Il permet de développer des projets client-serveur et bureautique sur Windows. C'est un langage de 4e génération. Les langages de 4e génération sont conçus pour réduire l'effort de programmation, le temps nécessaire pour développer d'un logiciel et le coût de ce dernier. Ainsi, au lieu d'utiliser du code, le programmeur définit sa logique en sélectionnant une opération dans une liste prédéfinie de commandes de manipulation de la mémoire ou des tables de données. PowerBuilder génère le code du projet défini par le programmeur graphiquement. Ce code peut être difficile à comprendre et maintenir.

La communauté et les ressources de maintenance disponibles sont très limitées [MOB 2018]. La version 2019 de l'environnement de développement intégré de PowerBuilder propose un module d'aide à la migration de PowerBuilder vers *C#*.

Dans la continuité de cette présentation, nous allons présenter l'environnement de développement intégré de PowerBuilder, les fichiers qui composent un projet PowerBuilder, quelques caractéristiques orientés objets (OO), l'outillage fournit par le langage pour la gestion du code et l'exécution d'un projet PowerBuilder.

Environnement de développement intégré de PowerBuilder : L'environnement de développement intégré de PowerBuilder permet de créer, développer, exécuter et déployer les projets PowerBuilder. Cet environnement permet au programmeur de créer des classes d'un projet PowerBuilder graphiquement. La Figure 3.1 montre un aperçu de l'environnement de développement avec un projet ouvert. Comme elle le montre, l'environnement de développement de PowerBuilder comporte 3 parties principales. La partie (1) permet au programmeur de voir la structure du projet PowerBuilder et d'ouvrir les classes pour les éditer. La partie (2) est une boite à outils qui permet au programmeur de créer de nouvelles classes ou d'ajouter des composants graphiques à un projet PowerBuilder. Elle permet aussi d'exécuter ou de déboguer le projet. La partie (3) permet d'afficher les différentes informations de sortie (*log*) relatives au projet. L'environnement offre un éditeur qui permet au programmeur d'écrire du code fonctionnel appelé *powerscript*. Pour ouvrir un pro-

FIGURE 3.1: Environnement de développement intégré de PowerBuilder

jet PowerBuilder dans l'environnement de développement, il faut nécessairement avoir les librairies PowerBuilder du projet.

Fichiers : Un projet PowerBuilder est caractérisé par un ensemble de fichiers : des fichiers binaires d'extension *.PBL*, des ressources(images, icônes, etc.), un fichier d'extension *.PBT* et un fichier d'extension *.PBW*. Les fichiers binaires d'extension *.PBL* désignent les bibliothèques PowerBuilder. Ils ont le rôle de regrouper les différentes classes du projet. Le fichier d'extension *.PBT* désigne la *Target* du projet PowerBuilder. C'est un fichier texte qui contient la liste des bibliothèques du projet PowerBuilder. Le fichier d'extension *.PBW* désigne l'espace de travail du projet. Ce fichier sert à ouvrir un projet PowerBuilder dans l'environnement de développement intégré de PowerBuilder. Il existe aussi les fichiers binaires d'extension *.PBD* (bibliothèques dynamiques PowerBuilder), et le fichier *.EXE* (exécutable Windows) qui sont obtenus après déploiement d'un projet PowerBuilder.

PowerBuilder/OO : D'après la documentation officielle de PowerBuilder [pbd 2011], les différentes catégories de classe qu'un projet PowerBuilder peut contenir sont : *Window*, *Application*, *DataWindow*, *Menu*, *Global function* (fonction), *Query*, *Structure*, *User object*, *Pipeline*, *Project*.

— Les classes *Windows* sont des classes qui représentent les fenêtres de l'interface graphique.

— Les classes *Applications* sont des classes qui définissent les éléments qui ont une visibilité globale dans un projet PowerBuilder, tel que la déclaration des variables globales, la déclaration des fonctions ainsi que la logique du projet à l'ouverture et à la fermeture.

— Les classes *DataWindows* sont des classes qui permettent de récupérer et manipuler des données à partir d'une base de données relationnelle ou d'une autre source de données. Elles peuvent se présenter sous-forme de tableau de données, de formulaire, de graphe, etc.

— Les classes *Menus* représentent la liste des commandes ou des options qu'un utilisateur peut sélectionner dans une fenêtre active.

— Les classes *Global functions* sont des fonctions. Elles ont une visibilité globale et servent à effectuer des traitements généraux dans un projet PowerBuilder.

— Les classes *Queries* sont des requêtes SQL.

— Les classes *Structure* sont des classes qui représentent un ensemble d'une ou plusieurs variables reliées entre elles et regroupées sous un seul nom.

— Les classes *User objects* sont des modules de traitement réutilisable ou ensemble de contrôles.

— Les classes *Pipelines* permettent de reproduire les données dans une base de données ou entre les bases de données.

— Les classes *Projets* permettent de packager un projet PowerBuilder pour la distribuer aux utilisateurs.

Toutes ces classes peuvent contenir des méthodes sauf les classes *Structure*, *Datawindow* et *Query* qui ne peuvent ni contenir de méthodes ni être sous-classées.

En PowerBuilder on distingue plusieurs catégories de méthodes. Les *évènements* PowerBuilder qui sont des méthodes à exécuter quand un évènement se produit (ce sont des *events handlers*), dans ce document elles sont appelées des *méthodes évènementielles*. Nous avons les méthodes avec et sans valeurs de retour appelées respectivement *fonctions* et *sous-routines* en PowerBuilder. Nous allons garder l'appellation *méthode* pour ces deux catégories.

Outillage pour la gestion de code source : PowerBuilder impose de stocker le code source dans un format propriétaire difficilement manipulable par des outils externes.

Avant la version 2017 de l'environnement de développement intégré de PowerBuilder, ce denier n'intégrait pas directement de système de contrôle de version. En effet, il fallait lui rajouter un module externe. Sa version 2017 intègre des systèmes de contrôle de version. Malgré cela, il y a des dysfonctionnements. Par exemple, l'environnement de développement intégré retient le chemin des projets versionnés de telle sorte que si l'on crée un nouveau projet du même nom qu'un projet versionné antérieurement avec l'environnement de développement intégré, il considère que le nouveau projet est versionné même si ce n'est pas le cas. La version 2019 est en partie concentrée sur l'amélioration de ces problèmes.

Exécution d'un projet PowerBuilder Un projet PowerBuilder peut être exécuté directement depuis l'environnement de développement intégré de PowerBuilder dans ce cas les bibliothèques (fichiers d'extension *.PBL*) sont utilisés. Ce mode est utile pour déboguer le projet en développement par les programmeurs. De la même manière, un projet PowerBuilder peut être exécuté avec l'exécutable *.EXE* et ses bibliothèques dynamiques (fichiers d'extension *.PBD*). Ce mode est normalement utilisé par les utilisateurs finaux ou testeurs pour lancer un projet PowerBuilder en invite de commande pour des tests fonctionnels.

3.3 Présentation du contexte de la société CIM

Ici, nous allons présenter le contexte de cette thèse sur la base des recommandations vues au chapitre précédant à la page 16.

3.3.1 Le logiciel maintenu

Cette section présente le logiciel principal de la société CIM.

La taille du logiciel : Izy Protect est un logiciel de plus de 3 MLOC [1]. Izy Protect compte 117 bibliothèques. La plus large a une taille d'environ 300 000 lignes de code. Il est constitué de 117 librairies PowerBuilder. Il contient près de dix mille objets PowerBuilder.

Le domaine d'application du logiciel : Le logiciel est un logiciel de gestion de l'assurance de personnes en Santé et Prévoyance.

L'âge du logiciel : Izy Protect est un système de plus de 20 ans écrit et maintenu dans un vieux langage de programmation.

La maturité du logiciel : Izy Protect est en phase adulte. C'est-à-dire que le nombre d'utilisateurs a augmenté. Il y a beaucoup de défauts ainsi que d'évolution. Par contre, depuis quelques années, la société a du mal à faire évoluer Izy Protect ou résoudre les défauts. Ceci motive la société à moderniser sa méthode de développement logiciel afin d'éviter qu'Izy Protect passe à la phase de sénilité.

La composition du logiciel : La société CIM a développé Izy Protect suivant une architecture propriétaire. Les classes sont constituées de code source généré par PowerBuilder et du code écrit par les programmeurs. En fonction du métier, les bibliothèques d'Izy Protect peuvent être regroupées en modules. Ainsi, Izy Protect comprend 13 modules.

La qualité du logiciel et ces artefacts : Les régressions sont courantes sur Izy Protect. La couverture des tests n'est pas assez pour détecter ces régressions. Il y a beaucoup de classes patrimoniales qui prennent plus de responsabilités qu'ils ne devraient. De surcroît, il n'y a pas de séparation entre la vue et la logique métier. Il n'est pas rare de rencontrer du code mort ou soit des méthodes avec beaucoup de paramètres qui ne sont parfois pas tous utilisés. Ces méthodes ont bien souvent une forte complexité cyclomatique, ce qui entache leurs compréhensions.

3.3.2 Les activités de maintenance

Cette section décrit les activités de maintenance effectuée par la société CIM sur le logiciel.

Activité d'investigation : Les différentes investigations mises en place par la société sont les analyses de son équipe de tests fonctionnels, l'équipe de développement ainsi que celui des responsables clients en vue de la détection des défauts.

1. million de lignes de code

Activité de modification : Les principales activités de modification faites sur le logiciel de la société CIM sont : la correction des défauts ou bien les développements de nouvelles fonctionnalités. Toutefois, il faut noter qu'en 2019, la société avait en projet de migrer le système, mais ce projet a été reporté pour des raisons de priorités de la société.

Activité de gestion : Les activités de maintenance ou de développement faites par la société sur Izy Protect sont gérées avec des tickets. Les tickets sont stockés dans la base de données des tickets ou « fiches navettes depuis 2000 ». La base de données des tickets pilote l'ensemble du processus d'évolution du logiciel : attribution du travail aux développeurs, gestion du flux de travail pour répondre à une demande du client, informations de facturation sur chaque tâche. Il existe des tickets pour la correction de défauts, la rédaction de documentation, l'ajout de nouvelles fonctionnalités, etc.

Un ticket comporte entre autres les caractéristiques suivantes :

- La date de création.
- La date de clôture.
- L'estimation du temps nécessaire au(x) programmeur(s) pour travailler sur le ticket.
- Le temps passé.
  - Le temps d'analyser.
  - Le temps de développement.
  - Le temps de test.
- La ou les bibliothèque(s) impactée(s).

Activité d'assurance de qualité : Dans la société, les activités d'assurance de qualité Izy Protect peuvent être scindées en deux parties : (1) les activités de revue de code et (2) les activités de tests.

- Des réunions de revue de code en équipe (dans les équipes de développement) sont organisées périodiquement pour promouvoir une meilleure qualité du code. Elles ont lieu toutes les deux semaines et durent environ une heure. On demande aux programmeurs de préparer des extraits de code dénonçant de mauvaises pratiques et des extraits de code qui montre les bonnes pratiques de programmation. Ces exemples sont ensuite discutés afin de garantir l'homogénéité du codage.

  Les sujets abordés dépendent beaucoup de ce que les programmeurs ont trouvé et il y a parfois très peu de choses à discuter lors d'une réunion, soit par manque de temps, soit parce que les programmeurs ne sont pas sûrs de ce qui constituerait une pratique « nouvelle et intéressante » pour leurs collaborateurs.

Ces réunions sont souvent l'occasion de rappeler à tous les programmeurs certaines règles de programmation de base de la société : conventions de dénomination des variables, utilisation d'une constante spéciale « infini », etc. Cependant, de nombreuses violations de ces règles peuvent être trouvées dans le code.

Les programmeurs sont censés les corriger lorsqu'ils trouvent de telles violations, mais cela est rarement fait par manque de temps.

Une autre action de promotion de la qualité du code est mise en place pour les jeunes programmeurs : *revue de code individuelle*. Elle est systématique lorsqu'un nouveau programmeur rejoint l'équipe, ou lorsqu'un ancien programmeur travaille pour la première fois sur un nouveau module du logiciel.

— Les tests sont effectués manuellement par chaque programmeur sur le code qui vient d'être développé. Il faut noter qu'aucun outil ou pratique de test unitaire n'est mis en place.

Certaines parties du système sont identifiées comme non stables, sur la base de l'expérience de la société. Si le code développé est lié à une partie non stable du système, des tests de non-régression sont exécutés sur le système. Ces tests de non-régression comprennent (i) des appels en ligne de commande des fonctionnalités de base suivie d'une comparaison des résultats de l'exécution du système avec les données de référence ; (ii) des simulations des interactions utilisateur avec l'interface d'Izy Protect, en utilisant l'outil de tests automatisé TOSCA [2].

3.3.3 Procédé de maintenance logiciel

Cette section décrit le procédé de maintenance logiciel de la société CIM.

Procédé de maintenance des ingénieurs : Les programmeurs font les modifications en PowerBuilder et en SQL pour des requêtes sur les données. Le paradigme de programmation est fonctionnel et orienté objet. C'est orienté objet dans le sens où les programmeurs créent des objets et les manipulent. C'est fonctionnel dans le sens où les programmeurs peuvent créer des fonctions globales. Ceci est beaucoup utilisé dans le code source du logiciel.

Procédé de maintenance logicielle de la société CIM : Le procédé de maintenance de la société englobe toutes les pratiques de génie logiciel que la société applique pour la production de son logiciel. Ces pratiques sont :

Stand-up meeting La société utilise une sorte de *Stand-up meeting* pour l'équipe de programmation inspirée des méthodes agiles. Ces réunions ont

2. https ://www.tricentis.com/products/automate-continuous-testing-tosca/

lieu deux fois par semaine (le mardi matin et le jeudi matin). Chaque programmeur explique à tour de rôle ce qu'il a fait la veille, les problèmes rencontrés et ce qu'il prévoit de faire le lendemain. Ces réunions permettent aux programmeurs de demander de l'aide à leurs collaborateurs. Mais d'un autre côté, cette aide peut rendre les réunions très longues, car un problème particulier est discuté par deux collaborateurs, laissant les autres attendre.

Gestion du code source En ce qui concerne la gestion du code source, elle est manuelle. La société CIM versionne Izy Protect en versions correctives, versions majeures et quelques versions spécifiques au client. Toutes les versions sont archivées dans des répertoires sur l'intranet de l'entreprise. Il y a un peu plus de 1000 versions. Les versions antérieures à 2012 sont complètement perdues. Cette pratique a entraîné plusieurs problèmes :

- Les programmeurs s'appuient sur une communication informelle pour éviter de travailler en même temps sur les mêmes parties du code. De peur de perdre du code.
- L'intégration des modifications est entièrement manuelle, souvent en utilisant un outil générique de comparaison de fichiers comme WinMerge.
- Les modifications sont intégrées dans la base de code commune en copiant manuellement le code de la version du programmeur dans la version de la base de code commune.
- Lorsqu'ils effectuent un changement sur plusieurs parties du logiciel, les programmeurs ont besoin de garder une liste de toutes les parties impactées pour les porter dans la base de code commune un par un par la suite. Oublier une partie signifie que le changement ne sera pas complètement porté dans la base de code commune et que la version arrêtée ne sera pas correcte.
- Après une modification du code source, le programmeur doit marquer la modification avec une « propriété » spéciale dans PowerBuilder. S'il ne le fait pas, sa modification peut se perdre.
- Il est usuel de mettre un commentaire dans le code pour décrire la modification, la date et le programmeur qui l'a effectuée. Cela peut aider à déboguer le code plus tard.

Cycle de développement logiciel Dans la société, les activités sur Izy Protect sont organisées en cycles d'environ trois mois. Au début du cycle, un ensemble de tickets est sélectionné pour être résolu en fonction de leurs priorités et leurs anciennetés. Les tickets sont assignés aux programmeurs disponibles. À la fin de chaque cycle, une version majeure est publiée. Au cours

du cycle, certains tickets urgents (généralement des défauts, mais aussi de petites évolutions de moins de trois jours de travail) apparaissent, nécessitant qu'un programmeur mette de côté son travail pour les résoudre.

3.3.4 Ressources humaines

Cette section décrit les caractéristiques des internes de la société CIM engagés dans le projet de maintenance de la société.

L'attitude des ingénieurs : L'attitude des ingénieurs est considérée comme correcte.

La responsabilité des ingénieurs : Dans la société CIM, il n'y a pas de séparation entre l'équipe de maintenance et l'équipe de développement.

Les personnes impliquées dans la maintenance d'Izy Protect sont organisées en 4 équipes à savoir : l'équipe chargée de la clientèle, l'équipe d'analyse, l'équipe de programmation et l'équipe de test.

L'*équipe chargée de la clientèle* est en contact direct avec les clients, elle reçoit des retours des défauts et des demandes d'amélioration de leur part. Elle formalise ces demandes sous forme de tâches appelées « tickets » dans la société. Elle donne la priorité aux tickets et suit leur développement. Une fois qu'un ticket est fermé, l'équipe assure la livraison du produit.

L'équipe d'analyse compte 4 analystes et est appelée à analyser les tickets ainsi que rédiger des spécifications (fonctionnelles ou non fonctionnelles). La spécification résultante est formalisée dans un document joint au ticket.

L'équipe de programmation comporte 15 programmeurs. Elle est responsable de l'implémentation des tickets dans Izy Protect.

Enfin, *l'équipe de test* effectue des tests fonctionnels et non fonctionnels sur Izy Protect. Elle comprend 8 testeurs.

Compétence des ingénieurs : Les ingénieurs sont composés des personnes expérimentées ainsi que de débutants. Le plus expérimenté a 22 ans d'expérience en PowerBuilder. Quant au moins expérimenté, il a 1 an d'expérience.

3.4 Résumé du chapitre

Dans ce chapitre, nous avons abordé la description du contexte de cette thèse en parlant de la société *CIM* d'Izy Protect.

Nous avons vu que le logiciel phare de la société (Izy Protect) souffre des problèmes traditionnels des vieux systèmes, notamment la diminution de la qualité et la difficulté d'évolution. Nous avons aussi vu que la technologie utilisée est propriétaire et ne facilite pas les interactions avec les outils modernes de génie logiciel.

Nous avons décrit les activités de développement dans la société en soulignant certains problèmes.

La société souhaite remédier à ces problèmes, notamment en rénovant ses pratiques de développement. Dans le chapitre suivant, nous présentons une évaluation rigoureuse de la situation qui nous permet de vérifier par la suite les effets concrets de ces changements de pratiques.

CHAPITRE 4

Situation avant les changements de pratiques

Sommaire

4.1 Introduction 35
4.2 Etude quantitative 36
4.3 Étude qualitative du processus de développement dans la société CIM 45
4.4 Résumé du chapitre 53

4.1 Introduction

L'un des facteurs de réussite des éditeurs de logiciels est leur capacité à fournir rapidement des produits de bonne qualité. Pour cela, ils doivent améliorer leurs pratiques de développement logiciel. C'est le cas de la société CIM avec laquelle nous travaillons. La société modernise son processus de développement logiciel en se basant sur les modèles agiles. Même si les avantages des modèles agiles sont bien documentés, l'impact de ces changements sur les développeurs est moins bien connu.

Pour évaluer cet impact, il faut mesurer l'état des pratiques de développement avant et après les changements. D'évidence, il est nécessaire de trouver des indicateurs mesurables. Nous avons fait deux types d'évaluations : quantitative et qualitative. Avant les changements des pratiques de développement, nous avons analysé les tickets de la base de tickets de la société CIM (présentée dans la Section 3.3.2, page 28) pour proposer des indicateurs quantitatifs du processus de développement de la société CIM.

Pendant les changements dans la société, nous avons fait des entrevues avec les développeurs de la société CIM, pour avoir des points de vue qualitatifs sur les pratiques de la société.

Dans la Section 4.2 de ce chapitre, nous allons aborder les mesures quantitatives des pratiques de développement de la société CIM avant les changements.

Ensuite, nous allons aborder l'étude qualitative des pratiques de la société avant les changements dans la Section 4.3. Enfin, nous allons conclure le chapitre dans la Section 4.4.

4.2 Etude quantitative

La société CIM dispose d'une base de tickets, présentée dans le Chapitre 3, où elle enregistre ses activités de développement depuis 2000. Pour mesurer l'état du processus de développement, nous proposons d'analyser les données résultant de la mise en exécution des différentes activités régies par ce dernier. Dans le contexte de la société CIM ces données sont l'historique des tickets. Ici, nous allons analyser ces données historiques pour proposer des mesures quantitatives de l'état du processus de développement de la société.

4.2.1 Collecte des données

Le logiciel principal de la société CIM est Izy Protect, un logiciel vieux de plus de 20 ans. En d'autres termes, Izy Protect témoigne du résultat des pratiques de développement dans la société CIM durant toutes ces années. Par conséquent, les données relatives aux activités sur Izy Protect peuvent aider à comprendre et évaluer les pratiques de développement de la société CIM. Pour cela, nous avons analysé l'historique des données liées aux différentes tâches effectuées sur Izy Protect.

D'après la description présentée à la page dans la Section 3.3.1(28), Izy Protect est un logiciel de plus de 3 millions de lignes de code et maintenu sur plus de 20 ans par la société CIM. Les programmeurs à l'origine de l'application ne sont plus présents, de sorte qu'une grande partie des connaissances (et ce à différents niveaux de granularité) est dispersée ou perdue. Pour ces raisons, les connaissances actuelles des programmeurs sur Izy Protect ne sont pas complètement fiables. Les seules informations utilisables sont : son code source et une base de tickets.

Dans le cadre de l'évaluation et du suivi de processus de maintenance logiciel, des travaux de la littérature ont lié l'historique du code source et des défauts(*bugs*) pour proposer des indicateurs de mesure [Lenarduzzi 2017, Port 2017]. Ces indicateurs se basent sur du code versionné ou les anciennes versions du code sont disponibles. Par contre, le code source d'Izy Protect n'est pas versionné et certaines versions antérieures d'Izy Protect sont perdues. Par conséquent, nous ne pouvons pas utiliser l'historique du code source. Nous avons analysé la base de tickets uniquement.

4.2.2 Nettoyage des données

Les tickets peuvent concerner plusieurs logiciels de la société CIM ou simplement des demandes de configuration interne dans la société et différentes équipes. La base de tickets est mise à jour depuis 2000 jusqu'aujourd'hui. Un ticket est caractérisé par (Chapitre 3) sa date de création ; la (les) librairie(s) à laquelle il est associé ; la durée pour le traiter ; sa catégorie (évolution ou défaut), etc. Toutefois, les données ne sont pas toujours cohérentes. Ici, nous avons présenté les problèmes identifiés et les solutions que nous avons adoptées.

Problèmes

Nous avons identifié quatre problèmes : les informations manquantes, les fautes de frappe, le module métier des tickets, la catégorisation en défaut ou évolution.

Les tickets créés avant 2004 sont aux nombres de 20 sur 98000 tickets au total et les informations de ceux-ci ne sont pas toujours cohérentes (date de création ou catégorie manquante/incorrecte). Il existe aussi dans la base de tickets, des tickets qui n'avaient pas de date de création après 2004.

Les tickets sont souvent enregistrés avec des fautes de frappe. Par exemple, les durées sont enregistrées dans un format de texte libre avec certaines conventions. Comme on peut s'y attendre, les conventions ne sont pas toujours respectées (par exemple « 5d » ou « 5days »). De plus, jusqu'à récemment, le nom de la bibliothèque concernée par un ticket est rempli manuellement (texte libre). Par conséquent, il y a quelques fautes de frappe dans les noms qui doivent être corrigées. Un cas simple et fréquent est celui de deux caractères intervertis : *cwm_liq_ou* au lieu de *cwm_liq_uo*(« *uo* » pour *User object*). Certains cas plus complexes nécessitent une compréhension du domaine d'application et du système, par exemple, *uo_liq* au lieu de *cwm_liq_uo*. Ces fautes peuvent être nombreuses : *cwm_liq_uo* est écrit de 108 façons différentes par exemple.

Izy Protect comprend 13 modules (Chapitre 3). Par contre, le module qu'un ticket touche n'est pas renseigné dans le ticket.

La catégorie d'un ticket est fine alors que nous nous intéressons uniquement aux tickets d'évolutions et de défauts.

Solutions

Nous avons commencé le nettoyage des données en éliminant les tickets créés avant 2004 ainsi que les tickets qui n'avaient pas de date de création.

La correction du nom de la bibliothèque concernée par les tickets est plus difficile compte tenu du nombre de variations qu'on peut identifier pour le nom d'une bibliothèque. Pour cela, nous avons réalisé une liste des noms des bibliothèques d'Izy Protect. Ensuite, nous avons fait une analyse manuelle sur tous les tickets

pour créer un dictionnaire qui associe les noms des bibliothèques identifiés sur les tickets au nom de bibliothèque correcte. Pour assurer la répétabilité du nettoyage, nous avons créé un correcteur qui corrige automatiquement les noms des bibliothèques en se basant sur le dictionnaire de nom créé précédemment. Ensuite, nous avons sélectionné les tickets relatifs à Izy Protect : les tickets, dont les noms de bibliothèques, sont inclus dans la liste faite précédemment. Ces tickets sont près de 35000.

Avec l'aide de la chef de l'équipe de programmeurs, nous avons regroupé les noms des bibliothèques par modules. Ceci nous a permis d'ajouter à chaque ticket le nom du module qu'il concerne en se basant sur la bibliothèque qu'il concerne.

Le chef de l'équipe de programmeurs nous a aussi expliqué les différentes conventions, notamment d'enregistrement des dates et des durées sur les tickets. Nous nous sommes servis pour créer un convertisseur pour corriger les fautes de frappe liées aux conventions. Ce convertisseur couvre tous les cas et les convertis en valeurs numériques. Enfin, nous avons recatégorisé (en évolution ou défaut) les 35000 tickets en nous basant sur un dictionnaire simple fourni par la chef de l'équipe de programmeurs.

4.2.3 Indicateurs à mesurer

Pour analyser les données, il faut tout d'abord déterminer les indicateurs à mesurer. Comme nous l'avons présenté dans le Chapitre 2, plusieurs indicateurs sont proposés. Ces indicateurs utilisent souvent l'historique des tickets et l'historique du code source. Certains de ces indicateurs mesurent l'effort pour traiter les tickets [Port 2018]. Selon [Pigoski 1996], la proportion d'effort de correction de défauts est habituellement de 20% à 25% dans un processus de développement. Dans le cas de la société CIM, nous nous sommes tournés vers la base de tickets et élu des questions visant à évaluer le temps nécessaire pour traiter un ticket ouvert et le fermer à partir des informations disponibles sur les tickets. Ces questions sont :

— Quelle est la proportion de temps d'évolution par rapport au temps de correction de défaut ?

— Combien de temps faut-il pour fermer un ticket ?

— Quel est le temps de développement passé sur un ticket ?

— Quel est le temps de test, par les programmeurs, sur un ticket ?

— Quel est le temps de test, par les testeurs, sur un ticket ?

— Quelle est la différence entre le temps estimé et le temps réel pour un ticket ?

Les modernisations de pratiques ont commencé dans la société CIM fin 2019, donc pour évaluer les pratiques avant la modernisation, nous considérons les tickets fermés en 2019.

4.2.4 Analyse des données

Ici nous allons présenter l'analyse de données faite pour toutes les questions formulées ci-dessus.

La proportion de temps d'évolution par rapport à la correction de défaut : Pour connaitre la proportion de temps d'évolution par rapport à la correction de défaut, nous avons calculé le temps de fermeture de chaque ticket (la durée entre la date d'ouverture et la date de fermeture du ticket). Ensuite, nous avons calculé la moyenne de temps de fermeture respectivement pour les tickets d'évolutions et de défauts.

Le temps qu'il faut pour fermer un ticket : Pour connaitre le temps qu'il faut pour fermer un ticket, nous avons respectivement regroupé les tickets d'évolutions et de défauts par an. Ensuite, nous avons calculé le temps de fermeture de chaque ticket (la durée entre la date d'ouverture et la date de fermeture du ticket). Enfin, nous avons calculé la moyenne de temps de fermeture par an respectivement pour les tickets d'évolutions et de défauts.

Le temps de développement passé sur un ticket : Un ticket peut être développé plusieurs fois à différents moments par les programmeurs. Ainsi, à chaque intervention, un temps de développement est renseigné sur le ticket. Pour connaitre le temps de développement, nous avons respectivement regroupé les tickets d'évolutions et de défauts par année. Ensuite, nous avons calculé le temps de développement de chaque ticket en additionnant les différents temps de développement du ticket. Nous avons enfin calculé la moyenne par an du temps de développements des tickets d'évolution et de défaut.

Le temps de test, par les programmeurs, sur un ticket : Un ticket peut être testé plusieurs fois à différents moments par les programmeurs. Ainsi, à chaque test de programmeur, un temps de test de programmeur est renseigné sur le ticket. Pour connaitre le temps de test, nous avons respectivement regroupé les tickets d'évolutions et de défauts par an. Ensuite, nous avons calculé le temps de test de programmeurs pour chaque ticket en additionnant les différents temps de test de programmeur du ticket. Nous avons enfin calculé la moyenne par an du temps de test de programmeurs des tickets d'évolution et de défaut.

Le temps de test, par les testeurs, sur un ticket : Un ticket peut être testé plusieurs fois à différents moments par les testeurs. Ainsi, à chaque test de testeurs, un temps de test de testeurs est renseigné sur le ticket. Pour connaitre le temps de test de testeurs, nous avons respectivement regroupé les tickets d'évolutions et de défauts par an. Ensuite, nous avons calculé le temps de test de testeurs pour chaque ticket en additionnant les différents temps de

test de programmeur du ticket. Nous avons enfin calculé la moyenne par an du temps de test de testeurs des tickets d'évolution et de défaut.

La différence entre le temps estimé et le temps réel pour un ticket : Pour un ticket d'Izy Protect donné, la chef d'équipe de programmeurs estime le temps nécessaire pour le traiter. Ce temps est renseigné sur le ticket. Il y a également un champ enregistrant le temps total passé sur le ticket par le programmeur. Pour connaitre la différence entre le temps estimé et le temps réel, nous avons respectivement regroupé les tickets d'évolutions et de défauts par an. Ensuite, nous avons calculé la différence entre le temps total passé et le temps estimé pour chaque ticket. Nous avons enfin calculé la moyenne par année de la différence entre le temps estimé et le temps réel des tickets d'évolution et de défaut. Si cette différence est positive, il y a eu une sous-estimation, et si elle est négative, il y a eu surestimation.

Après avoir mesuré les indicateurs, nous allons présenter ces derniers sur un graphique à deux dimensions. Nous allons également tracer la régression linéaire de la courbe pour chaque graphique afin d'avoir une meilleure idée de la tendance.

4.2.5 Résultat de mesure quantitative

Cette Section présente les résultats des mesures des différents indicateurs ainsi que leurs interprétations.

Proportion de temps d'évolutions par rapport aux corrections des défauts

TABLE 4.1: Tickets

	correction de défauts	**évolution**	**Total**
% Temps passé	28%	72%	100%

Le tableau 4.1 donne le nombre de tickets de défaut par rapport aux tickets d'évolution. Nous remarquons que 28% du temps est passé sur les défauts contre 72% sur les évolutions. La proportion d'effort pour la correction des défauts de 28% est proche de 25%. Donc la proportion d'effort des défauts est normale.

Temps pour fermer un ticket

La Figure 4.1 présente les valeurs du temps de fermeture pour les tickets d'évolution (en bleu) et de défaut (en rouge). On peut remarquer qu'il faut plus de temps

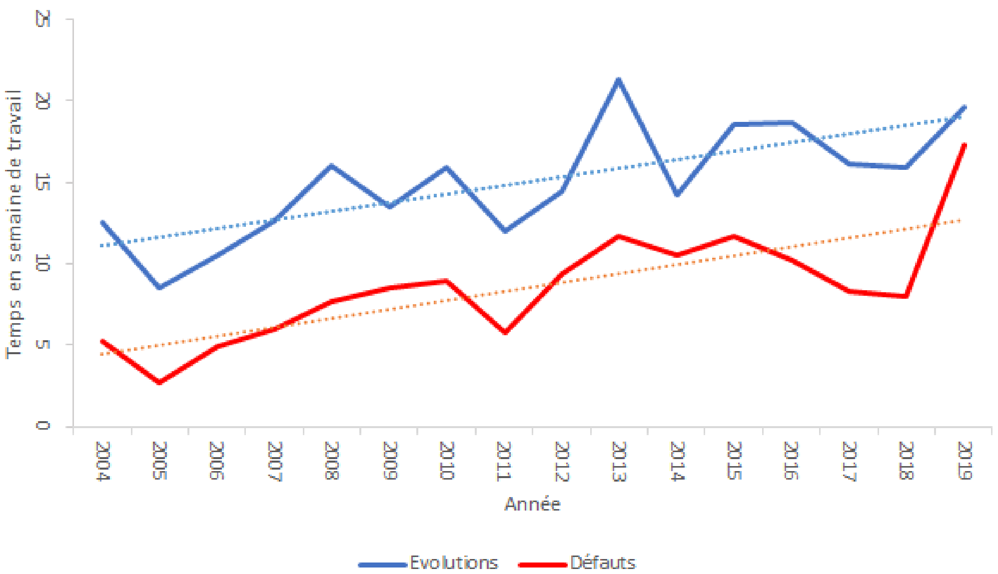


FIGURE 4.1: Temps pour fermer les tickets d'évolution (en bleu) et de défaut (en rouge)

(en moyenne) pour fermer un ticket d'évolution qu'un ticket de défaut, ce qui semble naturel.

Nous notons également que le temps augmente au fil des ans pour les deux catégories. Ainsi, nous notons que le temps de clôture sur les 11 années a été multiplié par 1,6 pour les tickets d'évolution et par 2,6 pour les tickets de défaut.

Cette augmentation peut être liée à une baisse de qualité d'Izy Protect. Une autre explication pourrait être l'augmentation du nombre de tickets au fil des années et le manque de temps pour tous les traités, ce qui ferait qu'ils restent ouverts plus longtemps. Celle-ci peut être aussi causée par la stratégie de planification et l'ordre de priorité que la société CIM donne au ticket. Sachant que les tickets sont planifiés en fonction de leurs priorités et leurs anciennetés (Chapitre 3), les tickets non prioritaires pourraient être traités et fermés plus tard, ce qui affecterait la moyenne du temps de fermeture par an et la courbe. Donc nous ne pouvons pas tirer une conclusion définitive par rapport à cette augmentation.

Temps de développement sur un ticket

La Figure 4.2 présente le temps passé par les programmeurs pour implémenter une solution aux tickets d'évolution (en bleu) et de défaut (en rouge).

Il faut plus de temps (en moyenne) pour mettre en œuvre une solution pour un ticket d'évolution que pour un ticket de défaut. Cela semble naturel.

Nous notons également que le temps augmente au fil des ans pour les deux

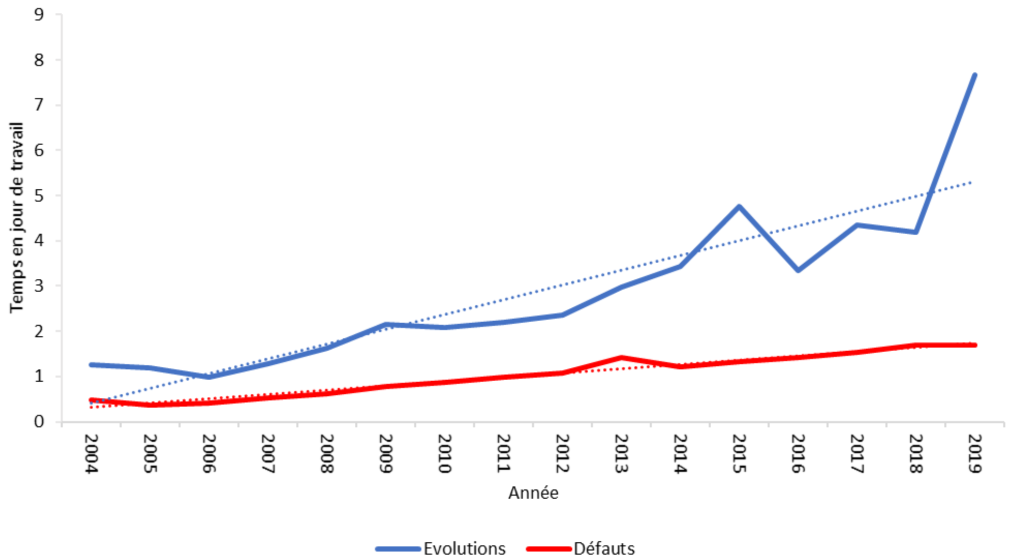


FIGURE 4.2: Temps passé par les programmeurs implémenter une solution pour les tickets d'évolution (en bleu) et de défaut (en rouge).

catégories.

Enfin, nous notons que le temps passé par les programmeurs pour mettre en œuvre une solution a été multiplié par un facteur de 6 pour les évolutions et 3,6 pour les défauts.

De nouveau, cela peut être le signe de baise de qualité d'Izy Protect et/ou d'augmentation de la charge de travail sur les tickets. Cette augmentation peut aussi être le signe d'un manque d'effectif au niveau des programmeurs.

Temps de test manuel par les programmeurs

La Figure 4.3 présente la tendance du temps passé par les programmeurs à tester leur solution pour les tickets d'évolution (en bleu) et de défaut (en rouge). Nous pouvons observer que le temps est en baisse pour les tickets d'évolution et les tickets de défaut. Les programmeurs passent respectivement en moyenne environ 1 heure pour tester un ticket d'évolution et 40 minutes pour un ticket de défaut. Cette tendance est en opposition à la tendance de la Figure 4.2. Les programmeurs passent de plus en plus de temps à traiter les tickets tandis qu'ils passent de moins en moins de temps à tester leur code. Ceci est peut-être signe qu'ils n'ont pas beaucoup de temps pour traiter et tester les tickets.

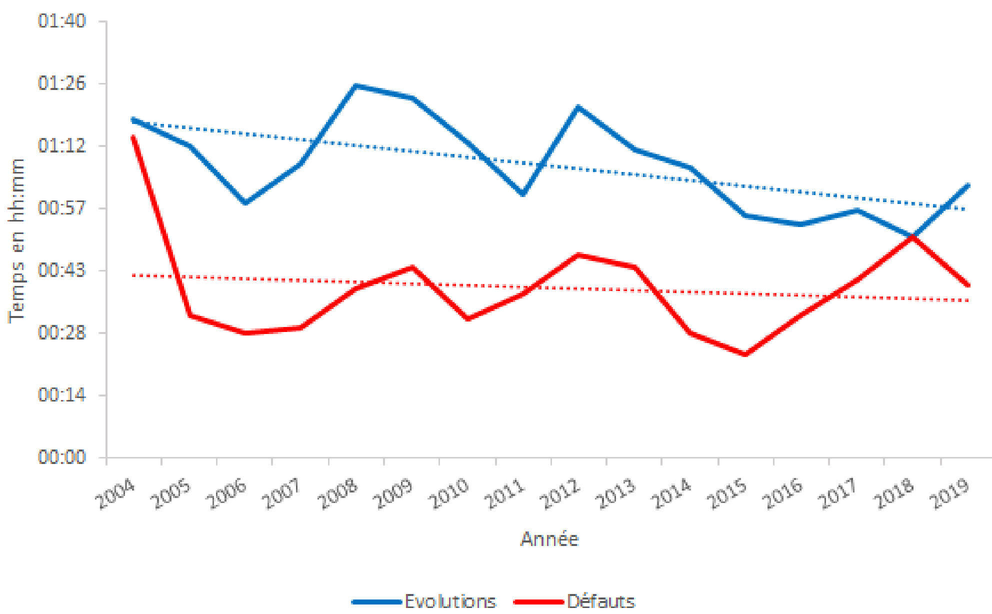


FIGURE 4.3: Temps passé par les programmeurs pour tester les tickets d'évolution (en bleu) et de défaut (en rouge)

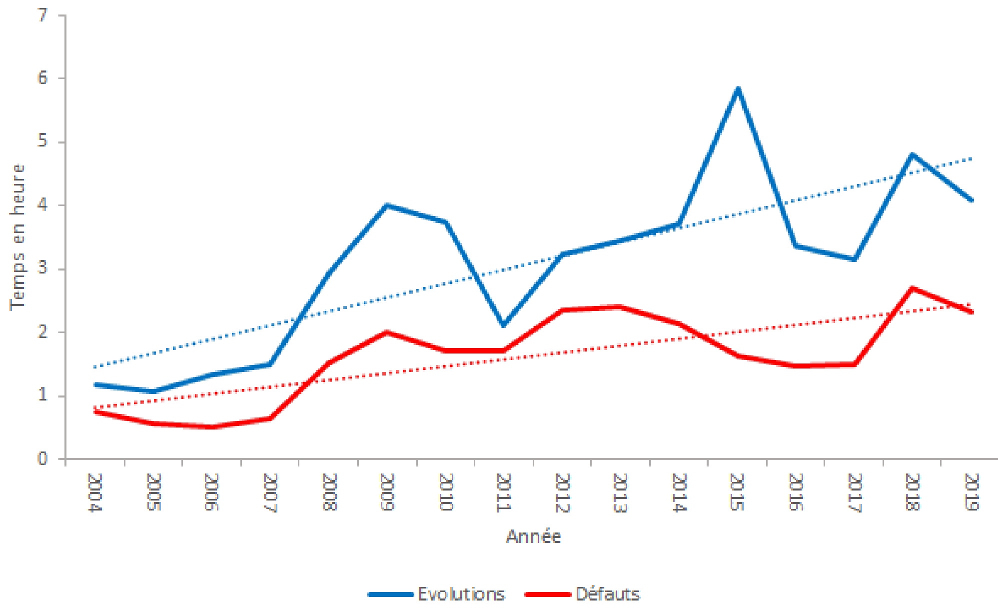


FIGURE 4.4: Temps passé par l'équipe de testeur pour tester les tickets d'évolution (en bleu) et de défaut (en rouge)

Temps de test par le testeur

La Figure 4.4 présente la tendance du temps passé par les testeurs pour tester (tests fonctionnels) les tickets d'évolution (en bleu) et de défaut (en rouge). On remarque que ces tendances croissent. Autrement dit, les programmeurs passent de moins en moins de temps à tester (comme nous l'avons vu ci-dessus) et les testeurs passent de plus en plus de temps pour tester les tickets.

On peut supposer que les testeurs passent plus de temps à tester, parce que les programmeurs testent de moins en moins leur code.

Estimation de temps

FIGURE 4.5: Différence entre le temps passé par les programmeurs et le temps estimé pour les tickets d'évolution (en bleu) et de défaut (en rouge)

La Figure 4.5 montre la différence entre le temps passé par les programmeurs et le temps estimé pour les tickets d'évolution (en bleu) et de défaut (en rouge). Nous pouvons observer deux périodes évidentes : avant et après mai 2013. Avant mai 2013, le temps passé par le programmeur est supérieur au temps estimé avec une différence croissante pour les tickets d'évolution et de défaut. Cela signifie que les programmeurs n'avaient pas assez de temps pour résoudre les tickets. Cela a contraint les programmeurs dans leur tâche, ce qui a pu avoir un impact négatif sur la qualité des solutions et donc du code.

Après mai 2013, le temps estimé est supérieur au temps passé pour les tickets d'évolution et de défaut. La différence se rapproche de 0 pour les tickets d'évolu-

tion. Mais il y a une autre grande sous-estimation en 2019. Nous pourrions relier cela à la diminution du temps de test des programmeurs (page 42) pour les tickets d'évolution et de défaut (qui pourrait être dû à un manque de temps). Cependant, il faut noter que nous n'avons pas vu la même différence de deux périodes dans la section précédente, il n'est donc pas clair que la sous-estimation du temps soit à l'origine de la diminution du temps de test.

4.3 Étude qualitative du processus de développement dans la société CIM

Nous avons présenté l'étude quantitative du processus de développement de la société CIM. Nous allons maintenant présenter l'étude qualitative. Jusqu'à mi-2019, le processus de développement de la société CIM est composé des différentes pratiques présentées dans le Chapitre 3. Ces pratiques sont : *Stand-up meeting*, organisation d'équipe, cycle de développement du logiciel, gestion du code source, et qualité du code. Elles sont décrites dans la Section 3.3.2 (page 28) et la Section 3.3.3 (page 30).

En mi-2019, la société CIM a introduit certaines pratiques recommandées dans les méthodes agiles, dans l'espoir d'améliorer la qualité de ses produits et ses délais de livraison. Elle a aussi amélioré ses anciennes pratiques suivant les recommandations des méthodes agiles. Après l'introduction et l'amélioration des pratiques, nous avons collecté les avis des différents développeurs d'Izy Protect sur les pratiques de développement dans la société CIM. Notre objectif est d'essayer de corréler les données qualitatives avec les résultats de l'étude quantitative faite dans la Section 4.2. Dans cette section, nous allons présenter le point de vue des développeurs sur les pratiques de développement telles qu'elles étaient avant l'amélioration de ces dernières.

4.3.1 Collecte des données

Pour recueillir les points de vue des développeurs d'Izy Protect sur les pratiques de développement de la société CIM, nous avons mené une étude exploratoire en faisant passer des entrevues semi-structurées à 17 développeurs d'Izy Protect.

Les participants ont des responsabilités variées. En effet, ils sont composés de dix programmeurs, un testeur, le chef d'équipe de testeurs, un responsable client, deux analystes, la chef d'équipe de programmeurs ainsi que le *CTO* [1].

Le Tableau 4.2 donne quelques données sur tous les développeurs d'Izy Protect : leur expérience globale (dans des activités liées au développement de logi-

1. directeur technique et technologique

TABLE 4.2: Liste des développeurs d'Izy Protect. Expérience professionnelle totale et ancienneté dans l'entreprise en années. La dernière colonne indique les participants aux entrevues.

Membres d'équipe	Expérience	Ancienneté	entrevue
Programmeur 1	1 an de PowerBuilder	1 an	✓
Programmeur 2	12 ans de PowerBuilder	7 ans	✓
Programmeur 3	6 ans de PowerBuilder	6 ans	✓
Programmeur 4	10 ans de PowerBuilder	10 ans	✓
Programmeur 5	20 ans	3 ans	✓
Programmeur 6	22 ans de PowerBuilder	22 ans	✓
Programmeur 7	20 ans de Cobol & PLSQL	2,5 ans	✓
Programmeur 8	1 an de PowerBuilder	1 an	✓
Programmeur 9	2 ans de PowerBuilder	2 ans	-
Programmeur 10	1,5 an de PowerBuilder	1,5 an	✓
Programmeur 11	7 ans de PowerBuilder	7 ans	✓
Programmeur 12	11 ans	11 ans	-
Chef d'équipe Programmeurs	7 ans chef d'équipe programmeurs.	18 ans	✓
CTO	19 ans de gestion de projet	2 ans	✓
Testeur 1	1 an	1 an	✓
Testeur 2	9 ans	9 ans	-
Testeur 3	15 ans	7 ans	-
chef Testeurs	13 ans analyse & programmation 7 ans chef Testeurs	3 ans	✓
Analyste 1	17 ans analyse & programmation	9 ans	✓
Analyste 2	23 ans analyse & programmation	3 ans	✓
Analyste 3	7 ans	6 ans	-
Analyste 4	11 ans	11 ans	-
Chargée clientèle 1	17 ans	17 ans	✓
Chargée clientèle 2	9 ans	2.5 ans	-
Chargée clientèle 3	2 ans	2 ans	-

ciels), leur ancienneté dans l'entreprise, et s'ils ont participé aux entrevues.

En raison de la pandémie de COVID-19 et du travail à distance, certaines entrevues ont été réalisées à distance. Toutes les entrevues ont été enregistrées pour faciliter l'analyse et nous avons demandé le consentement des participants à cet effet. Nous avons également assuré les participants de l'anonymat de leurs réponses. Notons que pour promouvoir cet anonymat, toutes les personnes interrogées sont désignées par le terme « il » dans le document, même les femmes qui sont peu nombreuses et donc plus facilement identifiables à certains postes.

Selon les participants, les entrevues ont duré de 30 minutes à 2 heures. Les entrevues ont suivi un questionnaire où cinq questions étaient posées pour chacune des pratiques considérées. Les pratiques considérées durant les entrevues sont les anciennes et les nouvelles pratiques de développement de la société CIM. Le Tableau 4.3 montre les questions et les pratiques considérées pour chaque catégorie de participants : *CTO*, développeur technique (les programmeurs et la chef d'équipe de programmeurs), et développeur non technique (tous les autres développeurs). Le questionnaire du *CTO* est différent, car, premièrement, il est l'initiateur de beaucoup de changements de pratiques (il sait « pourquoi ») et, deuxièmement, il est moins directement impacté par les changements puisqu'il ne participe pas aux activités concrètes de développement (analyse, codage, tests).

4.3.2 Analyse des données

Pour analyser les 17 entrevues semi-structurées, nous avons suivi l'approche de la *Grounded Theory*, une technique qualitative, inspirée par [Khadka 2014]. Nous avons présenté la *Grounded Theory* dans le Chapitre 2. C'est une méthode de recherche exploratoire qui vise à découvrir de nouvelles perspectives et idées, plutôt qu'à confirmer celles qui existent déjà [Charmaz 2014].

Après l'enregistrement des entrevues, ces dernières ont été transcrites et rendues anonymes. Les transcriptions ont été analysées, avec MAXQDA [Kuckartz 2019], par le biais d'un *codage* (ou étiquetage). Chaque transcription a été décomposée en une base de données d'*étiquettes* : une étiquette est attribuée à l'idée principale des phrases jusqu'à la saturation. Ensuite, nous avons itérativement regroupé les *codes* en *concepts* qui sont ensuite organisés par les questions des entrevues.

Dans les sections suivantes, nous allons présenter la synthèse de l'analyse des entrevues sur les pratiques de développement de la société CIM avant la modernisation. Dans ce sens, la Section 4.3.3 va présenter le point de vue des développeurs sur la *Stand-up meeting* ; la Section 4.3.4 va présenter les positions des développeurs sur l'organisation d'équipe ; la Section 4.3.5 va présenter les positions sur le cycle de développement logiciel de la société ; la Section 4.3.6 va présenter les positions sur la gestion de code source et la Section 4.3.7 va présenter les positions

TABLE 4.3: Liste des questions des entrevues

Catégories	Membre	Pratiques	Questions
Non-Technique	Un responsable client	Cycle de développement	Êtes-vous au courant de l'introduction de cette pratique ?
	Un testeur	Organisation d'équipe	En quoi consiste cette pratique ?
	Chef d'équipe de testeurs	Tableau de bord	Quelle était la situation avant l'introduction de cette pratique ?
	Analystes		Quelle est la situation actuelle avec la pratique en place ?
			Quelle amélioration pouvez-vous proposer à cette pratique ?
Technique	Programmeurs	*Stand-up meeting*	Êtes-vous au courant de l'introduction de cette pratique ?
	Chef d'équipe de programmeurs	Organisation d'équipe	En quoi consiste cette pratique ?
		Cycle de développement	Quelle était la situation avant l'introduction de cette pratique ?
		Gestion du code source	Quelle est la situation actuelle avec la pratique en place ?
		Tableau de bord	Quelle amélioration pouvez-vous proposer à cette pratique ?
		Revue de code	
		“Linter"	
CTO		*Stand-up meeting*	En quoi consiste cette pratique ?
		Organisation d'équipe	Quelle était la situation avant l'introduction de cette pratique ?
		Cycle de développement	Pourquoi c'était-il un problème ?
		Gestion du code source	Comment la pratique a-t-elle été choisie ?
		Tableau de bord	Quelles ont été les difficultés de mise en œuvre ?
		Examen du code	Quelle est la situation actuelle avec la pratique en place ?
		“Linter"	Quels changements cela a-t-il apporté en mieux/moins bien ?
			Quelle amélioration pouvez-vous proposer à cette pratique ?

sur la qualité de code.

4.3.3 *Stand-up meeting*

Comme nous l'avons présenté dans le Chapitre 3, le *Stand-up meeting* concerne seulement l'équipe de programmeurs. Elle est faite deux fois par semaine. Selon le chef de l'équipe de programmeurs « *ces réunions permettent d'éviter que les programmeurs travaillent sur le même sujet. Elles permettent aussi d'apporter rapidement de l'aide au programmeur en cas de blocage* ». Les programmeurs 1, 2, 3 et 4 appuient ce point de vue en rapportant que le *Stand-up meeting* est une bonne chose, car il apporte de la cohésion et l'entraide dans l'équipe : « *ces réunions apportent de la cohésion dans le groupe et l'entraide entre les collaborateurs* ». Malgré les avantages perçus du *Stand-up meeting*, les participants ont identifié quelques défauts à cette pratique. Les programmeurs affirment qu'il ne tient pas souvent dans la durée prévue. De plus, le programmeur 5 affirme : « *ces réunions sont plus un moment d'échange, mais sans le contexte agile qui nous impose de dire ce qu'on a fait, ce qu'on va faire et les points de blocage et nous arrêter là. Le* Stand-up meeting *est beaucoup plus un bureau de plainte au départ* ». Le programmeur 3 continue en disant : « *le* Stand-up meeting *n'est pas assez orienté pour qu'on puisse remonter systématiquement les problèmes. On y pense de temps en temps, mais on n'est pas censé dire pourquoi on n'avance pas et l'analyser. Par conséquent, ça reste seulement du ressenti* ».

En somme, les programmeurs trouvent positif le *Stand-up meeting*. Par contre, il manque le fait que la réunion dépasse la durée prévue pour cette dernière. De plus, le *Stand-up meeting* ne permet pas aux programmeurs d'analyser ce qui les bloque, car la réunion ne force pas les participants à dire ce qu'ils ont fait, ce qu'ils vont faire et leurs points de blocage.

4.3.4 Organisation d'équipe

Tous les intervenants des entrevues ont pointé du doigt l'organisation des équipes comme une des causes majeures des difficultés dans le processus de développement logiciel dans la société CIM. En effet, le PO 1 dénonce le problème de communication en disant : « *on a du mal à se trouver des créneaux pour se parler* ». D'ailleurs, le *CTO* rapporte que le manque de communication entre les équipes a provoqué des tensions entre les développeurs.

De plus, les programmeurs sont réticents pour organiser des réunions avec l'équipe d'analyse pour demander davantage d'informations (manquantes dans un document de spécification d'un ticket par exemple) en raison des retards que cela pourrait engendrer. Le programmeur 2 rapport que « *le lien entre services est compliqué et long* ». Les programmeurs 1 et 3 à leur tour disent que « *parfois, la spéci-*

fication est écrite longtemps avant. Ils n'osent pas trop déranger la personne qui a écrit cette dernière pour poser des questions ou pour revenir sur la spécification ». En conséquence, les solutions implémentées pour les tickets ne sont pas toujours conformes aux attentes du client (programmeur 5).

De la même façon, les testeurs qui testent les tickets traités par les programmeurs ne sont pas épargnés par ce problème. Le chef d'équipe de testeurs explique :« *quand on a une question, on doit envoyer un mail et attendre que l'analyste soit disponible ou prendre rendez-vous pour avoir une réponse. Ou bien il faut contacter le chef de l'équipe de programmeurs pour pouvoir contacter les programmeurs. On perd du temps et en efficacité* ». Ceci fait que les testeurs testent parfois des fonctionnalités qui ne correspondent pas à la spécification (ou la demande) du client parce qu'ils manquent d'informations sur les tickets testés : « *il arrive que le ticket soit mal testé* » (programmeur 3).

Il en résulte des demandes de corrections sur les tickets déjà traités de la part du client et beaucoup de travail à refaire. De telles corrections de tickets après la livraison signifient revenir sur un travail effectué plusieurs semaines ou mois auparavant : « *quand tu as un retour, tu dois te remettre dans du code que tu as fait trois mois avant. Donc, on perd beaucoup de temps.* » (programmeur 2 et 3).

Normalement, la chef d'équipe de programmeurs planifie et supervise le traitement des tickets des programmeurs. Mais au fur et à mesure que de nouveaux programmeurs sont engagés, il devient de plus en plus difficile pour le chef d'équipe de superviser le travail de tous les programmeurs. Les programmeurs 1 et 4 rapportent : « *il y a beaucoup du monde dans l'équipe PowerBuilder et c'est compliqué à gérer pour le chef d'équipe au niveau de la planification* ».

L'organisation des équipes fait que les tickets doivent être étudiés et reprogrammés à différents stades, d'abord par l'équipe d'analyse, puis par l'équipe de développement et enfin l'équipe de test. Cela entraîne des retards (car chaque équipe a ses priorités) et une duplication du travail.

En somme, les développeurs trouvent l'organisation des équipes négative. Ils ont rapporté que cette organisation cause des problèmes de planification des tickets, de qualité, de latence de livraison et de communication.

4.3.5 Cycle de développement du logiciel

Le cycle de développement logiciel de la société est sur une période de 3 mois, au bout desquels une version est arrêtée et livrée au client. Toutefois, durant cette période, il peut y avoir des versions *correctives* arrêtées pour des correctifs tickets de défauts. Les programmeurs 3 et 4 dénoncent les interruptions causées par les tickets urgents (défauts) qui fragilisent la qualité du travail des programmeurs. De plus, les programmeurs ont du mal à avoir une vision sur le projet sur lequel ils sont en train de travailler. Par exemple, les programmeurs 2 et 5 considèrent qu'il

n'y a pas de cycle de développement : « *il y a ni de début ni de fin* ». De la même façon, le PO 1, le PO 2, *CTO*, et les programmeurs 2 et 6 rapportent qu'il n'y a pas de visibilité par rapport à quand une fiche va finir : « *il y a un manque de visibilité sur le retard potentiel qu'on a sur un ticket* » (*CTO*).

Les PO 1 et 2 expliquent aussi qu'avec ce cycle de développement, il y a une perte de temps entre le moment où la spécification du ticket est rédigée, développée et testée : « *par exemple pour une revue de spécifications qui est faite en janvier où le sujet est chaud, le développement est fait 2 mois plus tard et la recette est faite 2 mois plus tard après le développement* » (PO 1).

Pour le programmeur 4, les interruptions dues aux tickets urgents créent des décalages dans la planification : « *on a une planification qui est faite par la chef d'équipe de programmeurs, mais elle bouge tout le temps à cause des imprévus* ». Le *CTO* et le RC appuient l'affirmation du programmeur 4 en notifiant que la société CIM a des difficultés à gérer la planification et la priorisation des tickets à cause du cycle de développement : « *on a une grosse difficulté à ordonner les sujets qu'on a et à garder les priorités stables* » (*CTO*). « *Cette situation pose de problèmes parce qu'on donne plus d'importance aux gros clients. En général, les clients souvent mis de côté sont mécontents* » (RC). Par conséquent, la société CIM a du mal à livrer à temps : « *on a du mal à tenir nos engagements envers les clients* » (chef de l'équipe de testeurs, *CTO*).

En somme, les développeurs trouvent le cycle de développement logiciel négatif. Les participants rapportent que le cycle de développement pose des problèmes de visibilité sur le planning des tickets, la priorisation des tickets et surtout un retard de livraison.

4.3.6 Gestion du code source

Comme nous l'avons expliqué dans le Chapitre 3, le code source d'Izy Protect est géré avec une méthode propre à la société CIM. Néanmoins, cette pratique a causé des problèmes que nous allons discuter ici.

En effet, le programmeur 7 rapporte que pour éviter de travailler en même temps sur les mêmes parties du code, les programmeurs s'appuient sur une communication informelle : « *Si on ne communique pas entre nous, on n'a pas la connaissance de qui a fait quoi* ».

Le programmeur 11 explique que la fusion des modifications est souvent faite avec un outil générique de comparaison de fichiers comme WinMerge. Mais cela pose parfois de problème : « *on peut utiliser l'outil de merge. Le merge marche une fois sur 2. Cependant, de temps en temps, il nous dit que notre version est trop vieille et qu'il n'arrive pas à le mettre à jour* ».

Les modifications sont intégrées dans le code « référentiel » d'Izy Protect en copiant manuellement le code de la version du programmeur vers la version du « ré-

férentiel ». Par contre, les programmeurs 3 et 11 rapportent que la gestion et l'intégration du code source sont difficiles : « *le travail collaboratif du programmeur est à la fois laborieux et archaïque pour l'archivage du code, et l'intégration des développements* » (programmeur 3). « *Souvent sur de petites modifications, c'est long, car cela revient à revenir sur des fonctions et copier-coller des lignes. Ce qui est souvent source d'écrasement. Le gros problème est qu'en cas d'écrasement de code source, il est très difficile de savoir où l'écrasement a eu lieu, et de remonter vers ce dernier. Parfois, on peut perdre beaucoup de temps* » (programmeur 11).

Lors d'un changement sur plusieurs parties d'Izy Protect, les programmeurs doivent garder une liste de toutes les parties impactées pour les porter manuellement dans le code source « référentiel » d'Izy Protect une par une plus tard. L'oubli d'une partie signifie que le changement n'est pas complètement reproduit dans le code source « référentiel » d'Izy Protect et donc que la version définitive n'est pas correcte.

De plus, après une modification du code source, le programmeur doit marquer la modification avec une « propriété » spéciale dans PowerBuilder pour avertir les autres programmeurs. Si cela n'est pas fait, quelqu'un d'autre, ignorant la modification, peut copier l'ancien code source sur le nouveau code (non marqué) dans le code source « référentiel » d'Izy Protect.

Il est courant de mettre un commentaire dans le code pour décrire la modification, la date et le programmeur qui l'a effectuée. Il est difficile de retrouver un changement qui a un introduit de défaut dans le code. Il faut, soit utiliser les commentaires dans le code, soit comparer une version stable avec la version qui pose de problème en utilisant un outil de comparaison de fichier. Cela peut aider à déboguer le code plus tard. « *Mais parfois, il y a eu quatre modifications successives, donc on a du mal à se retrouver. Quelquefois, tout simplement, au bout d'un moment, certains commentaires ont disparu* » (programmeurs 3 et 11).

En somme, les développeurs trouvent négative la gestion de code source. Ils soulignent la difficulté du travail collaboratif, de l'intégration des développements et de la détection du code défaillant.

4.3.7 Qualité du code

Avec la complexité du code source d'Izy Protect, les programmeurs ont de plus en plus du mal à le maintenir. « *Travailler sur Izy Protect devient de plus en plus compliqué. On fait juste ce qu'on peut pour livrer au client* ». De plus, les programmeurs risquent d'introduire des régressions dans le code (casser une fonctionnalité existante) à tout moment, parce que le code source d'Izy Protect n'est pas couvert avec suffisamment de tests.

Comme présenté dans le Chapitre 3, les programmeurs ont souvent des réunions de revue de code. Pour ces réunions, les programmeurs doivent apporter les sujets

à discuter. Mais ils manquent de temps pour réfléchir aux sujets à discuter dans ces réunions. De plus, les programmeurs 1, 7 et 10 affirment que ces réunions sont orales, ce qui fait que les choses qui y sont dites peuvent être oubliées : « *On essaie de prendre en compte ce qui est indiqué aux revues de code dans nos développements. Par contre, on ne fait pas forcément référence à tout, étant donné que c'est oral, il peut arriver qu'on oublie certaines choses* » (programmeur 7).

En somme, les programmeurs ont rapporté la difficulté de travailler sur Izy Protect à cause de sa qualité d'une part. D'autres par les réunions de revue de code sont perçus positivement. Néanmoins, les programmeurs souhaiteraient avoir plus de disponibilité pour les préparer. Il faudrait aussi que les informations passées lors de ces réunions soient enregistrées pour permettre aux programmeurs de le consulter plus tard.

4.4 Résumé du chapitre

Dans ce chapitre, nous avons présenté une étude quantitative et qualitative des pratiques de développement utilisées au sein de la société CIM. D'un point de vue quantitatif, les indicateurs (1) temps pour fermer les tickets, (2) temps de développement, (3) le temps de test par l'équipe de testeurs sont en augmentation tandis que le temps de test des programmeurs diminue. Ainsi, à travers ces indicateurs que le processus de développement de la société CIM n'est pas dans une évolution positive avant les changements. Ce qui peut être causé par : la baise de qualité d'Izy Protect ; l'augmentation du nombre de tickets au fil des années et le manque de temps pour les traités ; la stratégie de planification des tickets par la société CIM ; l'augmentation de la charge de travail sur les tickets, etc.

Selon l'étude qualitative des anciennes pratiques de développement de la société CIM, les participants à l'étude jugent certaines pratiques telles que l'organisation d'équipe, le cycle de développement du logiciel et la gestion du code source, contre-productives. Nous avons vu que l'organisation d'équipe a causé des problèmes de communication, de latence dans le traitement des tickets et peut affecter la qualité de livraison. Le cycle de développement du logiciel fait que la société a du mal à respecter les délais de livraison des tickets. Nous avons aussi vu que la gestion de code source pose des problèmes d'intégration de code difficile et d'écrasement de code.

Après cette caractérisation de la situation avant la modernisation des pratiques de développement, nous allons, dans le prochain chapitre, voir en quoi consistent ces modernisations et évaluer leur impact.

Chapitre 5
Modernisation des pratiques de développement

Sommaire

5.1 Introduction 55
5.2 Pratiques modernisées 56
5.3 Étude qualitative des pratiques modernisées 59
5.4 Résumé du chapitre 68

5.1 Introduction

Nous avons présenté dans le Chapitre 2 les normes actuelles en matière de développement logiciel de qualité et efficace. Nous avons aussi présenté dans le Chapitre 3 la situation dans la société CIM au regard de son processus de développement, soulignant les manques dans plusieurs pratiques. La société désireuse d'améliorer la situation a entrepris de moderniser ses pratiques vers les pratiques agiles. En même temps, elle a dû faire face aux conditions exceptionnelles liées à la crise de la COVID-19 avec la généralisation du travail à distance. De nombreux agilistes prônent l'importance de la collocation, de l'interaction en face à face et des artefacts physiques incorporés dans l'espace de travail partagé, que la pandémie de COVID-19 a rendu impossible [Smite 2021, Mikalsen 2021b].

Au Chapitre 4, nous avons caractérisé la situation avant ces changements aussi bien quantitativement que qualitativement. Nous allons maintenant décrire les actions de modernisation menées et évalué

En continuité du Chapitre 4 où nous avons présenté une étude quantitative et qualitative des pratiques de développement de la société CIM avant la modernisation, dans ce chapitre nous allons refaire l'étude qualitative sur les pratiques de développement de la société CIM après la modernisation. Dès que possible, nous allons corréler les données qualitatives avec les résultats (mis à jour) de l'étude quantitative déjà effectuée dans le Chapitre 4. Nous cherchons à identifier les bénéfices perçus de ces changements et les défis auxquels les professionnels sont

confrontés face à la modernisation des pratiques. Nous évaluons également si le contexte du travail à distance dû à COVID-19 a eu une influence sur ces pratiques. Ce chapitre est structuré comme suit : la Section 5.2 va présenter les pratiques modernisées par la société CIM ; ensuite la Section 5.3 va présenter l'étude qualitative que nous avons menée sur ces pratiques ; enfin nous allons conclure dans la Section 5.4.

5.2 Pratiques modernisées

La société CIM a introduit plusieurs pratiques recommandées dans le développement agile, espérant améliorer la qualité de ses produits et le temps de mise sur le marché. Ces actions portent sur différents aspects allant des ressources humaines à la gestion du code source : (1) *Stand-up meeting*, (2) changement organisationnel, (3) cycle de développement logiciel, (4) gestion du code source, (5) vérification de la qualité du code. Nous allons maintenant présenter ces nouvelles.

Stand-up meeting Début 2020, la pratique préexistante des réunions *Stand-up meeting* a été rendue plus conforme aux recommandations du développement agile avec des réunions quotidiennes au lieu de deux par semaine. Elle a lieu au début de la journée. Les participants ne sont pas seulement des programmeurs (comme auparavant), mais aussi quelques testeurs et analystes des exigences (voir Chapitre 3). La réunion est recentrée pour ne traiter que les trois questions pour tout le monde : ce qui a été fait la veille ; quels problèmes ont été rencontrés ; et quels sont les plans pour la journée en cours. Pour que la réunion soit courte (15 à 20 minutes), si un participant a un problème que quelqu'un pourrait l'aider à résoudre, les deux sont encouragés à en discuter après la réunion.

Changement organisationnel Un changement fondamental est intervenu dans la structuration en équipes. Il s'agit d'une conséquence normale de la modernisation des pratiques de développement logiciel, comme l'indique [Bassil 2001].

En fin 2020, la société a réorganisé les développeurs en fonction des domaines d'activité et une équipe supplémentaire a été créée (l'« *équipe run* ») pour traiter les petits problèmes urgents.

Plutôt que de diviser le cycle de vie d'Izy Protect en phases distinctes, avec des équipes séparées pour chacune d'entre elles, les limites des équipes sont redéfinies. Comme dans [Dörnenburg 2018], les équipes sont réorganisées de manière à ce que tous les intervenants de la société pour un domaine d'activité donné soient regroupés dans une seule équipe comprenant des programmeurs, un *Product owner*, et un QA produit. Deux équipes de ce type ont été créées.

Le *Product owner* la responsabilité du « propriétaire » du produit est d'analyser et d'envoyer les tickets aux programmeurs. Il doit notamment s'assurer que les développements répondent aux exigences des clients.

Product QA La tâche principale du *Product QA* est d'effectuer des tests fonctionnels sur les nouveaux développements. Il valide les développements ou signale les régressions. Cette approche est qualifiée de « *you build it, you run it* » [Dörnenburg 2018].

Cette réorganisation permet aux programmeurs d'être mieux intégrés à la source des évolutions (le *Product owner*) et au testeur. Elle évite les malentendus et le rejet de la faute sur les autres.

« Equipe Run » Cette troisième équipe a été créée pour gérer les demandes urgentes. Elle ne traite que les demandes ne dépassant pas deux jours de travail (correction de bogues ou évolution). Cela permet de s'assurer que ces demandes ne perturbent pas le flux de travail des autres équipes. Pour accélérer les choses, les demandes ne sont pas spécifiées de manière formelle, mais de manière interactive par un programmeur et un gestionnaire de clientèle.

L'« Équipe Run » est composée de programmeurs, de gestionnaires de clientèle et d'un expert technique. Les programmeurs alternent entre l'« équipe Run » et une des deux autres équipes. Au début, ils effectuaient une rotation à chaque cycle de développement (voir Cycle de développement logiciel ci-dessous), mais cela rendait difficile la planification des développements. Maintenant, les programmeurs effectuent une rotation tous les trois cycles de développement.

Cycle de développement logiciel Comme expliqué dans le Chapitre 3, la société CIM utilise des cycles de développement, mais ils n'étaient pas formalisés comme le recommandent les approches agiles. Le principal changement introduit, au début de 2020, a été d'avoir des cycles plus courts, de deux semaines (10 jours ouvrables) au lieu de trois mois. Cela a permis à la société de mieux planifier les cycles : huit jours sont consacrés au développement et deux jours sont conservés pour faire face aux imprévus.

Pour préparer une itération, les chargées clientèles sélectionnent avec leurs clients respectifs les tickets ou les demandes qui sont prioritaires pour eux. Les responsables des clients notent tous ces tickets avec leurs priorités dans un fichier central unique (fichier des priorités).

À la veille de chaque cycle, une *réunion de planification*, d'une durée d'environ 1,5 heure, décide des tickets à prendre en compte pour ce cycle. Les participants à la décision sont le *Product owner* (pour une équipe donnée),

le chef d'équipe de chargées clientèles, le CTO [1] et le chef d'équipe de programmeur. Les décisions sont basées sur la priorité des tickets et leurs temps de travail estimé.

Au début du cycle, il y a une réunion de lancement, où les tickets sont discutés dans l'équipe de développement et assignés aux programmeurs. À la fin de cette réunion de lancement, chaque programmeur dispose d'un calendrier des tâches pour le cycle.

Si un ticket n'est pas terminé à la fin d'un cycle, il est reprogrammé pour l'itération suivante.

A la fin d'un cycle, il y a une réunion de rétrospective avec l'équipe de développement, le CTO et le chef d'équipe de développement. Cette réunion dure une heure et analyse les problèmes et les retards du cycle et propose des solutions pour y remédier. La réorganisation de l'équipe décrite (au niveau du Changement organisationnel, ci-dessus) est en fait le résultat de ces réunions de rétrospective.

Contrairement aux pratiques agiles recommandées, les versions ne se produisent pas à la fin de chaque cycle de développement, mais conservent l'ancien calendrier : version-correctif chaque semaine et version « majeure » tous les trois mois.

Gestion du code source Une autre étape de la modernisation des pratiques dans la société a été l'introduction d'un système de contrôle de version à la fin de l'année 2019. Cette introduction a été confrontée à deux défis majeurs. Le premier est dû à PowerBuilder, l'environnement de développement du langage de programmation. Comme expliqué dans le Chapitre 3, cette technologie ne disposait pas de gestion des versions logicielles jusqu'à relativement récemment. Et cette gestion présente encore des problèmes. Le deuxième défi est que, conséquence naturelle, les programmeurs n'étaient pas familiers avec la pratique elle-même.

Avec ces contraintes, la société a choisi d'introduire le système de contrôle de version Subversion (SVN) en fin 2019. Pour les programmeurs qui n'avaient aucune connaissance préalable de la gestion du contrôle de version, SVN, avec son référentiel central et ses copies de travail locales, a été jugé plus facile à comprendre que des outils plus récents (décentralisés) comme Git. Otte [Otte 2009] conclut par exemple que SVN a une meilleure interface utilisateur. Ensuite, la société a organisé huit heures de formation pour l'équipe de programmeurs.

Vérification de la qualité du code Pour vérifier la qualité de code à la société CIM, il existait les revues de code manuelles que nous avons décrit dans la

1. directeur technique et technologique

Section 3.3.2 (page 28). Deux nouvelles actions ont été mises en place pour promouvoir la qualité du code : les revues de code automatisées (outil d'analyse statique du code ou « *linter* ») et les tests unitaires. Les tests unitaires sont présentés en détail dans le Chapitre 6.

Pour vérifier automatiquement les règles de programmation de la société et promouvoir la qualité de code, nous avons implémenter et mis en place un outil d'analyse statique du code (« *linter* ») au début de 2021.

L'outil est un simple vérificateur de règles qui a été mis en place avec neuf règles internes : six règles vérifiant les conventions de nommage des variables, des règles sur le nombre maximum de paramètres des fonctions, l'obligation d'utiliser une `ElseCase` dans toutes les `ChooseCase`. (c.-à-d. `switch`), etc. Cet outil simple vérifie chaque commit et envoie un courriel à l'auteur du commit pour chaque nouvelle violation introduite ou chaque ancienne violation supprimée.

5.3 Étude qualitative des pratiques modernisées

Si les avantages des processus de développement sont bien documentés, la transition des anciennes pratiques présente également des défis [Chen 2015]. Afin d'identifier les bénéfices perçus de ces changements, et les défis auxquels les professionnels sont confrontés face à la modernisation des pratiques, nous avons fait des interviews semi-structurées pour recueillir les avis des développeurs de la société avant et après la modernisation. Ensuite nous avons analysé les données avec la *Grounded Theory*, une méthode d'analyse présentée dans le Chapitre 2. Nous avons détaillé notre méthodologie de collecte et d'analyse de donnée dans le Chapitre 4. Dans cette section, nous présentons le point de vue des personnes interrogées sur les actions de modernisation initiées par la société.

5.3.1 Les *stand-up meetings*

La modernisation du *Stand-up meeting* a eu des aspects positifs et négatifs d'après les participants. Nous allons les présenter maintenant.

Selon les programmeurs, avec des réunions tous les deux jours (comme c'était le cas avant les changements), il y avait moins de pression pour analyser la cause profonde des retards. Les gens avaient l'habitude de répondre à la question « Qu'est-ce que j'ai fait ? », mais ne réfléchissaient pas toujours à « Qu'est-ce qui m'a empêché d'avancer ? ». Le fait de tenir des réunions quotidiennes oblige à réfléchir sur les problèmes et leurs solutions. Le programmeur 3 rapporte qu' « *avant, il y [avait] moins de pression. On [était] censé dire ce sur quoi on travaillait, et*

c'est tout. Mais maintenant, chacun est obligé de réfléchir à ce qui l'a retenu. Parce que chaque jour, il doit analyser ses problèmes ».

Certains y voient un bon moyen d'organiser leur travail (ou leur journée). Cela crée également une motivation supplémentaire pour terminer dans la journée ce qui est prévu le matin. C'est un avantage connu des réunions quotidiennes que les réunions tous les deux jours ne semblaient pas apporter.

Dans l'ancienne version, les gens avaient tendance à s'entraider pendant la réunion de *Stand-up meeting*, ce qui faisait « perdre du temps » à toute l'équipe alors que deux personnes pouvaient discuter d'un problème particulier. Avec des discussions plus ciblées, la résolution de problèmes spécifiques est laissée à l'appréciation de l'équipe après la réunion de synthèse. D'ailleurs, le programmeur 5 voit ceci d'un œil positif en déclarant « *on n'essaie plus de résoudre des problèmes techniques pendant la réunion, mais on se concentre sur les trois questions suivantes : Qu'est-ce que j'ai fait ?, Qu'est-ce qui m'a retenu ?, Qu'est-ce que je vais faire aujourd'hui ?* ».

Le CTO aimerait aller plus loin en reliant les *stand-up meetings* (perspective micro) aux objectifs du cycle de développement (perspective plus large, voir Cycle de développement logiciel), ce qui permettrait de vérifier quotidiennement si l'équipe est toujours sur la bonne voie pour atteindre cet objectif. Ceci n'a pas encore été mis en œuvre.

Malgré les points positifs du *Stand-up meeting* plusieurs programmeurs notent que les réunions ne se terminent pas toujours après les 15 à 20 minutes prévues et ne sont pas encore assez ciblées. Un autre regrette que les remarques ne soient pas toujours constructives. « *Des fois, on est trop dans les détails. Il ne faut pas que ça prenne trop de temps. Il faut éviter de dire des choses non constructives* » (programmeur 2). Un programmeur unique a exprimé un sentiment très négatif à l'égard des anciennes réunions *Stand-up meeting* (tous les deux jours), allant jusqu'à dire qu'il pouvait se sentir humilié. Son opinion sur la nouvelle organisation (tous les jours) s'est améliorée dans le sens où elle est maintenant neutre.

Deux programmeurs ont déclaré que lorsqu'ils travaillent sur des projets importants, le fait de répéter tous les jours la même chose est un peu redondant. « *Moi par exemple régulièrement sur les gros sujets, je n'ai rien à dire. Faire le* Stand-up meeting *les matins, cinq jours, c'est un peu trop redondant. Mais ça ne me choque pas forcément* » (programmeur 11).

Impact de COVID-19 Les réunions se déroulaient à distance au moment de cette étude, à l'aide d'un outil collaboratif de type tableau kanban (Trello). « *Avec la COVID-19 on a un peu transformé ça en point Trello mais c'est la même logique* »(programmeur 11). Mais les programmeurs notent que ces réunions virtuelles ne sont pas aussi bonnes, rendant la communication informelle plus difficile et entraînant « *un manque de motivation* ». Le programmeur 4 ajoute que « *l'on ne*

peut pas savoir si les gens écoutent ou pas ». De plus, « *lors des* Stand-up meetings *où tout le monde doit parler, nous ressentons parfois un manque de motivation dû à l'isolement des employés. Cela a un impact [négatif] sur la productivité de l'équipe* ».

5.3.2 Changement organisationnel en équipes

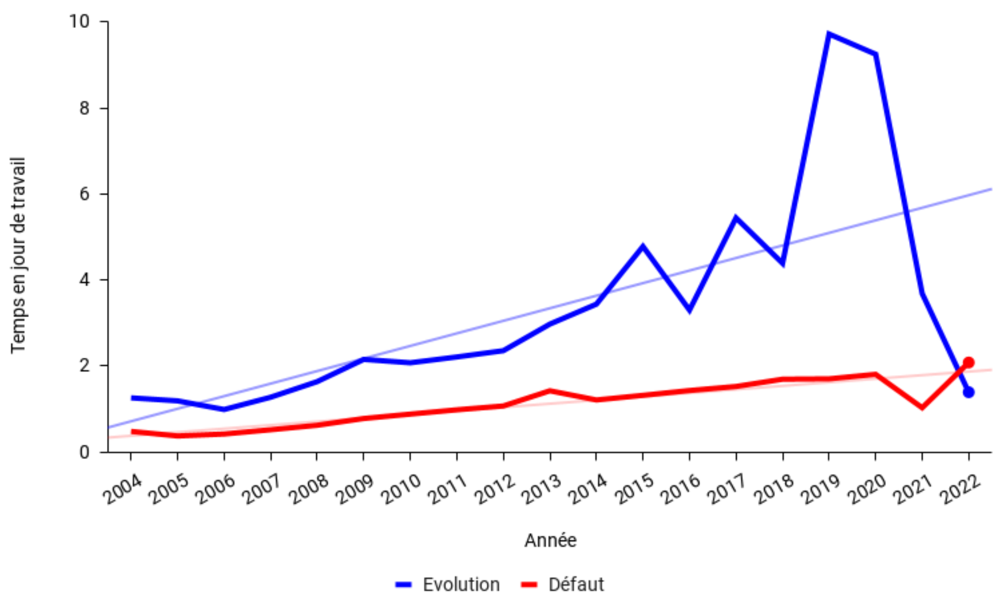


FIGURE 5.1: Temps de développement sur les tickets fermés

La nouvelle organisation en équipes orientées métier est généralement appréciée par les participants. En effet, la nouvelle organisation a amélioré la communication entre les programmeurs, les testeurs et les analystes. De plus, elle est perçue comme accélérant les boucles de rétroaction entre eux, ce qui permet aux programmeurs et aux testeurs d'atteindre plus facilement leurs objectifs. Le programmeur 5 rapporte : « *maintenant nous avons de meilleures relations avec les analystes et les testeurs. L'avantage est que les développeurs comprennent mieux les exigences et que le testeur sait ce qu'il doit tester* ». Le responsable des tests a ajouté : « *nous livrons plus rapidement au client* ». Le résultat perçu est que les tickets sont fermés plus rapidement et avec plus de qualité. Nous avons testé cette perception par rapport à des données réelles. La Figure 5.1 montre bien une baisse significative du temps de travail par ticket en 2020 et 2021, mais 2019 a été une année particulièrement mauvaise dans ce sens. La durée de vie des tickets (Figure 5.2) a bien baissé en 2020 et 2021, mais la durée de vie des tickets de défaut est repartie légèrement

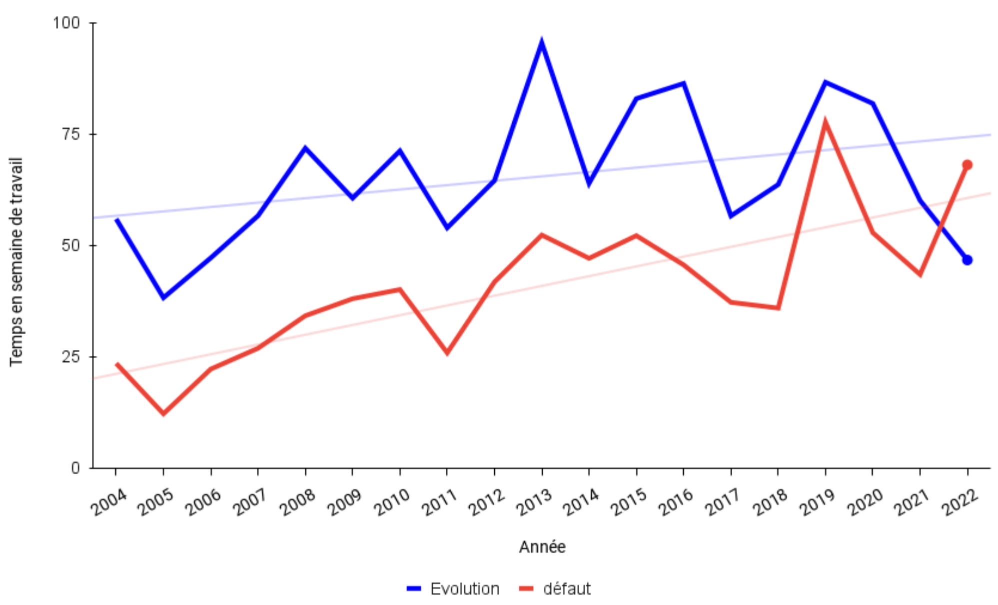


FIGURE 5.2: Durée de vie des tickets (de la date d'ouverture à la date de fermeture)

à la hausse pour les deux premiers mois de 2022. Ces données ne fournissent pas de confirmation claire. Le nombre moyen de tickets fermés par programmeur, Figure 5.3, montre bien une augmentation, passant d'un point bas de 35 en 2019 à 86 en 2021. Les données partielles sur les deux premiers mois de 2022 (17 tickets fermés par programmeur) sont également très encourageantes. Enfin, la Figure 5.4 montre un pourcentage légèrement plus élevé de retour (retravail) sur les tickets fermés en 2020 (16,3%) qu'en 2019 (14,3%). Ce chiffre chute ensuite en 2021 (6,4%) puis diminue légèrement sur les 2 premiers mois de 2022 (5%). Ces données sont encourageantes pour la qualité des tickets fermés. Ces chiffres ne peuvent pas être attribués uniquement à la nouvelle organisation de l'équipe. Toutes les nouvelles pratiques ont pu avoir un impact sur la productivité des programmeurs.

Concernant la nouvelle organisation de l'équipe, deux problèmes ont été mentionnés. Premièrement, les analystes se sont retrouvés avec plus de travail, devant maintenant s'occuper de la gestion du cycle de développement (planification, suivi). Deuxièmement, l'une des deux équipes métier se retrouve parfois à être l'équipe pour « tout ce qui n'est pas pour la première équipe », ce qui rend la gestion plus difficile. Pourtant, l'opinion générale reste que les avantages l'emportent sur ces deux points.

« Equipe Run » Comme décrit (au niveau du Changement organisationnel, ci-dessus), cette équipe gère les demandes urgentes. Les participants ont perçu des

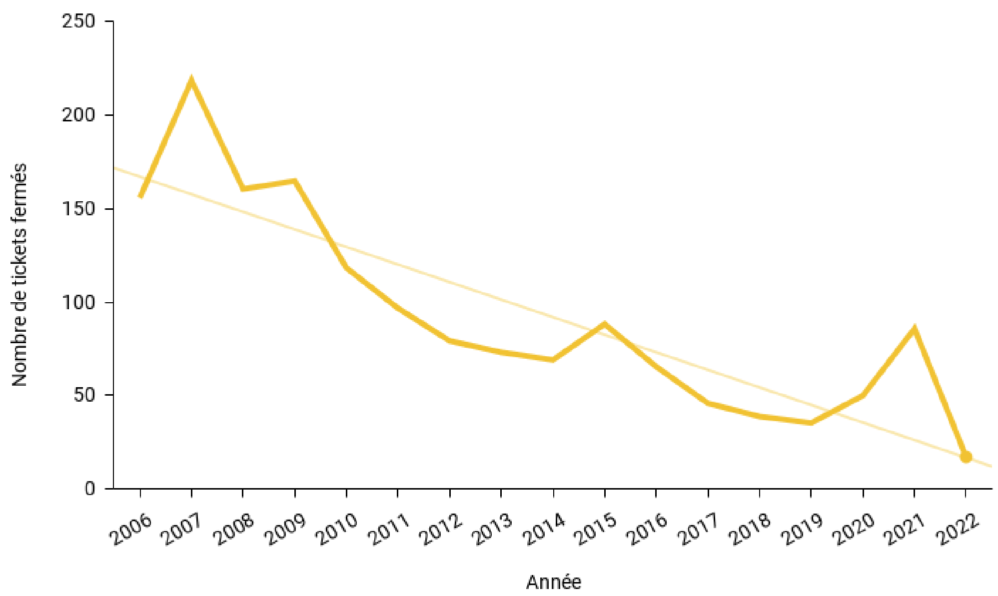


FIGURE 5.3: Nombre de tickets fermés par programmeur

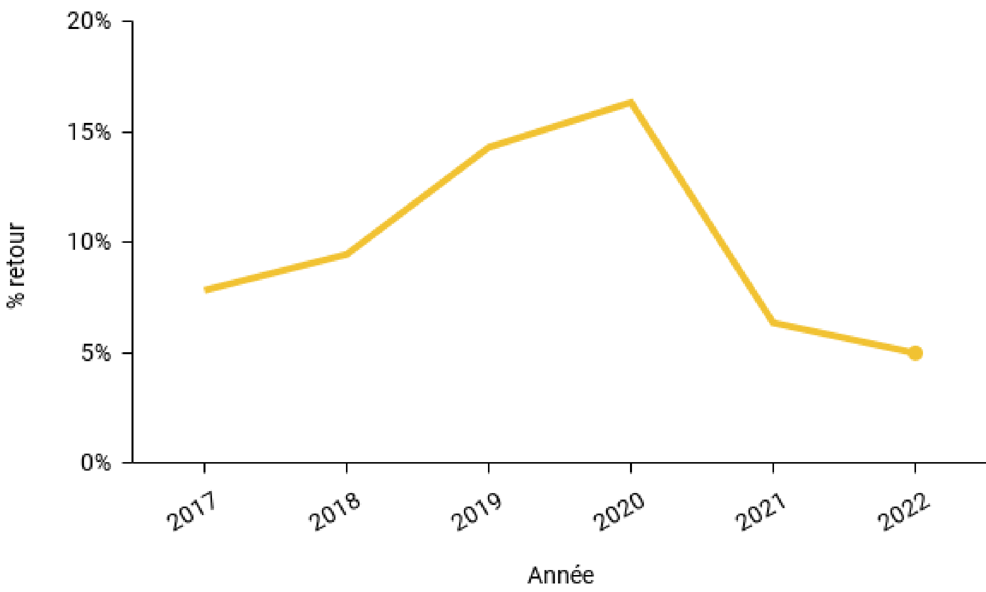


FIGURE 5.4: Retour sur les tickets fermés

points positifs et négatifs sur cette équipe que nous allons présenter maintenant. Du côté positif, un analyste (ne travaillant pas dans cette équipe) exprime que cela facilite la planification des cycles de développement (pour les deux autres équipes) : « *Nous avons un point d'entrée qui [. . .] peut traiter [un ticket urgent] et il ne va pas aux équipes métiers. Par conséquent, le planning [des équipes métiers] est moins impacté.* » Deux programmeurs pensent que « *L'équipe [run] a gagné ses galons [. . .] nous voyons beaucoup de tickets résolus. Même si ce sont des petits tickets, on ne peut pas les laisser de côté. Et cela satisfait les clients* ». La direction des tests a une impression tout aussi bonne : « *Les programmeurs travaillent de manière plus fluide et ont moins de changement de priorités* ». Enfin, le programmeur 1 dit que « *cela permet de varier le travail, ce qui est intéressant* ».

Cependant, les programmeurs qui travaillent réellement dans cette équipe mentionnent la difficulté, le stress, voire « *une certaine crainte* », d'en faire partie (la participation à l'« équipe run » est tournante, voir la Section 5.2). Les principaux inconvénients mentionnés sont les suivants : (1) la rotation implique que les programmeurs n'ont pas toujours les compétences requises pour traiter un ticket urgent « *La difficulté actuellement est que comme les programmeurs tournent, je n'ai pas forcément les mêmes niveaux de compétence, donc je ne peux pas donner certain sujet à certains programmeurs faute de compétence. Ce qui impact sur mon objectif par semaine.* » (chargée clientèle 1) ; (2) lorsqu'il y a un retour de ticket, le programmeur d'origine peut être de retour dans son équipe métier et un autre doit prendre le ticket et le réanalyser. « *Dans l'« équipe run » on ne te fait pas toujours les retours en cas de défaut (*bug*). C'est-à-dire qu'en gros tu fais quelque chose, tu penses que c'est bon, mais tu apprends plus tard qu'il y a un défaut et un de tes collègues a eu un retour là-dessus. Je trouve très frustrant de savoir que je fais quelque chose et ce n'est pas bon, mais personne ne me le dit.* »(programmeur 8). De tels problèmes se produisent rarement dans les équipes métier qui ont un personnel plus permanent.

Il y a aussi des difficultés dans la gestion de l'« *équipe run* » : Par exemple, le responsable client 1 rapporte que « *la difficulté actuellement est que, comme les programmeurs tournent toutes les trois itérations, je n'ai pas forcément les mêmes niveaux de compétences [disponibles]. Je ne peux donc pas donner certains tickets à certains programmeurs en raison d'un manque de compétences. Cela a un impact sur mon objectif hebdomadaire* ».

Notez cependant que, bien que cela rende la vie plus difficile aux programmeurs, cela est voulu par la direction comme une stratégie tendant à la propriété globale du code.

Parmi les améliorations possibles, les gens suggèrent que cette équipe devrait être dotée d'un personnel permanent (et non rotatif), mais il n'est pas clair s'ils ont réellement voulu dire qu'ils préféreraient être affectés en permanence à leur équipe métier (ce qui est considéré comme préférable). D'autres programmeurs souhaitent

que l'« *équipe run* » soit organisée comme les deux autres, avec une analyse formelle des tickets, une meilleure planification du cycle de développement.

En début de l'année 2022, lors d'une discussion informelle, les programmeurs nous ont annoncé que l'« *Equipe Run* » a été fermée.

Sur les données quantitatives, on peut remarquer une légère hausse du temps de travail par ticket de défaut (voir Figure 5.1), suivi d'une augmentation de la durée de vie des tickets de défaut (voir Figure 5.2) sur les 2 premiers mois de 2022. Toutefois, il est trop tôt pour disposer des données solides sur la fermeture de cette équipe.

Impact de la COVID-19 Avec le travail à distance, il peut être difficile pour les programmeurs d'entrer en contact avec le responsable clientèle, et ils finissent parfois par devoir faire tout le travail d'analyse seuls. Cela se ressent surtout dans l'« *équipe run* », où aucune analyse formelle n'est effectuée sur les tickets et où la configuration prévue est que les tickets soient résolus conjointement par le programmeur et le responsable clientèle. Le travail à distance a rendu cette partie plus difficile. Certains programmeurs rapportent que « *les avantages de l'équipe [run] sont réduits à cause de la lenteur des communications due au COVID-19* ».

5.3.3 Cycle de développement du logiciel

Le changement du cycle de développement (passage de cycles de trois mois à deux semaines) est unanimement perçu comme positif. Les avantages perçus par les développeurs de la société CIM interrogés sont : les demandes des clients sont mieux gérées ; les clients et les développeurs ont une meilleure visibilité sur le travail en cours et la fonctionnalité à livrer ; la planification est mieux gérée.

Maintenant, les tickets sont testés tout de suite après le développement. Ainsi, les développeurs obtiennent un retour rapide et le client est livré plus rapidement.

Un développeur chevronné mentionne que la planification du cycle permet d'abord aux développeurs de s'impliquer. Ensuite, de savoir ce que font les autres, pour savoir s'il y a des tickets qu'ils peuvent partager. « *Ceci permet de nous impliquer, de savoir ce que les autres font, s'il n'y a pas des sujets qu'on peut se partager, de savoir pourquoi on doit rajouter un bouton sur une fenêtre* ». (programmeur 5).

Les réunions de rétrospective sont considérées comme une occasion pour la société de réfléchir sur comment résoudre les problèmes rencontrés lors du cycle de développement.

Malgré tous les aspects positifs rapportés par les participants, certaines difficultés ou inconvénients ont également été signalés : La pression du temps est toujours forte, les programmeurs 3 et 4 regrettent qu'avec des cycles courts il n'y ait pas de temps pour la réflexion ou pour appliquer les bonnes pratiques. « *vu les délais qui*

sont accordés on n'a pas le temps de la réflexion et plus pour respecter les bonnes pratiques. »(programmeur 3). Selon le programmeur 3, cela génère une frustration « *par rapport aux objectifs annoncés en début de cycle et ce que l'on réalise réellement* ».

Le directeur technique (CTO) a déclaré que les tickets ne sont pas toujours faciles à découper et à planifier dans le cycle de développement. Un autre point est que les livraisons ne sont pas liées à des cycles, ainsi les programmeurs et analystes regrettent que « *nous n'ayons aucune visibilité sur la version pour laquelle un ticket donné doit être traité.* » (programmeur 7).

Impact de la COVID-19 Les personnes interrogées n'ont pas soulevé de problème particulier sur le cycle de développement en raison du travail à distance.

5.3.4 Gestion de code source

La société CIM à introduit un système de contrôle de version, SVN, pour mieux gérer son code source.

Après 18 mois d'utilisation, il y a 4639 commits dans le dépôt SVN et les programmeurs perçoivent maintenant presque unanimement comme une étape positive. « *C'est plus rassurant* »(programmeur 11).

Par ailleurs, comme il fallait s'y attendre, les programmeurs étaient méfiants et peu enclins à changer leurs habitudes de travail. Le programmeur 5, qui a une connaissance préalable de SVN, s'est fait le champion de la technologie en aidant ses collègues. Le changement des habitudes de travail a été difficile à mettre en place : « *Il a généré des frustrations, certaines personnes étaient réticentes [. . .] cela n'a pas facilité la transition* »(programmeur 11).

Le principal inconvénient est et reste la résolution des conflits dans le code. PowerBuilder stocke le code source dans un format binaire (fichiers *PBL*) alors que les outils de contrôle de version ont habituellement besoin de texte. Cela oblige les programmeurs à commiter à la fois la forme binaire (*PBL*) et la forme textuelle du projet dans SVN. La forme binaire est nécessaire pour pouvoir réimporter une version dans l'IDE. La forme textuelle est nécessaire pour les capacités de fusion. Ceci est souvent source de conflits.

La gestion des branches dans SVN n'est pas considérée par le directeur technique (CTO) comme aussi avancée que dans d'autres systèmes de contrôle de version tels que Git. L'organisation actuelle est d'avoir une seule branche (par version livrée du produit). Cela implique que tous les commits sont poussés vers la même branche et doivent parfois être supprimés ultérieurement si le commit ne fait pas partie d'une livraison. Cela entraîne un travail manuel supplémentaire. La solution envisagée à ce problème est de passer prochainement à la gestion du contrôle de

version Git. Cela permettra d'isoler les tickets dans des branches spécifiques et devrait faciliter le travail d'intégration.

Impact de la COVID-19 Les personnes interrogées n'ont pas vu de contribution spécifique de SVN au travail à distance. Il semblerait naturel que l'absence de SVN aurait rendu plus difficile la synchronisation du travail entre les programmeurs, mais ils ont explicitement déclaré qu'ils ne le voyaient pas de cette façon.

5.3.5 Qualité de code

La direction considère la révision du code en équipe (voir le Chapitre 3) comme un moyen d'obtenir la propriété de l'équipe sur l'ensemble de la base de code (par opposition à la propriété de l'auteur sur son propre code).

Tous les programmeurs ne semblent pas partager entièrement cette compréhension. Les nouveaux arrivants le reconnaissent, un programmeur junior (le programmeur 1) regrette que « *parfois les revues de code globales, qui a lieu chaque deux semaines, soient annulées lorsqu'il y a une livraison urgente à assurer* » et qu'« *il devrait y en avoir plus* (hebdomadaires) ». Mais les programmeurs plus expérimentés proposent d'en avoir moins (mensuelles). Leur argument est qu'il est difficile de trouver quelque chose d'intéressant à montrer à l'équipe toutes les deux semaines. Les programmeurs rapportent que « *nous ne pensons pas toujours à regarder un morceau de code pour une revue de code lorsque nous codons, parce que nous sommes souvent pressés* »(programmeurs 2). Même si un programmeur chevronné reconnaît que cela lui a « *permis de réutiliser une fonction au lieu de la recréer* » à une occasion (programmeur 11). Une autre critique est que ces revues sont orales et que les gens oublient les informations partagées après un certain temps.

Un programmeur a suggéré de garder une trace écrite de ces revues.

Une autre proposition pour renouveler l'intérêt serait d'organiser les revues de code de l'équipe par domaine d'activité ou par domaine de qualité du logiciel (optimisation, lisibilité ...), ou d'avoir des exercices sur lesquels tout le monde travaillerait ensemble (dojo de programmation).

Le directeur technique (CTO) est satisfait de la pratique actuelle, estimant que « *la qualité globale du code s'améliore* ».

Une autre mesure a été prise pour améliorer la qualité du code : un « *linter* » interne pour certaines règles (voir la Section 5.2) que nous avons développé. Le *linter* s'exécute sur chaque commit et rapporte, par email à l'auteur du commit ainsi que le chef d'équipe de programmeur, les violations ajoutées et supprimées. Depuis son installation au début de l'année 2021 [2], 178 commits ont donné lieu à un

2. Nous n'avons pas pu mettre cette donnée à jours, ayant perdu l'accès au serveur de la société CIM suite à la fin de notre contrat

email avec 48 positifs (violations supprimées) et 130 négatifs (violations ajoutées). Sur la branche principale de développement, il y a eu 76 courriels et le nombre de violations de règles est passé de 1113 à 669 (60%). L'outil a été bien accueilli tant par les programmeurs que par le responsable du développement : « *Ces courriels obligent à corriger petit à petit les erreurs signalées. Chacun fait sa petite part, et c'est moins contraignant. Les courriels positifs après la correction des erreurs sont bons* » (programmeur 1).

Un aspect de cette bonne perception pourrait être que les règles vérifiées sont des règles internes bien connues de tous, mais qui présentent encore de nombreuses violations. Des recherches antérieures [Hora 2012] ont montré que les règles spécifiques sont mieux acceptées que les règles plus génériques. La société a mis en place un outil externe (Visual Expert [3]) plus robuste pour vérifier les règles de programmation. La question de savoir s'il sera en mesure de vérifier les règles internes devrait être un point important dans la décision.

Impact de la COVID-19 Là encore, le principal impact est que les réunions de révision globale du code se tiennent à distance. Cela est généralement considéré comme un inconvénient, mais aucun problème particulier n'a été soulevé au cours des entretiens.

COVID-19 n'a pas eu d'impact sur le « *linter* » interne, d'une part parce que cette action a été initiée en 2021, et d'autre part parce qu'il est entièrement automatisé (lors du commit dans le dépôt SVN) et communique les violations ajoutées ou supprimées aux programmeurs par email.

5.4 Résumé du chapitre

Ce chapitre présente une étude qualitative de l'introduction de plusieurs pratiques recommandées dans le développement agile dans la société CIM. Parmi les pratiques introduites, certaines ont présenté des difficultés, d'autres ont été bien perçus. Ainsi, nous avons vu que l'introduction d'un outil de gestion des versions (SVN) est perçue positivement, mais la migration a été difficile et a demandé plus d'efforts que les autres nouvelles pratiques, car les programmeurs n'avaient pas cette culture. Il faut noter que la technologie utilisée (PowerBuilder) crée des difficultés supplémentaires, car le code source est stocké dans un format propriétaire et n'interagit pas facilement avec les systèmes de contrôle de version. Nous avons vu que bien que les réunions quotidiennes *Stand-up meeting* sont perçues comme une bonne pratique, mais le travail à distance, à cause de la COVID-19, a eu un impact négatif sur cette pratique et en a réduit les avantages. Nous avons vu que l'« *équipe run* » (spécialisée dans la réponse aux demandes urgentes) a reçu des avis mitigés.

3. https://www.visual-expert.com/

Il est reconnu qu'elle facilite la planification, mais les programmeurs n'aiment pas en faire partie. Par conséquent, elle a été fermée. Nous avons aussi vu que l'outil d'analyse statique du code ou « *linter* » a été bien accueilli par tous. La réaction positive pourrait être liée au fait qu'il vérifie les règles internes qui sont plus adaptées à la réalité dans la société contrairement aux règles génériques de qualité du code.

Dans le Chapitre 6, nous allons discuter l'introduction des tests unitaires qui est une suite de la modernisation des pratiques par la société CIM.

Chapitre 6
Tests et exécution automatique

Sommaire

6.1 Introduction 71
6.2 Solutions existantes 72
6.3 Problèmes avec la pratique de tests 75
6.4 Piste d'améliorations 81
6.5 Résultat des tests unitaires 88
6.6 Conclusions du chapitre 89

6.1 Introduction

Dans les chapitres précédents, nous avons passé en revue le contexte de cette thèse avec la description de pratiques de développement logiciel qui n'étaient pas au niveau des pratiques recommandées actuellement, et un environnement technique (PowerBuilder) propriétaire et lui aussi en retard par rapport aux normes actuelles.

Nous avons vu quelles pratiques la société CIM a déjà commencé à améliorer et nous avons évalué l'impact de ces changements. Il a cependant une pratique commune que nous n'avons pas encore évoquée : l'utilisation de tests unitaires automatisés.

Les tests automatisés sont connus pour améliorer la qualité de code [Rafi 2012]. Dans la continuité de la modernisation des pratiques de développement, la société CIM pense à améliorer sa pratique des tests par l'introduction de tests unitaires et l'amélioration des tests fonctionnels existants. Par ailleurs, la mise en place de tests unitaires dans une équipe qui n'en a pas l'habitude n'est pas sans problèmes.

Dans ce chapitre, nous allons présenter les problèmes et les pistes de solutions que nous avons trouvées pour mettre en place les tests à la société CIM. Pour cela, nous allons présenter les premières pistes de solutions à la mise en place des tests basées sur les offres du marché dans la Section 6.2. Les premières solutions n'ayant pas donné entièrement satisfaction, nous poursuivrons avec les problèmes liés à la mise en place des tests dans la Section 6.3. Ensuite, nous présenterons les améliorations que nous avons apportées, Section 6.4. Finalement, nous présenterons

succinctement les premiers résultats de nos travaux, Section 6.5, avant de clore le chapitre dans la Section 6.6.

6.2 Solutions existantes

La société CIM utilise déjà une pratique semi-automatique de tests pour les tests fonctionnels. Mais cette pratique présente plusieurs défauts. Afin d'apporter des améliorations pour l'exécution automatique des tests sur Izy Protect, il est important de présenter les solutions déjà existantes. Dans cette optique, nous allons présenter cet aspect sur les tests fonctionnels et sur les tests unitaires.

6.2.1 Test fonctionnel sur Izy Protect

Certaines parties d'Izy Protect sont identifiées comme étant source de nombreux défauts, sur la base des années d'expérience et des rapports de défauts. Par conséquent, la société a mis en place des tests fonctionnels sur ces parties du code d'Izy Protect. Il existe deux types de tests fonctionnels mis en place à cet effet : (1) appel en ligne de commande des fonctionnalités de base ; (2) simulation des interactions de l'utilisateur avec TOSCA.

Les tests fonctionnels sont exécutés par l'équipe de testeurs. Pour les exécuter sur une version d'Izy Protect qui vient du développement, l'équipe de testeurs dispose d'une version de référence d'Izy Protect. Le testeur exécute d'abord la version de référence d'Izy Protect, et enregistre les résultats. Ensuite, le testeur exécute la version d'Izy Protect à tester, puis enregistre les résultats. Enfin, le testeur compare manuellement les résultats. C'est une tâche difficile, car elle nécessite la comparaison de milliers de lignes de données. Il faut noter que l'exécution des tests fonctionnels requiert une durée de 6 jours au minimum.

Le temps d'exécution fait que parfois, sous pression, la société livre une nouvelle version au client en se basant seulement sur les tests manuels effectués par les auteurs des modifications pour respecter les délais de livraison. Ainsi, cette version peut présenter des défauts chez le client. Ces défauts peuvent être rapportés par le client des mois après le développement. Le programmeur affecté pour corriger le défaut devra alors prendre le temps de comprendre le code, identifier la source du défaut et le corriger. Cette manœuvre peut nécessiter des jours.

Dans le cas où les tests fonctionnels détectent un défaut dans une nouvelle version d'Izy Protect, le défaut est identifié par rapport à une ou plusieurs fonctionnalités plutôt qu'à une ou plusieurs classes ou méthodes. Par conséquent, les retours des tests fonctionnels ne permettent pas toujours au programmeur de savoir directement quelles classes ou méthodes sont source de défaut.

6.2.2 Cadriciel (*framework*) de test unitaire

Pour écrire des tests unitaires pour Izy Protect, il faut un cadriciel (framework) de tests unitaires pour PowerBuilder. Mais ce langage de programmation ne possède pas de cadriciel de tests unitaires officiel.

La communauté a développé un cadriciel inspiré de *xUnit* [Hamill 2004] appelé *PBUnit*. Le cadriciel *PBUnit* permet d'écrire, d'exécuter et de rapporter les résultats des tests unitaires en PowerBuilder. *PBUnit* a cessé d'être maintenu en 2015. Il y a eu une tentative en 2020 pour relancer et améliorer les assertions de test *PBUnit* [1]. Cette version comporte des bibliothèques dynamiques PowerBuilder (fichiers d'extension *.PBD* que nous avons déjà présenté dans le Chapitre 3) qui ne fonctionnent pas sur la version 2017 de PowerBuilder. Étant donné qu'Izy Protect est implémenté sur la version 2017 de PowerBuilder, nous ne pouvons pas adopter cette version de *PBUnit* pour écrire les tests pour Izy Protect. Donc nous avons adopté la version 2015 de *PBUnit*.

PBUnit comporte deux parties : (1) une bibliothèque des tests unitaires ; (2) une interface graphique *PBUnit*.

La bibliothèque des tests unitaires de *PBUnit* comporte des classes abstraites, en l'occurrence la classe *Testcase*. La classe *Testcase* comporte les différentes méthodes d'assertions des tests en PowerBuilder.

Pour écrire des tests unitaires *PBUnit* pour un projet PowerBuilder, il faut rajouter la bibliothèque de test de *PBUnit* à la liste des bibliothèques du projet PowerBuilder. Ensuite, il faut créer les classes de tests en sous-classant la classe abstraite *Testcase*. Les tests unitaires sont des méthodes des classes de tests. Si le test unitaire est une méthode évènementielle (évènement PowerBuilder expliqué dans le Chapitre 3), *PBUnit* reconnaît automatiquement que c'est un test unitaire et le rajoute à la liste des tests à exécuter. Sinon, il faut rajouter la méthode de test explicitement à la liste des tests à exécuter dans la méthode d'initialisation de la classe de tests.

L'interface graphique de *PBUnit* permet d'exécuter les tests unitaires. La Figure 6.1 montre l'interface de *PBUnit*. Pour exécuter les tests d'un projet PowerBuilder, l'utilisateur a besoin de spécifier le chemin vers l'emplacement de la *Target* du projet PowerBuilder (fichier d'extension *.PBT* présenté dans le Chapitre 3) qu'il souhaite tester. La partie (1) de la Figure 6.1 montre la liste des tests du projet sélectionné. Les tests préfixés avec le symbole ✓ sont les tests qui passent. Tandis que les tests préfixés avec le symbole ✗ ne passent pas. S'il y a au moins un test qui ne passe pas, la classe du test qui ne passe pas et le projet PowerBuilder sont aussi préfixés avec le symbole ✗. Par exemple sur la Figure 6.1 les tests *test_add* et *test_add2* ne passent pas. La partie (2) de la figure montre la liste des tests qui ont échoué et leurs messages respectifs. La partie (3) montre tous les tests exécutés, leur état (passe ou non), ainsi que la durée d'exécution de ces derniers.

1. https ://github.com/a-sokolov-pb/pb-unit-test

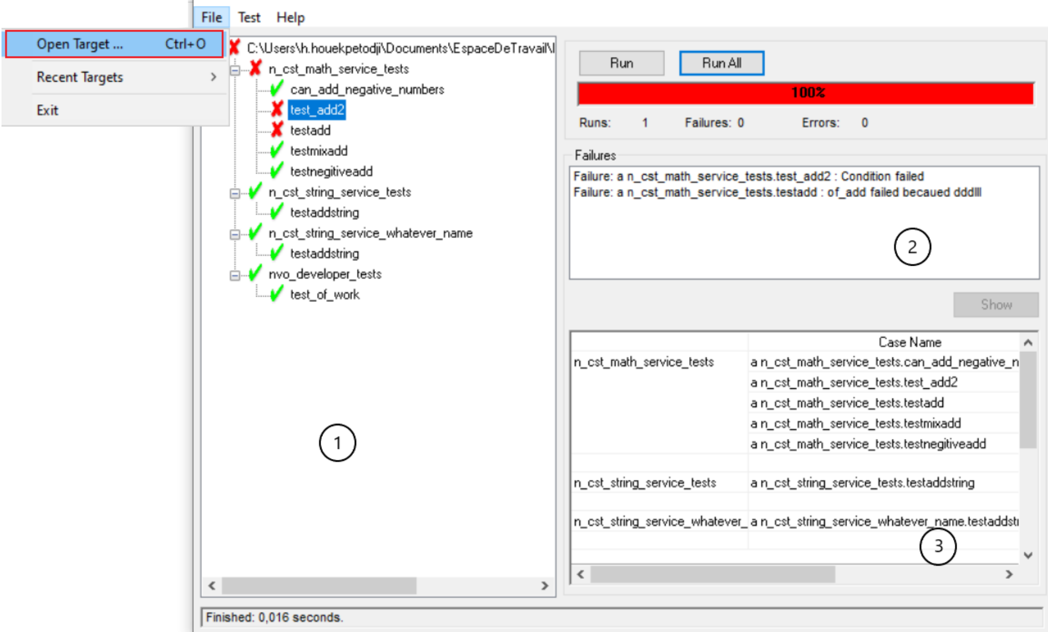


FIGURE 6.1: Interface de *PBUnit*

Comme déjà mentionnée dans le Chapitre 3, pour fonctionner, l'interface de *PBUnit* a besoin de la *Target* (fichier d'extension *.PBT*) avec les bibliothèques (fichiers d'extension *.PBLs*) ou les bibliothèques dynamiques (fichiers d'extensions *.PBDs* générés à partir des *.PBLs* quand on déploie un projet PowerBuilder) d'un projet PowerBuilder. Les tests de *PBUnit* peuvent être exécutés manuellement via l'interface de la Figure 6.1 avec de deux options d'exécutions (Boutons au-dessus de la partie (2)).

— L'option *RunAll :* Permet d'exécuter tous les tests unitaires de la *Target* sélectionnée.

— L'option *Run :* L'utilisateur sélectionne un test unitaire (ou une classe de test) en clinquant sur le test (ou la classe de test) à exécuter dans l'interface de *PBUnit*. L'option *Run* de l'interface de *PBUnit* permet d'exécuter uniquement le test unitaire sélectionné ou bien les tests unitaires de la classe de test sélectionnée.

Les tests unitaires de *PBUnit* peuvent aussi être lancés en invite de commande. Dans ce cas, *PBUnit* exécute automatiquement tous les tests unitaires en enregistrant les tests exécutés et leurs résultats en mémoire. À la fin de l'exécution, *PBUnit* présente un rapport des tests exécutés.

6.3 Problèmes avec la pratique de tests

Ayant présenté les pratiques de test déjà existantes dans société CIM, nous allons maintenant voir les problèmes que ces pratiques soulèvent.

La société CIM envisage deux formes de tests pour améliorer la qualité d'Izy Protect : introduction de tests unitaires, amélioration des tests fonctionnels. Pour l'instauration de tests unitaires, des problèmes peuvent venir du logiciel lui-même (ex. : couplage fort entre unités à tester), ou de son environnement aussi bien technique qu'humain : « *Dans les organisations,* [les coûts des tests] *sont influencés par les problèmes tels que les propriétés des systèmes à tester, les attitudes des employés, les limitations de ressources et les clients* » [Rafi 2012, Taipale 2011].

Dans cette section, nous allons passer successivement en revue les problèmes liés au logiciel Izy Protect lui-même, à l'environnement de programmation PowerBuilder et aux développeurs.

6.3.1 Problèmes techniques liés à Izy Protect

Pour l'introduction de tests unitaires, actuellement non existants, le propre logiciel Izy Protect pose des difficultés particulières que nous allons voir maintenant. Ces difficultés sont : (i) grande taille de certaines classes (« *God class* ») ; (ii) couplage fort entre certaines classes ; (iii) une classe particulière de stockage de données ; (iv) non séparation du code de calcul et d'interface graphique ; (v) méthodes avec beaucoup de chemins d'exécution ; (vi) effets de bord ; (vii) gestion des exceptions ; et finalement (viii) gestion de la mémoire. Ici, nous allons détailler ces problèmes.

(i) Grandes classes dans Izy Protect

Izy Protect comporte plusieurs classes utilitaires (ex : *wa_dw_simple*, *w_utile*, *uo_traitement*, et *uo_datawindow*) de grande taille. Ainsi, *wa_dw_simple* est d'environ 800 lignes de code, *w_utile* a plus de 2000 lignes de code, *uo_traitement* a plus de 3000 lignes de code et la dernière, *uo_datawindow* a plus de 7000 lignes de code. Ces classes cumulent chacune plusieurs responsabilités de l'application. Elles correspondent au *code smell* « *God class* » et sont, par conséquent, difficiles à comprendre. De plus, ce sont des classes abstraites dont héritent la majorité des autres classes d'Izy Protect. Elles « contaminent » donc tout le code le rendant difficile à comprendre et aussi à tester, puisque la même classe remplit plusieurs fonctionnalités. Il est difficile de tester une fonctionnalité séparément, car elle peut avoir des interactions avec plusieurs autres à l'intérieur de la même classe.

(ii) Couplage fort dans Izy Protect

Ces mêmes grandes classes, listées ci-dessus, ont aussi un fort couplage entre elles. La Figure 6.2 présente les interdépendances entre ces classes qui s'utilisent ou héritent toutes les unes les autres.

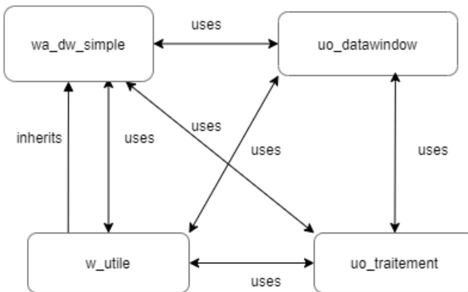


FIGURE 6.2: Dépendance entre des classes abstraites d'Izy Protect.

De nouveau, ce problème est hérité par la majorité des autres classes du système. Ce couplage fort empêche de tester les classes séparément. Pour tester l'une d'elles, il faut créer toutes les autres classes dont elle dépend.

(iii) Classe de données

Beaucoup de données de l'application sont gardées dans une unique instance de la classe *w_utile*, déjà citée ci-dessus. L'instance de cette classe est accessible par une variable globale utilisée un peu partout dans l'application.

Ceci correspond au *code smell* « *Data class* ». D'après [Fowler 2018, Lanza 2006], cette pratique réduit la compréhensibilité, la testabilité et la maintenabilité du code.

Pour les tests unitaires, la difficulté est d'instancier correctement cette classe et d'y stocker les bonnes données pour pouvoir faire n'importe quel test d'une autre classe (du fait du « Fort couplage » déjà cité).

D'autres exemples similaires sont les sous-classes de *Structure*. La classe PowerBuilder *Structure* (voir Section 3.2, page 24) sert à définir des structures de donnée du type des *struct* C. Ses sous-classes sont utilisées dans Izy Protect en paramètre de nombreuses méthodes. Le problème est que ces structures centralisent les paramètres de plusieurs méthodes à la fois. Par exemple, la structure *str_cot_cal* comporte 240 attributs et elle est passée en paramètre de 155 méthodes différentes. Pour tester une méthode qui utilise *str_cot_cal*, il faut identifier dans l'ensemble de ses 240 attributs ceux qui sont nécessaires pour invoquer la méthode et potentiellement toutes les méthodes qu'elle-même appelle récursivement.

(iv) Non-séparation du code de calcul et d'UI

Trois des grandes classes citées ci-dessus devraient avoir des objectifs bien définis : *wa_dw_simple* devrait être la super classe de toutes les fenêtres, *uo_datawindow* doit être en charge de l'accès aux données dans la base de données, et *uo_traitement* devrait s'occuper des calculs (traitements métier).

Mais dans le code cette séparation n'est pas très bien respectée. Par exemple, il y a des méthodes dans les sous-classes de *uo_traitement* qui ouvrent des fenêtres de type *popup* pour informer l'utilisateur (notamment en cas d'erreur).

De la même façon, il y a des méthodes dans les sous-classes de *wa_dw_simple* qui font des traitements métiers comme c'est souvent le cas lors d'utilisation d'outils de création d'interface graphique par *drag-and-drop* de *widget* [2].

Pour les tests unitaires, cela implique que des méthodes de traitement métier ouvrent des fenêtres, ce qui complique les interactions machine-machine et est donc un problème pour l'automatisation des tests. « *Ceci est particulièrement un problème pour les interfaces qui ne sont pas conçues pour les interactions machine-machine, comme les interfaces utilisateur graphiques* » [Wiklund 2017].

(v) Méthodes avec beaucoup de chemins d'exécution

Outre le problème de *God class* déjà cité, Izy Protect présente aussi un problème de *God method* : des méthodes de grande taille, avec de nombreux chemins d'exécution (forte complexité cyclomatique de [McCabe 1976]). Par exemple, certaines méthodes implémentent une sorte de gros *switch case* (implémenté en fait par une succession de conditions *if/then/else*) où chaque condition appelle une fonctionnalité différente.

La Figure 6.1 montre un extrait du code d'une méthode d'environ 600 lignes et d'une complexité cyclomatique de 123. Dans cet extrait, les lignes 1, 10 et 21 montrent le choix de traitements différents en fonction des valeurs des trois variables « principales » *is_rdg_arr_mul*, *as_cal.d_cal_deb* et *ll_annee_inversion_taxes_remises*. Ces choix sont ensuite raffinés par le test de variables « secondaires » aux lignes 2, 4, 11, 12, 14, 23 et 24. En tout, sur ce court extrait, le choix des traitements effectués dépend des valeurs de 11 variables différentes.

```
1 IF is_rdg_arr_mul <> "O" OR year(as_cal.d_cal_deb) <
     ll_annee_inversion_taxes_remises THEN
2   if as_cal.s_mt_regul <> "" and not isnull(as_cal.
     s_mt_regul) and ib_recalcul_regul then
3     ld_regul= uf_decimal(as_cal.s_mt_regul)
4     if ld_regul <> 0 then
```

2. Composants graphiques utilisés pour créer des interfaces utilisateur.

```
5       ad_mt = ad_mt  + ( ld_regul / 12 )
6     end if
7   end if
8 END IF
9
10 IF is_rdg_arr_mul <> "O" OR year(as_cal.d_cal_deb) <
     ll_annee_inversion_taxes_remises THEN
11   if is_ges_sec = "Z" then
12     if id_zon_coe <> 0 then
13       ad_mt = ad_mt * id_zon_coe
14       if is_rdg_arr_mul = 'O' then
15         uf_arrondi(ad_mt,FALSE)
16       end if
17     end if
18   end if
19 END IF
20
21 IF is_rdg_arr_mul <> "O" OR year(as_cal.d_cal_deb) <
     ll_annee_inversion_taxes_remises THEN
22
23   if as_cal.s_exo_taxe = "O" and not isnull( as_cal.
     s_exo_taxe) and ib_appl_tax then
24     if is_rdg_mul_tax = "O" then
25       ad_mt = uf_calcul_multi_taxes(as_cal, as_app, ad_mt)
26
27     else
28       ad_mt = uf_calcul_taxe(as_cal, as_app, ad_mt)
29     end if
30   end if
31 END IF
```

Code source 6.1: Exemple de *switch case*

Notez que dans cet exemple, la variable *as_cal* contient une instance de la structure *str_cot_cal* (décrite plus haut dans le problème « Classes de données ») qui contient 240 attributs différents.

(vi) Effets de bord

Dans le code d'Izy Protect, la convention est que les méthodes retournent un code d'erreur, ou 1 signifie habituellement que l'exécution s'est bien passée, alors qu'une valeur nulle ou négative indique habituellement un code d'erreur. Les retours des méthodes ne sont donc pas des résultats de calcul métier.

On en déduit que les résultats des calculs métier sont stockés par des effets

de bord, par exemple dans des classes de données telles que *w_utile* ou l'une des sous-classes de *Structure*. Pour les tests unitaires, cela signifie que l'on ne peut pas simplement tester le retour des méthodes, mais qu'il faut aussi identifier les différents effets de bord pour vérifier qu'ils ont eu lieu correctement.

(vii) Gestion des exceptions

PowerBuilder repose beaucoup sur le principe des exceptions pour gérer les erreurs à l'exécution. Ces exceptions peuvent être capturées à l'aide d'instructions *try/catch*.

Ceci résulte en deux types de problèmes : les exceptions sont parfois gérées par la création d'une fenêtre *popup* pour informer l'utilisateur, ou les exceptions ne sont pas capturées et provoquent l'arrêt intempestif de l'application.

Dans le cas de l'affichage d'une fenêtre *popup* pour informer l'utilisateur de l'erreur, un opérateur humain est nécessaire pour refermer cette *popup*, ce qui est incompatible avec l'exécution de tests automatisés. Ce problème a déjà été évoqué pour le problème de « Non-séparation du code de calcul et d'UI ».

Mais il faut aussi noter que dans Izy Protect les blocs de capture des exceptions ne sont pas suffisamment nombreux et plusieurs exceptions ne sont pas capturées, provoquant l'arrêt anormal de l'application.

(viii) Gestion de la mémoire d'Izy Protect

Le dernier problème technique concernant Izy Protect est la gestion de la mémoire. Dans Izy Protect la libération de la mémoire utilisée par un objet doit être faite par les développeurs dans des destructeurs de chaque classe. Il n'y a pas de *gerbage collect* comme en Java. Dans certains destructeurs, la destruction est conditionnée à la non-exécution d'un autre traitement et il arrive que la condition ne soit pas vérifiée. Par conséquent, il y a des objets qui ne sont pas détruits, créant ainsi une « fuite de mémoire ».

Nous avons vu que pour tester une méthode, plusieurs grosses classes peuvent être nécessaires (instanciées). Aussi, les propres méthodes testées peuvent instancier d'autres classes lors de leur exécution. Enfin, après un test, la mémoire peut ne pas être libérée, car le destructeur détecte l'exécution d'un traitement qui empêche cette libération.

Du fait de la façon dont les tests sont exécutés (voir Section 6.2.2), le résultat est qu'il y a de grosses « fuites de mémoire » lors de l'exécution des tests, l'utilisation de la mémoire grossit au fur et à mesure de l'exécution de tous les tests et très rapidement (quelques dizaines de tests) provoque un dépassement de capacité de la machine qui arrête toute exécution.

Ceci est bien sûr un problème lors de l'exécution automatique de tests qu'on espère nombreux pour une application de la taille et de l'importance d'Izy Protect.

6.3.2 Problèmes techniques liés à PowerBuilder

Pour l'exécution automatique d'une banque de tests unitaires, nous affrontons aussi des problèmes techniques liés à l'environnement de programmation de PowerBuilder lui-même : (ix) obtention automatisée du code source à jour, et (x) utilisation d'un analyseur de code source.

(ix) Mise à jour de code source

Pour que l'exécution automatique des tests unitaires détecte des défauts, il faut que le code testé soit à jour. En d'autres termes, il faut mettre à jour les bibliothèques (fichiers d'extension *.PBL*) ou les bibliothèques dynamiques (fichiers d'extension *.PBD*) automatiquement sans intervention humaine. Il est possible de mettre à jour les fichiers *.PBL*s d'un projet PowerBuilder à partir de la dernière version du code source des classes, des anciennes versions des fichiers *.PBL*s et de la *Target* du projet par trois moyens : (1) l'environnement de développement intégré de PowerBuilder, (2) *Powergen* [3] ou (3) *Orcascr*.

L'utilisation de l'environnement de développement intégré de PowerBuilder est inappropriée, car il nécessite l'intervention humaine.

L'outil *Powergen* est fortement conseillé par la communauté de PowerBuilder pour mettre à jour les fichiers *.PBL*s. Par contre, il est propriétaire. La société n'a pas encore acheté la licence de ce dernier, et nous n'avons pas pu l'utiliser.

Orcascr est un outil natif de PowerBuilder qui permet d'écrire des scripts notamment pour mettre à jour les fichiers *.PBL*s à partir des fichiers sources des classes qu'ils contiennent. Par contre, sur le code source d'Izy Protect, la mise à jour des *.PBL*s échoue sans fournir de message d'erreur. Nous n'avons donc pas trouvé de solution automatique de régénération des *.PBL*s et *.PBD*s.

(x) Analyseur de code PowerBuilder

Nous verrons plus loin qu'une solution envisagée était de générer automatiquement des tests unitaires. Mais pour que cette solution puisse fonctionner, il aurait fallu pouvoir analyser le code source de PowerBuilder (parseur) pour ressorti, pour chaque méthode dont on veut générer les tests unitaires, des informations comme : sa classe, sa signature, les variables d'instances auquel elle accède, sa valeur de retour, etc. Parmi les analyseurs de code libre de PowerBuilder présentés dans le Chapitre 2 (page 10), celui qui donne de résultat d'analyse de code plus détaillé est

3. https ://www.ecrane.com/powergen-overview/

en cours de développement. Mais, il manque des fonctionnalités importantes telles que l’analyse des accès aux variables d’instance des méthodes.

Donc nous ne disposons pas d’un analyseur de code qui nous fournisse de manière fiable, toutes les informations dont nous pourrions avoir besoin.

6.3.3 Problèmes liés aux ressources humaines

Les problèmes d’automatisation des tests ne sont pas seulement liés à Izy Protect. En effet, la mise en place des tests automatisée inclut l’écriture et/ou la génération des tests unitaires. L’équipe des programmeurs d’Izy Protect n’a pas la culture de tests unitaires. Donc, malgré la volonté de la société CIM, la majorité des programmeurs sont réticents à écrire des tests unitaires.

(xi) Culture de tests unitaires

Les programmeurs sont attachés à leur ancienne méthode qui consiste à tester manuellement les modifications sur Izy Protect. Lors des réunions relatives au sujet des tests unitaires dans l’équipe, ils avancent les raisons suivantes : « *On n’a pas le temps. On est tout le temps dans l’urgence* ». « *Les tests unitaires, c’est bien, mais c’est difficile* ». Ces raisons sont compréhensibles, car les programmeurs n’ont pas une culture préalable des tests unitaires.

(xii) Ressources de la société CIM

Comme le montre l’étude du Chapitre 5, malgré les améliorations présentées, la société CIM a encore des difficultés à livrer les clients. De ce fait, la mise en place des tests unitaires soulève les problèmes suivants :

— Comment intégrer cette nouvelle pratique de tests unitaires sans que cela pénalise les délais de livraison ?

— Quel effectif de programmeur faut-il pour la mise en place de cette pratique ?

— Quelle partie d’Izy Protect faut-il tester en premier ?

— Quelle est la stratégie à adopter pour écrire les tests unitaires ?

6.4 Piste d’améliorations

Nous avons maintenant vu les problèmes liés aux tests unitaires automatisés dans la société CIM. Dans cette section, nous allons proposer des pistes de solutions pour ces problèmes. Nous devons toutefois préciser que toutes ces pistes n’ont pas pu être implémentées dans la société.

Pour atteindre les objectifs de la société qui sont d'améliorer la couverture du code source d'Izy Protect, de réduire le coût d'exécution des tests, d'introduire un serveur d'intégration continue pour exécuter automatiquement les tests unitaires et de notifier les défauts aux développeurs et d'apporter un gage de qualité supplémentaire à ses clients, des améliorations sont proposées pour les tests unitaires. Ici, nous allons présenter ces dernières.

6.4.1 Résolution des problèmes

Ici, nous allons aborder les solutions de contournement que nous proposons pour les problèmes présentés dans la Section 6.3. Certaines ont déjà été mises en pratique dans le cadre d'une preuve de concept.

(i) Grandes classes

Comme vu à la page 75, Izy Protect comporte plusieurs classes de grande taille qui sont difficiles à comprendre et à tester. Une restructuration du code pour diminuer la taille des classes est jugée trop coûteuse et trop dangereuse. Donc, nous n'avons pas fait de proposition pour améliorer cette situation.

(ii) Couplage fort et (iii) classe de données

Comme vu, les classes d'Izy Protect sont fortement couplées, ce qui rend leurs instanciations difficiles. À nouveau, une restructuration du code pour diminuer le couple est jugée trop coûteuse et trop dangereuse. La solution choisie est donc de laisser la propre application initialiser elle-même toutes les classes nécessaires comme dans le cas d'une exécution normale. Pour cela, nous avons rajouté la bibliothèque de l'interface graphique de *PBUnit* à Izy Protect ; puis nous avons fait en sorte que les développeurs puissent exécuter l'interface graphique de *PBUnit* à partir de l'interface d'Izy Protect. Ainsi, Izy Protect initialise correctement ces variables globales à son démarrage, permettant aux tests d'utiliser correctement les classes à tester.

(iv) Non-séparation du code de calcul et d'UI

La séparation entre le code de calcul et UI n'est pas souvent respectée dans Izy Protect. Dans l'esprit d'automatiser l'exécution des tests unitaires, nous proposons dans un premier temps de tester uniquement du code qui n'ouvre pas de fenêtres *popups* ou de formulaires en cours d'exécution. Cela évite que les tests s'arrêtent en cours d'exécution automatique et attendent une intervention humaine pour continuer l'exécution. L'espoir est qu'une fois la culture de tests unitaires comprise et

établie, le code soit petit à petit restructuré pour permettre de tester toutes les fonctionnalités. Cette restructuration pourrait se faire par exemple lors de modifications du code pour apporter des nouvelles fonctionnalités ou corriger des défauts.

(v) Méthodes avec beaucoup de chemins d'exécution

Izy Protect comporte des grandes méthodes qui sont souvent de gros *switch case* où chaque cas implémente une fonctionnalité (voir page 75). Pour les tester, nous proposons d'écrire des tests unitaires par condition. Ainsi, pour chaque test, on fournirait les paramètres pour qu'une seule condition d'exécution de la méthode testée soit vérifiée. L'avantage de cette approche est de simplifier les tests des grandes méthodes à une seule fonctionnalité à chaque fois.

(vi) Effets de bord

Comme nous l'avons déjà présenté (page 78), les retours des méthodes ne sont pas des résultats de calcul métier, mais plutôt un code d'erreur (ex : 1 = « retour sans erreur »). Pour connaître les effets de bord d'une grande méthode que nous testons, il faut lire le code et l'analyser pour comprendre toutes ses implications. La connaissance du code par les programmeurs peut aider à mieux cibler la lecture et identifier plus rapidement les effets de bord. On peut aussi utiliser l'outil de débogage de PowerBuilder pour suivre pas à pas une exécution réelle. Il faut noter que ce travail d'analyse est difficile, car pour une grande méthode cela peut prendre plusieurs heures pour comprendre ses effets de bord et pouvoir proposer un premier test unitaire.

Dans les cas où nous n'arrivons pas à comprendre le fonctionnement d'une grande méthode, nous proposons d'écrire des tests unitaires de fumée (*Smoke test*) qui sont des tests unitaires qui permettent de tester simplement que la méthode s'exécute sans erreurs. Il faut noter que la justesse du résultat de l'exécution n'est pas testée. Pour améliorer la couverture des tests, nous proposons de générer des tests unitaires de fumée que nous allons détailler dans la Section 6.4.3 (page 87).

(vii) Gestion des exceptions

Nous avons vu que la gestion des exceptions dans Izy Protect pose de problèmes pour l'automatisation des tests unitaires. Pour remédier à ces problèmes, nous utilisons un bloc *try/catch* dans le corps des tests. En cas d'exception ou d'erreur, le test échoue en affichant l'exception ou l'erreur qui est survenue dans le message d'échec du test. Cela fait qu'il n'a pas de fenêtre de *popup* nécessitant un opérateur humain pour le fermer. Ce qui facilite l'automatisation des tests unitaires.

(viii) Gestion de la mémoire d'Izy Protect

Nous avons vu que l'exécution des tests peut rapidement saturer la mémoire de la machine.

Pour contrer le problème de gestion de mémoire d'Izy Protect, dans le cadre de l'exécution manuelle des tests, nous recommandons d'utiliser seulement l'option *Run* de *PBUnit*. Dans le cadre de l'exécution des tests en invite de commande, nous avons apporté quelques modifications à *PBUnit* :

- Au lieu d'enregistrer les tests exécutés et leurs résultats en mémoire, nous avons connecté une base de données à *PBUnit*. Nous avons modifié *PBUnit* afin qu'il enregistre directement les résultats de tests dans la base de données.
- Nous avons modifié *PBUnit* pour que son option *RunAll* exécute seulement les tests de la classe de tests passés en paramètres (l'option *Run* de *PBUnit* n'est pas disponible en invite de commande).
- Nous avons développé un outil *PWBCI* pour exécuter tous les tests d'Izy Protect avec *PBUnit*. Nous allons aborder *PWBCI* plus en détail dans la Section 6.4.2.

(ix) Mise à jour de code source

Comme nous l'avons déjà expliqué dans la Section 6.3.2 (page 80), il est difficile de mettre à jour automatiquement les *PBLs* et *PBDs* d'Izy Protect. En fin de journée, un développeur de la société CIM met déjà à jour Izy Protect, en régénérant manuellement l'exécutable et les fichiers *.PBDs* à partir du code à jour. Nous proposons simplement d'utiliser cette version d'Izy Protect à jour pour l'exécution automatique des tests unitaires.

(x) Analyseur de code PowerBuilder

Nous avons vu qu'une solution proposée pour aider les programmeurs de la société à écrire les tests unitaires est de générer les tests unitaires (page 83). Nous avons aussi mentionné que l'analyseur de code de PowerBuilder n'est pas complet (page 80). Dans ce sens, pour chaque méthode dont on veut générer des tests unitaires, nous limitons l'analyse de code à la signature de la méthode, la classe à laquelle elle appartient ainsi que ses différentes valeurs de retour.

(xi) Culture de tests unitaires

Pour instaurer une culture des tests unitaires dans l'équipe de programmeurs de la société CIM, nous avons écrit une documentation à suivre par les programmeurs pour écrire les tests sur Izy Protect (voir annexes A). Ensuite, nous avons eu deux

réunions d'une heure chacune avec les programmeurs pour les former sur le sujet des tests unitaires. Durant ces réunions, nous leur avons montré aussi des exemples de tests sur Izy Protect que nous avons déjà écrits et présentés comment écrire les tests.

Après ces réunions de présentation, nous avons continué à écrire les tests dans l'idée que ces tests capturent des défauts afin de démontrer leur utilité à l'équipe de programmeurs. Ensuite, une réunion hebdomadaire de deux heures a été mise en place pour écrire des tests avec des programmeurs de la société. L'objectif étant que ces programmeurs une fois montés en compétences forment progressivement les autres programmeurs.

(xii) Ressources de la société CIM

Nous avons vu que la mise en place des tests unitaire demande des programmeurs que la société CIM doit efficacement partager entre l'écriture de tests unitaires et d'autres tâches afin de continuer à livrer ses clients (page 81). Ainsi, deux programmeurs de la société sont chargés d'assister aux réunions de tests unitaires hebdomadaires.

Afin que la société CIM et ces développeurs voient la valeur des tests unitaires, nous proposons de mettre la priorité sur les parties d'Izy Protect qui sont souvent source de défauts. Ainsi, les tests unitaires ont plus de chances de remonter rapidement des défauts, démontrant ainsi leur utilité. Nous avons opté pour l'augmentation progressive de la couverture des tests unitaires sur le code source d'Izy Protect.

6.4.2 Serveur d'exécution automatique des tests unitaires

D'après [Rafi 2012], les tests unitaires doivent être répétables, réutilisables, couvrir tout le code et coûter moins cher en exécution par rapport aux tests manuels. Dans cet esprit, nous avons mis en place un serveur d'exécution automatique des tests unitaires. L'objectif de ce serveur est de vérifier la qualité d'Izy Protect et de notifier l'équipe de programmeurs des éventuelles régressions. Dans cette section, nous allons détailler le fonctionnement du serveur d'exécution automatique des tests unitaires d'Izy Protect.

Comme nous l'avons indiqué dans la Section 6.4.1, nous avons modifié *PBUnit* afin qu'en exécution en invite de commande, il exécute une seule classe de test. Nous avons développé un outil, *PWBCI*, pour exécuter tous les tests unitaires d'Izy Protect. *PWBCI* permet aussi d'envoyer un rapport de l'exécution des tests unitaires à l'équipe de développement comme le montre la Figure 6.3.

Pour exécuter tous les tests unitaires, *PWBCI*, collecte toutes les classes de tests d'Izy Protect. Ensuite, itérativement, *PWBCI* lance *PBUnit* via Izy Protect en lui

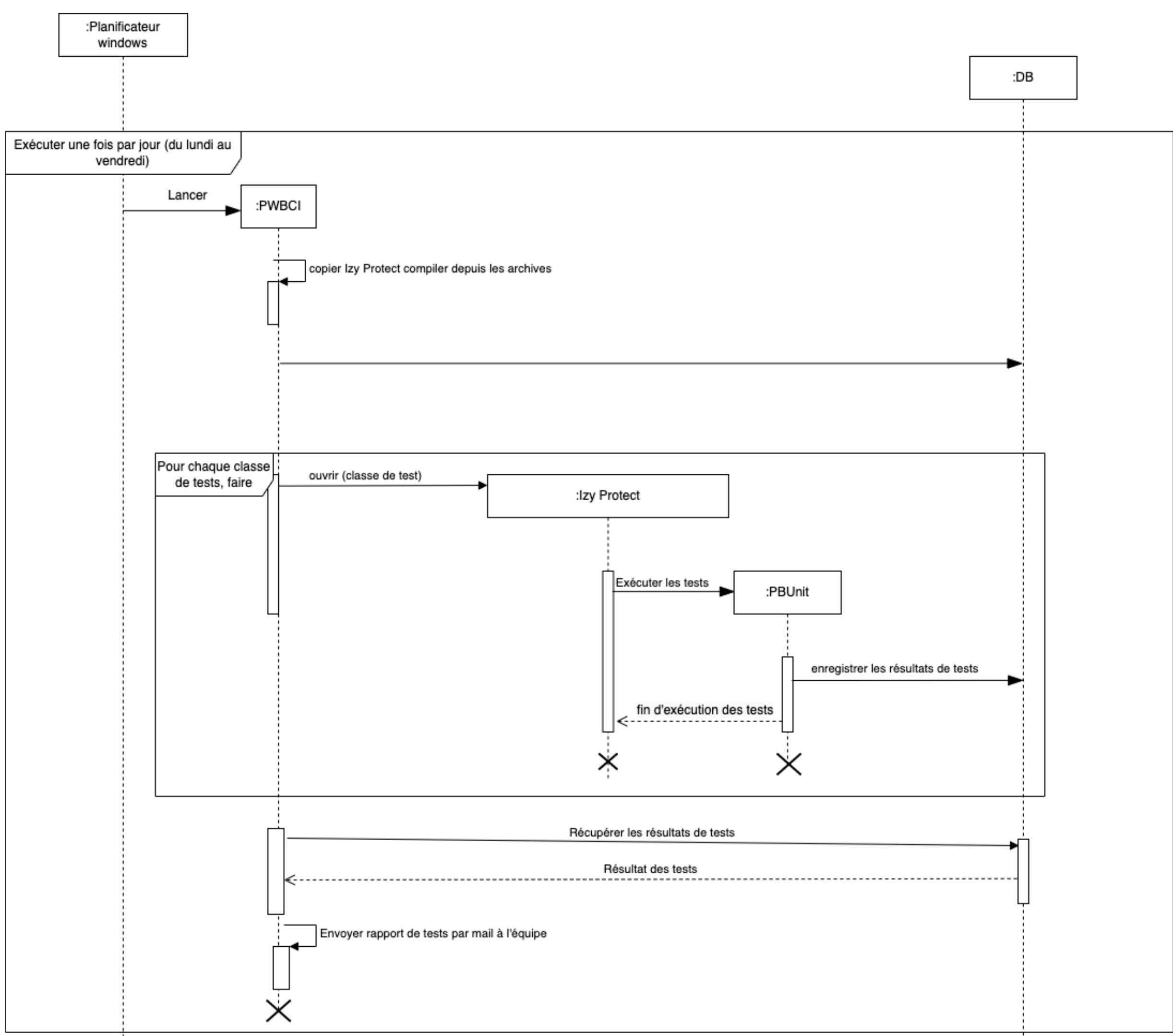


FIGURE 6.3: Diagramme de séquence d'exécution des tests unitaires

passant en paramètre une classe de test à exécuter. *PBUnit* exécute les tests de la classe et enregistre les résultats de ces derniers dans une base de données (Section 6.4.1).

À la fin de l'exécution des tests de la classe de test passée en paramètre, Izy Protect notifie *PWBCI*, et s'arrête. *PWBCI* lance Izy Protect avec la prochaine classe de test, etc.

À la fin de l'exécution de toutes les classes de test, *PWBCI* récupère tous les résultats de tests et envoie un rapport email à l'équipe de programmeurs.

La société CIM alloue un serveur, sur lequel est planifiée une tâche Windows qui permet de lancer *PWBCI*. Ainsi *PWBCI* est lancé une fois par jour (à 22 h) du lundi au vendredi.

6.4.3 Génération des tests unitaires

Izy Protect comporte beaucoup de grandes méthodes. Écrire des tests unitaires pour les grandes méthodes requiert du temps de compréhension et d'écriture des tests unitaires. Pour aider les personnes chargées des tests unitaires dans la société CIM ainsi qu'améliorer la couverture de tests unitaires sur Izy Protect, nous avons souhaité générer automatiquement des tests unitaires pour Izy Protect.

Pour introduire la génération automatique des tests unitaires, nous avons opté pour une approche déterministe (c'est-à-dire n'utilisant pas l'apprentissage automatique). Pour chaque méthode, nous analysons sa signature pour générer des tests de fumée. Comme vu à la page 76 (problème (iii) Classes de données), les paramètres des méthodes d'Izy Protect peuvent être des classes complexes. Ceci rend difficile la génération des tests unitaires. C'est pourquoi nous considérons pour l'instant uniquement les méthodes qui n'ont pas de paramètres.

Nous avons commencé par écrire à la main un certains nombres de ces tests de fumée. Le Code source 6.2 montre un exemple de test de fumée écrit à la main.

```
1  //Déclaration de la classe contenant l'évènement à tester
2   uo_traitement_cot_app luo_traitement_cot_app
3  //Debut du bloc try/catch
4   TRY
5      //Création de la classe contenant l'évènement  à tester
6      w_app_mdi.openUserObject(luo_traitement_cot_app)
7      //Invocation de l'évènement  à tester
8      luo_traitement_cot_app.triggerEvent("tr_annuler")
9   CATCH (runtimeerror er)
10     //faire echouer le test avec message d'erreur
11     this.assert(Er.getmessage(),false)
12  Finally
```

```
13    //Destruction de la classe contenant l'évènement  à
    tester
14  destroy luo_traitement_cot_app
15 END TRY
```

Code source 6.2: Exemple d'un test de fumee

Structure des tests de fumée d'Izy Protect : Après analyse de ces tests, nous avons remarqué une similitude dans leur structure (voir de nouveau le code source 6.2) :

- Déclaration de la classe à tester (ligne 2).
- Début d'un bloc *try/catch* (ligne 4).
- Création de la classe de test (ligne 6).
- Invocation de la méthode à tester (ligne 8).
- Instruction *catch* pour capturer les erreurs et renvoyer les messages d'erreurs (ligne 9).
- Destruction de la classe de test dans la partie *finally* du bloc *try/catch* (ligne 12).
- Fermer le bloc *try/catch* (ligne 15).

Génération de tests de fumée : En nous inspirant, de la structure décrite ci-dessus, nous avons développé un générateur de tests de fumée qui prend en entrée le code source de la classe dont on veut générer les tests. En sortie, le générateur produit un fichier de tests, qui contient des tests de fumée des méthodes de la classe passée en entrée.

La génération suit les étapes suivantes : l'analyseur de code (*parser*) de PowerBuilder analyse le code de la classe pour identifier la signature des méthodes de la classe. Ensuite, cette liste est passée dans le générateur pour créer une classe de tests du nom de la classe passé en entrée suffixé par « *test*". La classe de tests générée contient des tests de fumée qui suivent la structure décrite ci-dessus.

6.5 Résultat des tests unitaires

Dans cette section, nous donnons quelques résultats de nos efforts d'introduction des tests unitaires dans la société. Ces résultats concernent l'écriture manuelle de test unitaire, la génération automatique de tests de fumée, et la couverture de test obtenue.

Pour améliorer la culture de tests unitaires à la société CIM nous avons proposé des formations, des réunions hebdomadaires pour écrire les tests unitaires (Section

6.4.1 page 84) et la génération des tests de fumée (ci-dessus). Au total, nous avons fait 32 réunions hebdomadaires avec deux programmeurs juniors pour la création de test unitaires. Les réunions étaient d'une durée de deux heures. Au cours de ces réunions, nous nous sommes intéressés aux neuf classes les plus défectueuses selon l'historique des tickets et selon l'avis des programmeurs. Nous avons écrit 75 tests unitaires manuellement sur huit classes (Tableau ??). En moyenne, il a fallu environ 51 minutes de travail (à trois) par tests unitaires. Il faut cependant noter que, dans la pratique, il y a eu des réunions où nous lisions uniquement le code sans produire aucun test.

Nous avons généré automatiquement 79 tests de fumée sur six classes. Généralement, les tests de fumée générés automatiquement pour les sous-classes de *User object* d'Izy Protect ne passaient pas. Ils nécessitaient une réécriture. Ceci s'explique par le fait que le fonctionnement des méthodes de ces classes dépend d'autres classes qui doivent être identifiées avec des analyses syntaxiques du code plus avancées. Par contre, pour les sous-classes de *Windows*, seulement quelques tests de fumée ont eu besoin d'ajustements.

	Nombre de classes	Nombre de tests
Tests manuels	8	75
Tests de fumées	6	79
Total	*9*	*154*

TABLE 6.1: Nombre de méthodes testés

Izy Protect comporte au total 17 338 méthodes, ce qui fait que nous avons une couverture de tests de $0,8\%$ qui reste faible. Toutefois, les programmeurs sont maintenant capables d'écrire des tests unitaires tout seuls.

Malgré la faible couverture des tests unitaires, ils ont permis de détecter un défaut dans une grande méthode nommée *uf_detail_cct*. Ce défaut a été détecté avec seulement deux tests unitaires sur cette dernière. Le programmeur qui a introduit le défaut dans la méthode *uf_detail_cct* l'a corrigé.

6.6 Conclusions du chapitre

Dans ce chapitre, nous avons abordé l'introduction des tests sur Izy Protect. Nous avons présenté les défis que nous avons rencontrés et les solutions que nous avons proposées pour mettre en place les tests unitaires sur Izy Protect. De plus, nous avons présenté des améliorations, notamment la génération des tests de fumée, le serveur d'exécution automatique des tests unitaires, qui vont permettre à

la société CIM d'améliorer la couverture des tests unitaires sur Izy Protect ainsi qu'améliorer la qualité de code d'Izy Protect.

CHAPITRE 7
Conclusion

Sommaire

7.1 Synthèse 91
7.2 Contribution 93
7.3 Perspectives et travaux futurs 94

7.1 Synthèse

Cette thèse s'est déroulée dans le cadre d'un partenariat industriel avec la société CIM [1]. La société *CIM* est une SAS au capital social de 200 000 euros détenu à 100% par *DL Software*. Elle est une éditrice, intégratrice, hébergeur et infogérant de solutions pour l'assurance de personnes en santé et prévoyance. Elle offre une expertise en santé et prévoyance acquise après plus de 30 ans auprès de ses clients.

La société a effectué une analyse de risque pour son évolution et sa croissance en 2017, d'où il ressort qu'elle a des problèmes de qualité et de retard de livraison, notamment sur son logiciel principal Izy Protect écrit en PowerBuilder. Ainsi, pour assurer une livraison dans les délais, une qualité logicielle et une flexibilité par rapport aux exigences changeantes, de nombreux éditeurs de logiciels, comme la société CIM par exemple, modernisent leurs pratiques de développement vers les pratiques de développement agiles. Mais, l'existant dans ces sociétés peut être source de difficultés et peut causer l'échec de la modernisation de certaines pratiques. Ainsi, cela soulève deux questions : *l'évaluation et le suivi du processus de modernisation* et *les bénéfices et les défis perçus par les développeurs de la modernisation*.

Chapitre 2 : Présente l'étude de la littérature pour cette thèse. Elle présente en particulier une étude sur les processus de développement logiciel, la qualité logiciel en *SCRUM*, l'étude de la littérature sur les défis de la modernisation des pratiques vers les pratiques *SCRUM*. Le chapitre présente aussi les techniques d'évaluation quantitative et qualitative de processus de développement de logiciels.

1. https ://www.sa-cim.fr/

Chapitre 3 : Décrit le contexte de la thèse en présentant d'abord l'environnement technique PowerBuilder. Ensuite il décrit la société CIM, son processus de développement logiciel avant la modernisation de ses pratiques et son logiciel suivant les recommandations de [Kitchenham 1999].

Chapitre 4 : Détail notre méthodologie d'évaluation et de suivi du processus de développement de la société CIM avant sa modernisation. Dans ce chapitre, nous avons dégagé des indicateurs pour analyser quantitativement les pratiques du processus du développement de la société, puis analyser les données qualitatives des développeurs de la société. Il en ressort que :

- L'organisation d'équipe de la société CIM avant la modernisation, qui était un modèle en V était source de retards de livraison et causait une mauvaise qualité de développement et de test.
- Le cycle de développement logiciel (trois mois) était trop long. Cela faisait que les interruptions et urgences pendant les cycles reportaient les délais de livraison des tickets.
- La gestion informelle de code source rendait le travail collaboratif difficile.
- Les programmeurs n'avaient pas le temps pour tester leurs codes.

Chapitre 5 : Présente les pratiques modernisées par la société CIM ainsi qu'une évaluation que nous avons menée auprès des développeurs sur ces changements. Cette évaluation fait ressortir les avantages et les inconvénients perçus. Nos conclusions sont les suivantes :

- L'introduction d'un outil de gestion de version (SVN) est perçue positivement, mais la migration a été difficile et a demandé plus d'efforts que les autres nouvelles pratiques.
- Il faut noter que la technologie utilisée (PowerBuilder) crée des difficultés supplémentaires, car le code source est stocké dans un format propriétaire et n'interagit pas facilement avec les gestionnaires de contrôle de version orientés fichiers.
- Bien que les réunions quotidiennes de *Stand-up meetings* soient perçues comme une bonne pratique, le travail à distance a eu un impact négatif sur celle-ci et a diminué ses avantages.
- Les cycles de développement de logiciels plus courts et les changements organisationnels de l'équipe sont positifs même avec le travail à distance.
- L'« *équipe run* » (spécialisée dans la réponse aux problèmes à court terme) a reçu des avis mitigés. Elle est reconnue comme facilitant la planification, mais les programmeurs n'aiment pas en faire partie (la composition est tournante). Pour cela, elle a connu un échec et a été fermée.

— Nous avons mis en place un « *linter* » pour envoyer un e-mail aux programmeurs lorsqu'ils committent un code qui viole une règle interne. Il a été bien accueilli par tous. La réaction positive pourrait être liée au fait qu'il vérifie les règles internes par opposition aux règles génériques de qualité du code [Hora 2012].

Chapitre 6 : Présente les problèmes relatifs à la mise en place des tests automatisés dans la société CIM. Ensuite il présente les pistes de solution basées sur les offres du marché puis les améliorations que nous avons apportées. Nos conclusions sont :

— L'introduction des tests unitaires est perçue positivement, mais la société CIM dispose de ressources limitées pour sa mise en œuvre.

— Le logiciel Izy Protect présente des problèmes comme les grandes classes, les grandes méthodes, le couplage fort entre les classes, la gestion des exceptions, etc. qui rendent difficile la mise en place des tests unitaires.

— L'absence de culture de tests unitaires dans l'équipe de programmeurs a aussi été un frein pour les tests unitaires.

— Nos différentes propositions comme la génération des tests de fumée, adaptation de *PBUnit* à Izy Protect, etc. ont permises d'écrire des tests.

— Les formations et les réunions de tests unitaires ont permis à certains programmeurs de savoir écrire des tests unitaires pour Izy Protect.

7.2 Contribution

La principale contribution de cette thèse est de présenter une étude empirique sur l'introduction de quelques pratiques recommandées dans les modèles de développement agiles. Ainsi :

— Nous documentons la perception industrielle de certaines pratiques de développement agile ;

— Nous documentons les avantages et les inconvénients perçus de ces pratiques ;

— Nous identifions l'impact du travail à distance sur certaines pratiques agiles.

— Nous identifions l'impact de certaines technologies comme PowerBuilder sur l'adoption de pratique moderne de développement. Malheureusement, ces technologies ne peuvent pas être facilement abandonnées à cause de l'importance du code existant.

7.3 Perspectives et travaux futurs

Dans cette section, nous présentons les questions ouvertes non traitées dans la thèse. Ces questions ouvertes offrent des opportunités pour poursuivre nos recherches concernant la mise en place des pratiques agiles.

Durant les Chapitres 5 et 6, nous avons vu que certaines pratiques n'ont pas apporté les bénéfices espérés. Par exemple, nous avons vu que la mise de l'«*équipe run*» n'a pas été bien perçue par les programmeurs et la société a été obligée de le fermer. Nous avons aussi vu que la culture des programmeurs de la société CIM et la vieille technologie PowerBuilder ont posé des problèmes pour la mise en place de système de contrôle de version et des tests unitaires. Ainsi, nous déduisons les bénéfices et les défis de l'implémentation des pratiques agiles peuvent dépendre fortement de la culture de la société et de la technologie utilisées par ce dernier. Pour cela, un futur travail serait d'étudier la mise en place des pratiques agiles dans d'autres sociétés utilisant d'autres vieilles technologies comme *Cobol* par exemple.

ANNEXE A

Mode opératoire de *PBUnit*

A.1 Généralités sur les tests Unitaires

Définition 1. ***Tester un projet*** *est un processus manuel ou automatisé qui vise à vérifier qu'un système satisfait les propriétés requises par les spécifications, ou à détecter les différences entre les résultats produits par le système et ceux attendus par les spécifications*

A.1.1 Tests

Les tests :

- préviennent des erreurs introduites par les développeurs ;
- préviennent des échecs lors d'exécutions ;
- préviennent des imperfections sur des parties du système susceptibles de causer des échecs d'exécutions ;
- représentent la **confiance** sur l'état de santé du système ;
- se construisent **progressivement** :
  - pas besoin d'écrire tous les tests d'un coup
  - à chaque nouveau **dysfonctionnement ou ticket, écrire des tests** ;
- c'est d'ailleurs meilleur de les écrire avant d'implémenter la fonctionnalité
  - agissent comme les premiers **clients** et une meilleure interface ;
- représentent une documentation active et synchrone des fonctionnalités présentes dans le système.

Dans ce guide, je vais me concentrer sur les tests unitaires.

A.1.2 Tests unitaires

A.1.2.1 Vocabulaire

Définition 2. ***Un cas de test :*** *est généralement associé à la réussite d'un scénario de cas d'utilisation. Les développeurs ont souvent des scénarios de test à l'esprit,*

mais ils les réalisent de différentes manières (1) instructions d'affichage, (2) le débogage ou les fichiers de traces, etc. Quand il est automatisé pour être répétable, il devient un cas de test unitaire. Dans la pratique, un cas de test unitaire est un ensemble d'entrées de test, de conditions d'exécution et de résultats attendus, développé pour tester un chemin d'exécution particulier. Généralement, le cas est une méthode unique ou un évènement en Powerbuilder.

Définition 3. ***Une suite de test*** *est une liste de cas de tests liés. La suite peut contenir des routines communes d'initialisation et de nettoyage spécifiques aux cas de tests qu'il contient. Généralement, la suite de test est une classe.*

A.1.2.2 Présentation de cas de test unitaire

Un cas de test répond au principe **Essaie, vérifie, si ça marche**. La Figure A.1

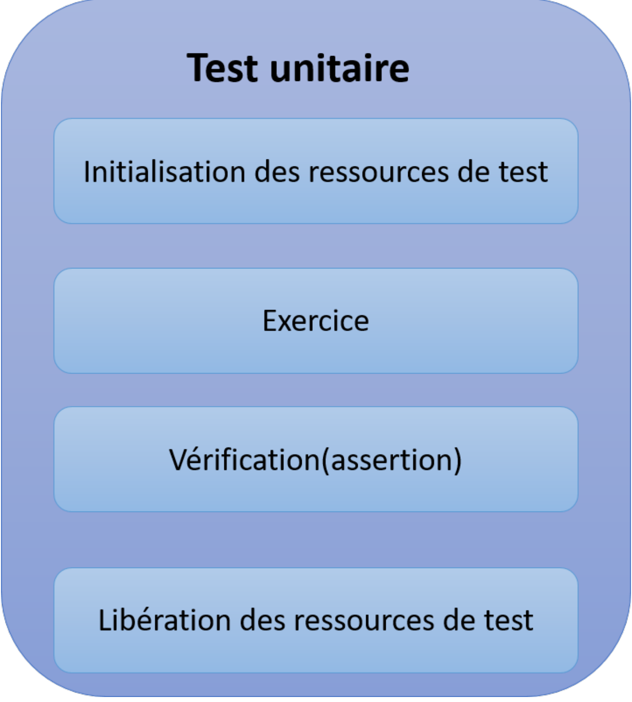


FIGURE A.1: Test case

montre les quatre étapes d'un cas de test unitaire. On distingue :

- **Set up.** consiste à préparer les ressources pour les tests ;
- **Exercise.** consiste à exercer la fonctionnalité à tester du système en tests sur les ressources précédemment préparées ;
- **Verify.** Consiste à vérifier que le résultat de l'exercice correspond bien au résultat attendu du système en test ;

— **Tear down.** Consiste à libérer les ressources utiliser durant le test

A.1.2.3 Caractéristiques d'un bon cas test unitaire

— Répétable

— Pas d'intervention humaine

— S'autodécrit

— Change moins souvent que le système

— Raconte une histoire

A.1.3 Les *smokes tests* ou tests de vérification de santé

Définition 4. *Les* smokes tests *sont des tests unitaires simples qui ont pour vocation de vérifier que le système qu'on teste fini bien et n'échoue pas à cause des erreurs systèmes. Par exemple ces tests pourrai permettre de vérifier que* Izy Protect *fini bien son exécution et ne s'arrête pas brusquement avec une* ***Erreur 6****. En revanche, on ne vérifie pas si les calculs* d'Izy Protect *sont justes ou pas.*

A.2 PBUnit

A.2.1 Présentation

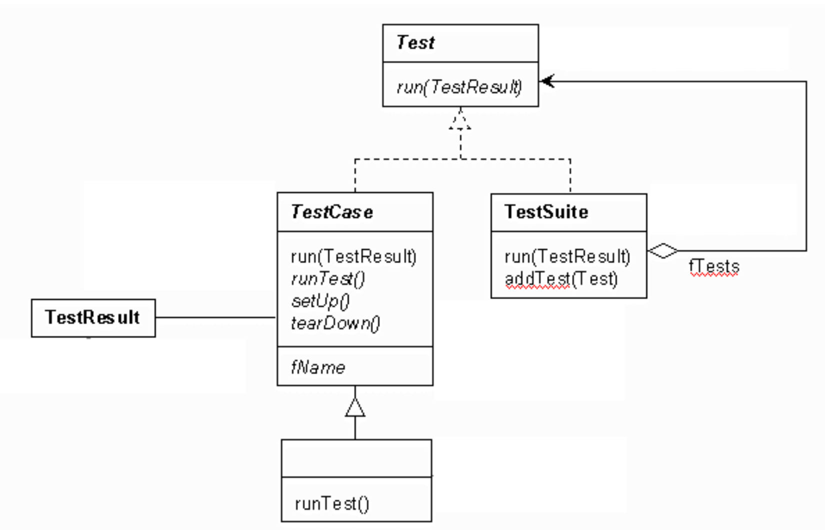


FIGURE A.2: PBUnit

PBUnit est le framework qui permet d'écrire les tests unitaires en Powerbuilder. Il est inspiré de JUnit. La Figure A.2 présente le diagramme de classe de PBUnit. La classe *TestCase* est la classe de base pour écrire les tests unitaire en Powerbuilder. Ainsi tous les tests héritent de la classe *TestCase*.

A.3 Ecriture de tests avec PBUnit

Dans le cadre de cet apprentissage, je vais prendre deux cas de figures.

1. Je vais écrire un *smoke test* simple dans le but de contrôler une **erreur 6**;
2. Je vais écrire un test unitaire normale qui test le résultat d'une exécution.

A.3.1 Préparation de l'environnement

À l'instant où j'écris ce document, PBUnit est intégré dans *Izy Protect*. Donc suivre ce mode opératoire, vous avez besoin de :

— Avoir l'IDE **Powerbuilder** installer sur votre machine

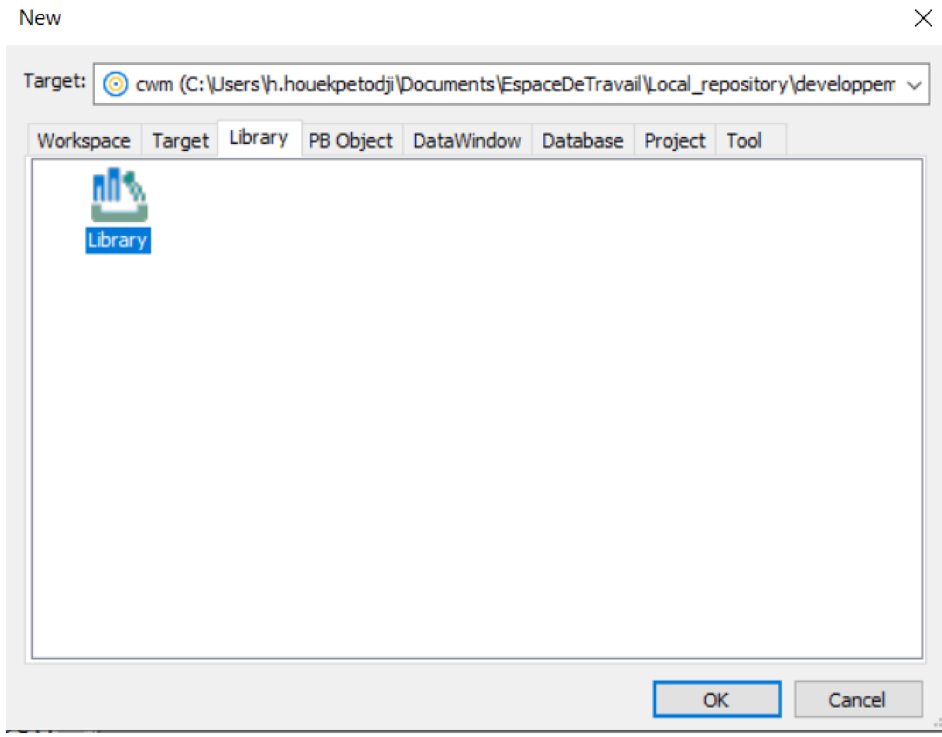
— Téléchager la dernière version de la branche de la **version évolutive** (dernière version majeur) d'*Izy Protect* (voir mode opératoire SVN pour savoir faire).

— Créer une bibliothèque suivant la convention *pbunit_{nom de la bibliothèque d'Izy Protect contenant l'objet dont on veut tester la méthode(fonction ou évènement ou sous-routine})*.

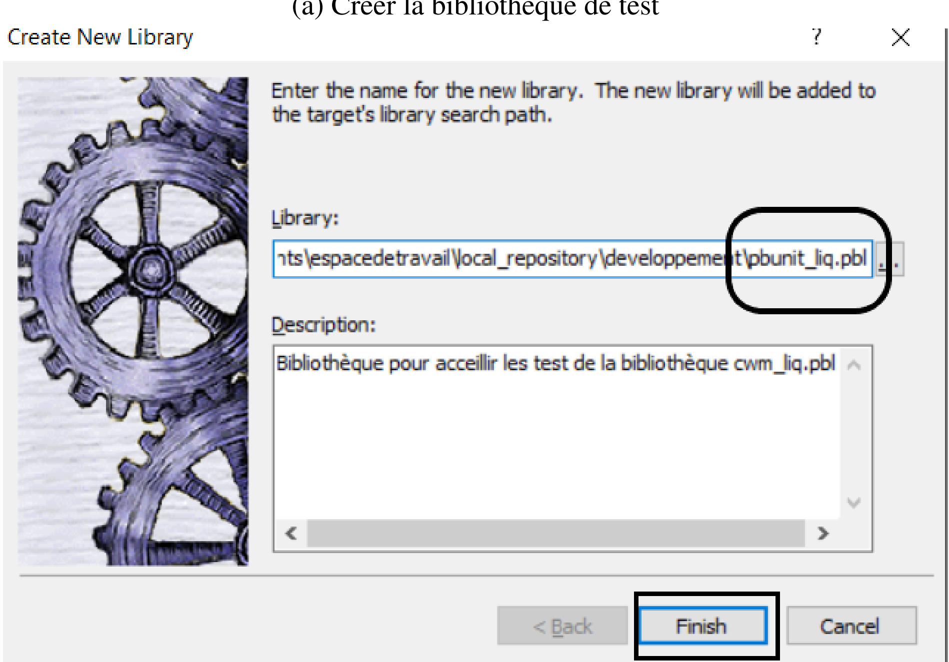
La Figure A.3 montre les différentes étapes pour créer une bibliothèque de test.

A.3.2 Création de cas de tests

Ici je vais créer trois cas de tests d'exemples. Le premier cas de test est un *smoke test*. Le deuxième cas de test sera un test simple. Le troisième cas de test sera aussi un cas de test simple, mais qui va accéder à la base de donnée **CIM999NC**. Cette base de donnée est la base de donnée dédiée aux tests unitaires. J'aborderai les informations générales plus loin. Dans le cadre des exemples, je vais considérer la fenêtre de saisie liquidation :*w_liquidation_saisie*. Donc je vais créer un héritage de l'objet utilisateur *testcase* que je vais nommer *w_liquidation_saisie_test*. Pour le *smoke test*, je vais prendre l'exemple de l'évènement *ue_init_job*. Je vais donc créer un évènement nommé *ue_init_job* dans l'objet *w_liquidation_saisie_test*. Après, je vais mettre le code source A.1.



(a) Créer la bibliothèque de test



(b) Nommer la bibliothèque de test

pbunit_trt.pbl (C:\Users\h.houekpetodji\Documents\EspaceDe
pbunit_liq.pbl (C:\Users\h.houekpetodji\Documents\EspaceDe
pbunit_test_generer.pbl (C:\Users\h.houekpetodji\Documents\

(c) Bibliothèque de test

FIGURE A.3: Création d'une bibliothèque de tests unitaire

Remarque : Il faut noter que le test du code source A.1, initialise les variables dont il a besoin pour exécuter l'évènement *ue_init_job* et se contente juste de le lancer. Il passe si l'événement s'exécute sans erreurs et échoue sinon. Ceci est un exemple.

Ceci est un exemple. On peut écrire de *smoke test* pour tous types de comportement : fonction, fonction globale, sous-routine, évènement, etc.

```
    /* EVENT ue\_init\_job */

/*Je test si ue\_init\_job se execute sans erreur*/
/* variables de tests */
gb_pbunit = true

/* declaration des objets*/
w_liquidation_saisie lw\_liquidation\_saisie

TRY
  /*initialisation des objets*/
  open(lw\_liquidation\_saisie)
  /*Exercice*/
  lw\_liquidation\_saisie.triggerevent("ue\_init\_job")
  /*verification*/
  this.assert(true)
CATCH (runtimeerror er)
    this.assert(Er.getmessage(),false)
FINALLY
/*Liberation des ressources*/
  close(lw\_liquidation\_saisie)
END TRY
gb_pbunit = false
```

Code source A.1: Test si *ue_init_job* se execute sans erreur

Pour le test simple, je vais prendre l'exemple de l'évènement *init_proprietes*. Pour ce fait, je vais créer l'évènement *test_event_int_proprietes* dans l'objet *w_liquidation_saisie_test*. Je vais y mettre le code source A.2.

Remarque : Comme on peut le voir, *test_int_proprietes*, initialise les variables dont l'évènement *int_proprietes* a besoin pour s'exécuter et puis l'exécute. Après, il vérifie que la variable d'instance *ii_dossier_type* de la fenêtre *w_liquidation_saisie* contient une valeur attendue. Il passe si ou et échoue sinon.

Une autre remarque est qu'en cas d'erreur, le test échouera aussi.

```
/*EVENT test\_int\_proprietes*/
```

```
2 /* variables de tests */
3 gb_pbunit = true
4
5 /*declaration des objets*/
6 w\_liquidation\_saisie lw\_liquidation\_saisie
7
8 /* test 1 : ouverture fenetre*/
9 TRY
10   /*initialisation des objets*/
11   open(lw_liquidation_saisie)
12   /*Exercice*/
13   lw\_liquidation\_saisie.triggerevent("init\_proprietes")
14   /*Verification ou oracle*/
15   this.assertequal(lw\_liquidation\_saisie.ii\_dossier\_type
      , DOSSIER\_CUSTOM)
16 CATCH (runtimeerror er)
17     this.assert(Er.getmessage(),false)
18 FINALLY
19   /*liberation des ressources*/
20   close(lw\_liquidation\_saisie)
21   gb\_pbunit = false
22 END TRY
```

Code source A.2: Test unitaire de *int_proprietes*

Le troisième test que je vais écrire est le test de la fonction *wf_verif_acte_cod_sui* (code source A.3). La particularité de ce test est que j'insère des données dans la base de donnée dont la fonction a besoin via la datawindow *dw_dev*. Après je teste la fonction. Mais je n'oublie pas de vider les données pour que ces dernières n'influence pas le résultat d'autres tests. Par contre, on peut pré-initialiser les données pour un ensemble de tests.

```
1 /*Event test wf\_verif\_acte\_cod\_sui */
2 /* declaration des variables*/
3 w\_liquidation\_saisie lw\_liquidation\_saisie
4 string ls\_prs\_coc
5 long ll_lig
6
7
8 TRY
9 /* initialisation des variables*/
10   open(lw\_liquidation\_saisie)
11   ls\_prs\_coc =  "TEST"
12   ll_lig = 2
13
```

```
14
15   /* dw_dev , directement dans la datawindow*/
16   lw\_liquidation\_saisie.dw\_dev.insertrow(0)
17   lw\_liquidation\_saisie.dw\_dev.insertrow(1)
18
19   lw\_liquidation\_saisie.dw\_dev.setItem(1 , "prs_no" , 2 )
20   lw\_liquidation\_saisie.dw\_dev.setItem(2 , "prs_no" , 1 )
21
22
23   /* dw_sui , dans la base avec un retrieve()*/
24   insert into prs\_prm ( prs\_no , prs\_sui )
25     values ( 1 , :ls\_prs\_coc )
26     using sqlca;
27   if sqlca.sqlcode <> 0 then
28     this.assert( "erreur insertion de data dans prs\_prm" ,
      false)
29     rollback using sqlca;
30   else
31     commit using sqlca;
32   end if
33
34   insert into prs (prs\_no , prs\_coc , prs\_pel )
35     values ( 1 , '' , 1 )
36     using sqlca;
37   if sqlca.sqlcode <> 0 then
38     this.assert( "erreur insertion de data dans prs" , false
      )
39     rollback using sqlca;
40   else
41     commit using sqlca;
42   end if
43
44
45   /* test 1 : ouverture fenetre*/
46
47
48   this.assertequal ( true , lw\_liquidation\_saisie.wf\
      _verif\_acte\_cod\_sui( ls_prs_coc , ll_lig ) )
49
50 CATCH (runtimeerror er)
51     this.assert(Er.getmessage(),false)
52 FINALLY
53 /*Liberation des ressources*/
```

```
54   close(lw\_liquidation\_saisie)
55   delete from prs\_prm
56   where prs\_no in ( 1 , 2 )
57 using sqlca;
58
59 delete from prs
60   where prs\_no in ( 1 , 2 )
61 using sqlca;
62
63 commit using sqlca;
64 END TRY
```

Code source A.3: Test unitaire de *wf_verif_acte_cod_sui*

Un autre exemple de test est celui de la fonction *f_decimal*.

```
1     /*  EVENT test\_integerString\_to\_decimal () */
2     this.assertEqual(f\_decimal('12'),12.00)
```

Code source A.4: f_decimal test 1

```
1     /*  EVENT test\_decimalString\_to\_decimal () */
2     this.assertEqual(f\_decimal('0.45127'),0.45127)
```

Code source A.5: f_decimal test 2

A.3.3 Explication de l'interface graphique de PBUnit

Au moment où j'écris ce mode opératoire, *PBUnit* est intégré dans Izy Protect. Pour y accéder, il faut lancer Izy Protect et cliquer sur le bouton de *PBUnit* dans le menu principal. Comme l'indique la Figure A.4 La fenêtre de la Figure A.5 s'ouvre

L'interface de *PBUnit* est composée de trois parties principales. La partie (1) présente la liste des tests groupés par les objets de test auxquels ils appartiennent selon la *target* sélectionnée. Parlant de *target*, sélectionnons notre target *cwm.pbt*. La sélection de *target* est une étape très importante dans l'exécution des tests de *PBUnit*. Pour choisir une *target*, dans le menu de *PBUnit*, je fais : File > Open Target ... comme indiquer sur la Figure A.6 ou bien faite Ctrl + O.

Ensuite, je navigue vers l'emplacement de la *target* qui m'intéresse dans le système fichier. Dans le cadre de ce mode opératoire la *target* sera *cwm.pbt* comme le montre la Figure A.7. La Figure A.8 montre la partie (1) de PBUnit après chargement des tests.

Pour exécuter un test, je sélectionne un test et cliquer sur ***run*** ou bien cliquer sur ***runAll***. La Figure A.9 présente *PBUnit* après exécution des tests.

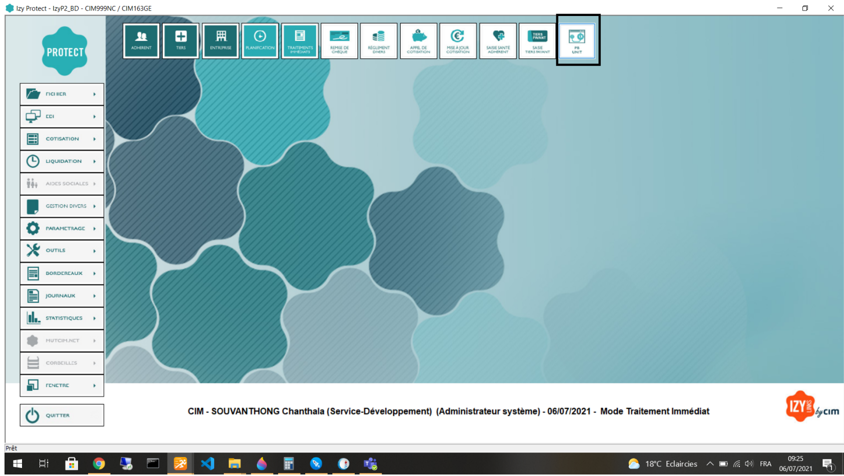


FIGURE A.4: Lancer *PBUnit*

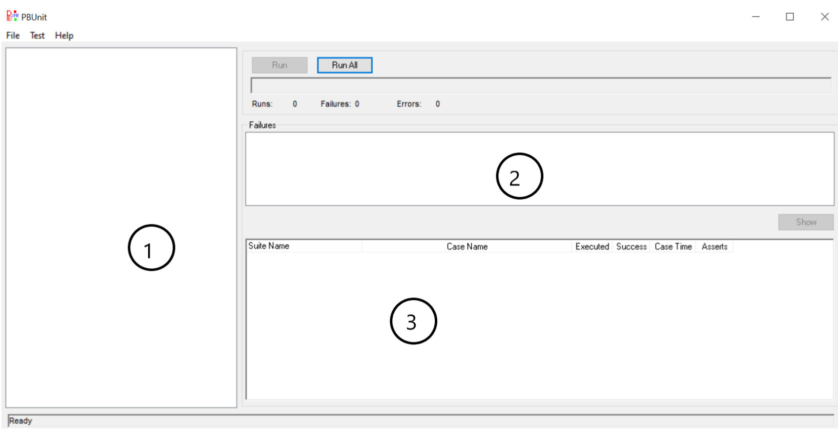


FIGURE A.5: *PBUnit*

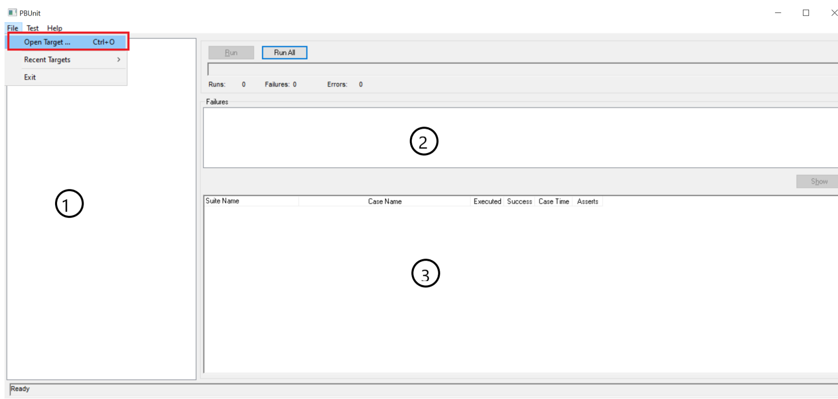


FIGURE A.6: Choisir une target

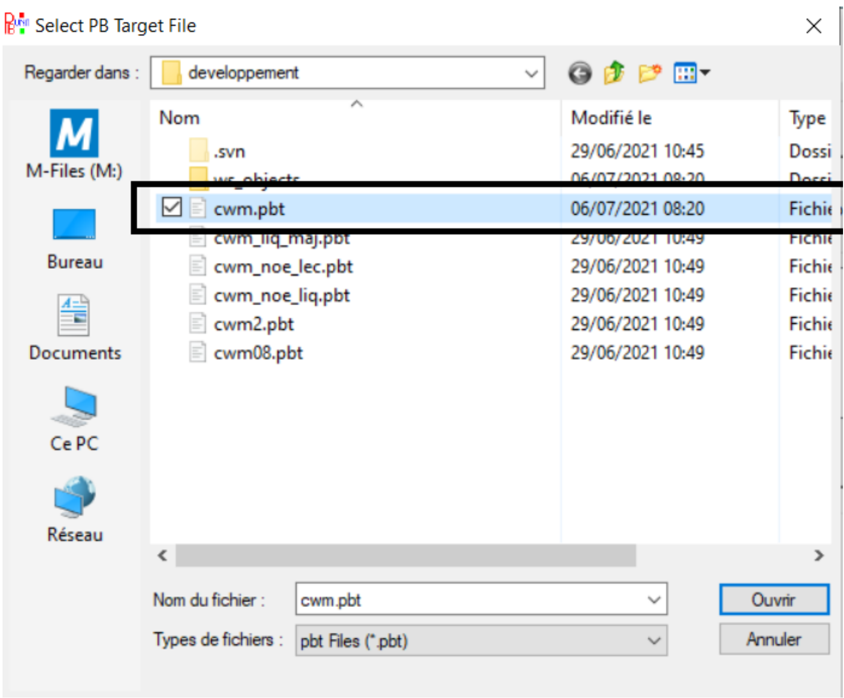


FIGURE A.7: Selectionner la *target* à son emplacement

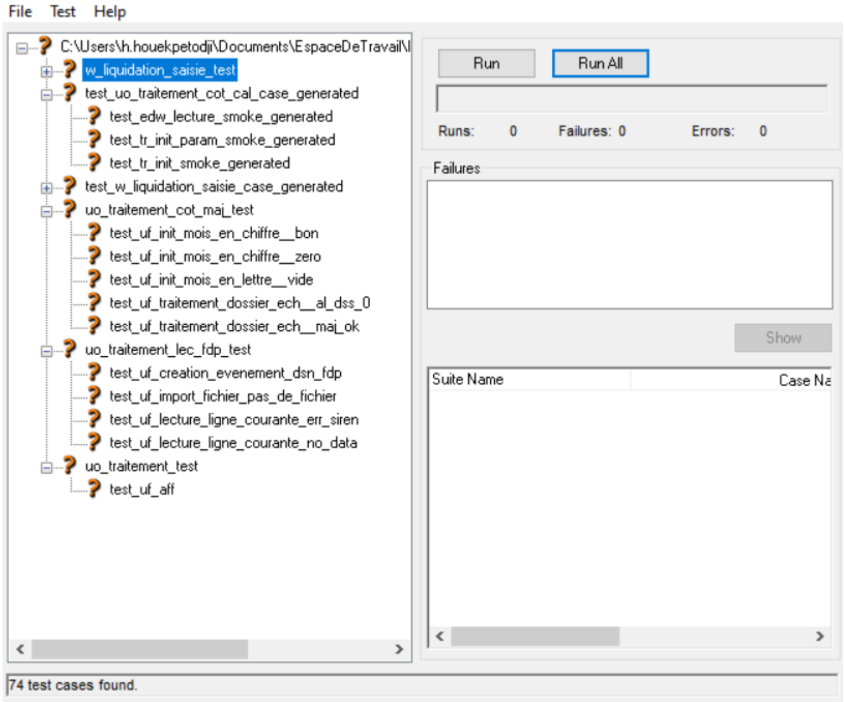


FIGURE A.8: PBUnit après chargement des tests

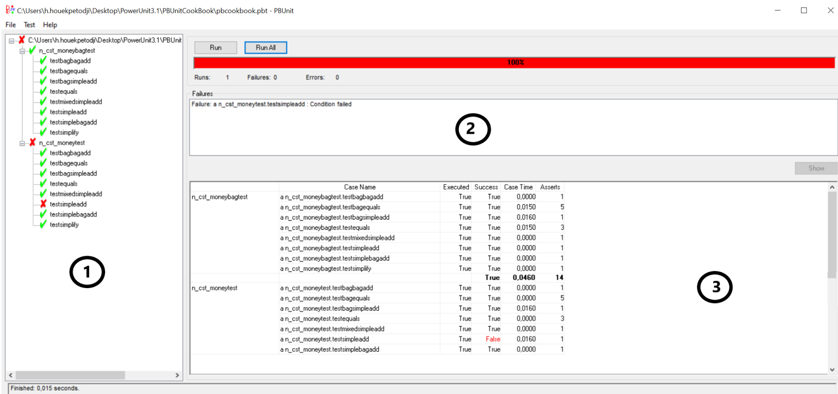


FIGURE A.9: PBUnit après exécution des tests

La partie (2) présente l'ensemble des tests qui ont échoué et la raison de l'échec. La partie (3) présente un rapport des exécutions, l'état, le temps d'exécution, etc. Dans le cas où tous les tests passent, *PBUnit* ressemble à la Figure A.10

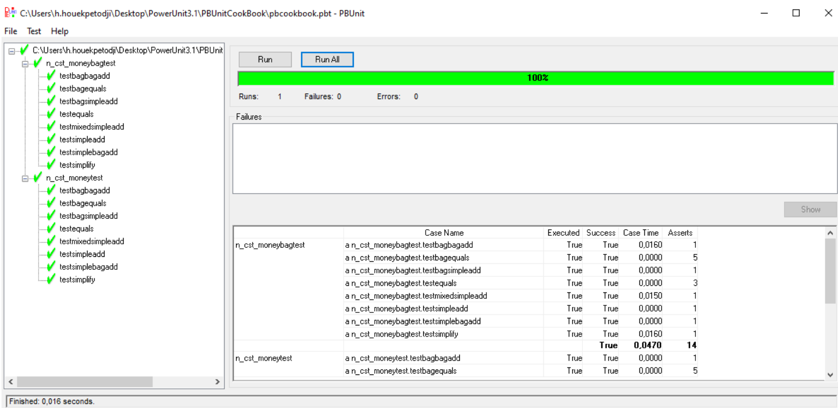


FIGURE A.10: *PBUnit* après exécution des tests

Bibliographie

[Adolph 2011] Steve Adolph, Wendy Hall and Philippe Kruchten. *Using grounded theory to study the experience of software development*. Empirical Software Engineering, vol. 16, no. 4, pages 487–513, 2011. 20

[Aitken 2013] Ashley Aitken and Vishnu Ilango. *A Comparative Analysis of Traditional Software Engineering and Agile Software Development*. In 2013 46th Hawaii International Conference on System Sciences, pages 4751–4760, 2013. 6

[Akinsola 2020] Jide ET Akinsola, Afolakemi S Ogunbanwo, Olatunji J Okesola, Isaac J Odun-Ayo, Florence D Ayegbusi and Ayodele A Adebiyi. *Comparative analysis of software development life cycle models (SDLC)*. In Computer Science On-line Conference, pages 310–322. Springer, 2020. 2

[Alex 2021] Alex. *sashazjukov/PBSCAnalyzer*, September 2021. original-date : 2016-08-30T17 :43 :05Z. 11

[Almeida 2017] Fernando Almeida. *Challenges in migration from waterfall to agile environments*. World Journal of Computer Application and Technology, vol. 5, no. 3, pages 39–49, 2017. 11

[Alshamrani 2015] Adel Alshamrani and Abdullah Bahattab. *A comparison between three SDLC models waterfall model, spiral model, and Incremental/Iterative model*. International Journal of Computer Science Issues (IJCSI), vol. 12, no. 1, page 106, 2015. 7, 8

[Anquetil 2020] Nicolas Anquetil, Anne Etien, Mahugnon H Houekpetodji, Benoit Verhaeghe, Stéphane Ducasse, Clotilde Toullec, Fatiha Djareddir, Jerôme Sudich and Moustapha Derras. *Modular Moose : A New Generation of Software Reverse Engineering Platform*. In International Conference on Software and Software Reuse, pages 119–134. Springer, 2020. 4

[Bassil 2001] Sarita Bassil and Rudolf K. Keller. *Software Visualization Tools : Survey and Analysis*. In Proceedings IWPC 2001, pages 7–17, 2001. 56

[Bhuvaneswari 2013] T Bhuvaneswari and S Prabaharan. *A survey on software development life cycle models*. International Journal of Computer Science and Mobile Computing, vol. 2, no. 5, pages 262–267, 2013. 7

[Bragagnolo 2021] Santiago Bragagnolo, Nicolas Anquetil, Stéphane Ducasse, Abderrahmane Seriai and Mustapha Derras. *Software Migration : A Theoretical Framework (A Grounded Theory approach on Systematic Literature Review)*. Empirical Software Engineering, 2021. 20, 21

[Charmaz 2014] Kathy Charmaz. Constructing grounded theory. sage, 2014. 47

[Charters 2003] Elizabeth Charters. *The use of think-aloud methods in qualitative research an introduction to think-aloud methods*. Brock Education Journal, vol. 12, no. 2, 2003. 18

[Chen 2015] Roger Chen, Ramya Ravichandar and Donald Proctor. *Managing the Transition to Agile Product Development —-Lessons from Cisco Systems*. Academy of Management Proceedings, vol. 2015, no. 1, page 11327, 2015. 2, 59

[Deshpande 2016] Advait Deshpande, Helen Sharp, Leonor Barroca and Peggy Gregory. *Remote Working and Collaboration in Agile Teams*. In International Conference on Information Systems, 2016. 13

[Despa 2014] Mihai Liviu Despa. *Comparative study on software development methodologies*. Database Systems Journal, vol. 5, no. 3, pages 37–56, 2014. 6, 8

[Dingsøyr 2013] Torgeir Dingsøyr and Yngve Lindsjørn. *Team Performance in Agile Development Teams : Findings from 18 Focus Groups*. In Hubert Baumeister and Barbara Weber, editeurs, Agile Processes in Software Engineering and Extreme Programming, pages 46–60, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. 17

[Dorairaj 2012] Siva Dorairaj, James Noble and Petra Malik. *Knowledge Management in Distributed Agile Software Development*. In 2012 Agile Conference, pages 64–73, 2012. 13

[Dörnenburg 2018] E. Dörnenburg. *The Path to DevOps*. IEEE Software, vol. 35, no. 5, pages 71–75, 2018. 56, 57

[Fowler 2018] Martin Fowler. Refactoring : Improving the Design of Existing Code. Addison-Wesley Professional, Boston, 2 edition édition, November 2018. 76

[Gandomani 2013] Taghi Javdani Gandomani, Hazura Zulzalil, Abdul Azim Abdul Ghani, Abu Bakar Md Sultan and Mina Ziaei Nafchi. *Obstacles in moving to agile software development methods ; at a glance*. Journal of Computer Science, vol. 9, no. 5, page 620, 2013. 12

[Groeneveld 2021] Wouter Groeneveld, Laurens Luyten, Joost Vennekens and Kris Aerts. *Exploring the Role of Creativity in Software Engineering*. In 2021 IEEE/ACM 43rd International Conference on Software Engineering : Software Engineering in Society (ICSE-SEIS), pages 1–9, 2021. 17

[Hamill 2004] Paul Hamill. Unit test frameworks : tools for high-quality software development. " O'Reilly Media, Inc.", 2004. 73

[Hanslo 2018] R. Hanslo and E. Mnkandla. *Scrum Adoption Challenges Detection Model : SACDM*. In 2018 Federated Conference on Computer Science and Information Systems (FedCSIS), pages 949–957, 2018. 9

[Honore 2020] HOUEKPETODJI Honore, Fatiha Djareddir, Jérôme Sudich and Nicolas Anquetil. *Improving practices in a medium french company : First step*. In RIMEL : Journée de travail Rimel / Lignes de produit / Sécurité, Paris, France, 2020. 4

[Hora 2012] Andre Hora, Nicolas Anquetil, Stéphane Ducasse and Simon Allier. *Domain Specific Warnings : Are They Any Better ?* In Proceedings of the 28th IEEE International Conference on Software Maintenance (ICSM'12), 2012. 68, 93

[Houekpetodji 2021] Mahugnon Honoré Houekpetodji, Nicolas Anquetil, Stéphane Ducasse, Fatiha Djareddir and Jérôme Sudich. *Report From The Trenches A Case Study In Modernizing Software Development Practices*. In 2021 IEEE International Conference on Software Maintenance and Evolution (ICSME), pages 515–524, 2021. 4

[Johanssen 2018] Jan Ole Johanssen, Anja Kleebaum, Barbara Paech and Bernd Bruegge. *Practitioners' eye on continuous software engineering : An interview study*. In Proceedings of the 2018 International Conference on Software and System Process, pages 41–50, 2018. 17

[Johnson 2013] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill and Robert Bowdidge. *Why don't software developers use static analysis tools to find bugs ?* In 2013 35th International Conference on Software Engineering (ICSE), pages 672–681. IEEE, 2013. 11

[Khadka 2014] Ravi Khadka, Belfrit V Batlajery, Amir M Saeidi, Slinger Jansen and Jurriaan Hage. *How do professionals perceive legacy systems and software modernization ?* In Proceedings of the 36th International Conference on Software Engineering, pages 36–47, 2014. 3, 17, 18, 20, 47

[Kitchenham 1999] Barbara A. Kitchenham, Guilherme H. Travassos, Anneliese von Mayrhauser, Frank Niessink, Norman F. Schneidewind, Janice Singer, Shingo Takada, Risto Vehvilainen and Hongji Yang. *Towards an ontology of software maintenance*. Journal of Software Maintenance : Research and Practice, vol. 11, no. 6, pages 365–389, 1999. 3, 13, 15, 92

[Kuckartz 2019] Udo Kuckartz and Stefan Rädiker. Analyzing qualitative data with maxqda. Springer, 2019. 20, 47

[Lanza 2006] Michele Lanza and Radu Marinescu. Object-oriented metrics in practice. Springer-Verlag, 2006. 76

[Larman 2004] Craig Larman. Agile and iterative development : a manager's guide. Addison-Wesley Professional, 2004. 8, 9

[Leau 2012] Yu Beng Leau, Wooi Khong Loo, Wai Yip Tham and Soo Fun Tan. *Software development life cycle AGILE vs traditional approaches*. In Inter-

national Conference on Information and Network Technology, volume 37, pages 162–167, 2012. 6

[Lenarduzzi 2017] Valentina Lenarduzzi, Alexandru Cristian Stan, Davide Taibi, Davide Tosi and Gustavs Venters. *A dynamical quality model to continuously monitor software maintenance*. In The European Conference on Information Systems Management, pages 168–178. Academic Conferences International Limited, 2017. 19, 36

[Mahalakshmi 2013] M Mahalakshmi and Mukund Sundararajan. *Traditional SDLC vs scrum methodology–a comparative study*. International Journal of Emerging Technology and Advanced Engineering, vol. 3, no. 6, pages 192–196, 2013. 9

[Mancl 2020] Dennis Mancl and Steven D Fraser. *COVID-19's influence on the future of agile*. In International Conference on Agile Software Development, pages 309–316. Springer, 2020. 13

[Marcilio 2019] Diego Marcilio, Rodrigo Bonifácio, Eduardo Monteiro, Edna Canedo, Welder Luz and Gustavo Pinto. *Are Static Analysis Violations Really Fixed ? A Closer Look at Realistic Usage of SonarQube*. In 2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC), pages 209–219, 2019. 10, 11

[McCabe 1976] Thomas J. McCabe. *A complexity measure*. In ICSE'76 : Proceedings of the 2nd International Conference on Software engineering, page 407, Los Alamitos, CA, USA, 1976. IEEE Computer Society. 77

[Mikalsen 2021a] Marius Mikalsen, Nils Brede Moe, Sut I Wong and Viktoria Stray. *Agile Information System Development Organizations Transforming to Large-Scale Collaboration*, 2021. 11

[Mikalsen 2021b] Marius Mikalsen, Nils Brede Moe, Sut I Wong and Viktoria Stray. *Agile Information System Development Organizations Transforming to Large-Scale Collaboration*. arXiv preprint arXiv :2111.06193, 2021. 55

[MOB 2018] *ROI of PowerBuilder Migrations*. https://cdn2.hubspot.net/hubfs/216184/ROI%20of%20PowerBuilder%20Modernizations.pdf, 2018. 24

[Mohammad 2017] Sikender Mohsienuddin Mohammad. *DevOps automation and Agile methodology*. International Journal of Creative Research Thoughts (IJCRT), ISSN, pages 2320–2882, 2017. 9

[Moo 2021] *PowerBuilderParser*, December 2021. original-date : 2018-12-07T14 :07 :40Z. 11

[Munassar 2010] Nabil Mohammed Ali Munassar and A Govardhan. *A comparison between five models of software engineering*. International Journal of Computer Science Issues (IJCSI), vol. 7, no. 5, page 94, 2010. 7, 8

[Nerur 2005] Sridhar Nerur, RadhaKanta Mahapatra and George Mangalaraj. *Challenges of Migrating to Agile Methodologies*. Commun. ACM, vol. 48, no. 5, page 72–78, May 2005. 5, 11, 12

[Nielson 1999] Flemming Nielson, Hanne Riis Nielson and Chris Hankin. *Type and effect systems*. In Principles of Program Analysis, pages 283–363. Springer, 1999. 10

[Novak 2010] Jernej Novak, Andrej Krajnc and Rok Žontar. *Taxonomy of static code analysis tools*. In The 33rd International Convention MIPRO, pages 418–422, 2010. 11

[Otte 2009] Stefan Otte. *Version control systems*. Computer Systems and Telematics, pages 11–13, 2009. 58

[pbd 2011] *Editing scripts - - Users Guide*, 2011. 26

[Picek 2009] Ruben Picek. *Suitability of modern software development methodologies for model driven development*. Journal of Information and Organizational Sciences, vol. 33, no. 2, pages 285–295, 2009. 9

[Pigoski 1996] Thomas M. Pigoski. Practical software maintenance : Best practices for managing your software investment. Wiley Publishing, 1st édition, 1996. 38

[Port 2017] Dan Port and Bill Taber. *Actionable analytics for strategic maintenance of critical software : an industry experience report*. IEEE Software, vol. 35, no. 1, pages 58–63, 2017. 36

[Port 2018] Dan Port and Bill Taber. *An empirical study of process policies and metrics to manage productivity and quality for maintenance of critical software systems at the jet propulsion laboratory*. In Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, page 37. ACM, 2018. 38

[Rafi 2012] Dudekula Mohammad Rafi, Katam Reddy Kiran Moses, Kai Petersen and Mika V. Mäntylä. *Benefits and limitations of automated software testing : Systematic literature review and practitioner survey*. In 2012 7th International Workshop on Automation of Software Test (AST), pages 36–42, 2012. 10, 71, 75, 85

[Rizvi 2015] Buturab Rizvi, Ebrahim Bagheri and Dragan Gasevic. *A systematic review of distributed Agile software engineering*. Journal of Software : Evolution and Process, vol. 27, no. 10, pages 723–762, 2015. 13

[Royce 1987] W. W. Royce. *Managing the Development of Large Software Systems*. In R.H. Thayer, editeur, Tutorial : Software Engineering Project Management, pages 118–127. IEEE Computer Society, Washington, 1987. fca. 7

[Runeson 2009] Per Runeson and Martin Höst. *Guidelines for conducting and reporting case study research in software engineering*. Empirical software engineering, vol. 14, no. 2, pages 131–164, 2009. 16

[Schwaber 1997] Ken Schwaber. *SCRUM Development Process*. pages 117–134, London, 1997. Springer London. 9

[Schwaber 2004] Ken Schwaber. Agile project management with scrum. Microsoft Press, USA, 2004. 8

[Seaman 1999] C.B. Seaman. *Qualitative methods in empirical studies of software engineering*. IEEE Transactions on Software Engineering, vol. 25, no. 4, pages 557–572, 1999. 17

[Singer 2008] Janice Singer, Susan E Sim and Timothy C Lethbridge. *Software engineering data collection for field studies*. In Guide to Advanced Empirical Software Engineering, pages 9–34. Springer, 2008. 2, 16, 17, 18

[Sjoberg 2007] Dag I. K. Sjoberg, Tore Dyba and Magne Jorgensen. *The Future of Empirical Methods in Software Engineering Research*. In Future of Software Engineering (FOSE '07), pages 358–378, 2007. 16

[Smite 2021] Darja Smite, Marius Mikalsen, Nils B. Moe, Viktoria Stray and Eriks Klotins. *From Collaboration to Solitude and Back : Remote Pair Programming during COVID-19*, 2021. 2, 55

[Stoica 2013] Marian Stoica, Marinela Mircea and Bogdan Ghilic-Micu. *Software development : Agile vs. traditional*. Informatica Economica, vol. 17, no. 4, 2013. 2, 6

[Taipale 2011] Ossi Taipale, Jussi Kasurinen, Katja Karhu and Kari Smolander. *Trade-off between automated and manual software testing*. International Journal of System Assurance Engineering and Management, vol. 2, no. 2, pages 114–125, 2011. 10, 75

[Upadhyay 2021] Neha Upadhyay, Manmohan Singh, Manish Shrivastava and Aishwarya Mishra. *Analysis And Comparison With Modern Software Development Approaches*. NVEO-NATURAL VOLATILES & ESSENTIAL OILS Journall NVEO, pages 7553–7559, 2021. 8

[Vohra 2013] Pankaj Vohra and Ashima Singh. *A contrast and comparison of modern software process models*. In International Conference on Advances in Management and Technology (iCAMT-2013), pages 23–27. Citeseer, 2013. 2, 8, 9

[Wiklund 2017] Kristian Wiklund, Sigrid Eldh, Daniel Sundmark and Kristina Lundqvist. *Impediments for software test automation : A systematic literature review*. Software Testing, Verification and Reliability, vol. 27, no. 8, page e1639, 2017. 77

[Yang 2020] Yanming Yang, Xin Xia, David Lo and John Grundy. *A survey on deep learning for software engineering*. ACM Computing Surveys (CSUR), 2020. 18

[Zafar 2018] Iqra Zafar, Asma Shaheen, Aiman Khan Nazir, Bilal Maqbool, Wasi Haider Butt and Jahan Zeb. *Why Pakistani Software Companies don't use Best Practices for Requirement Engineering Processes*. In 2018 IEEE 9th Annual Information Technology, Electronics and Mobile Communication Conference (IEMCON), pages 996–999. IEEE, 2018. 16

[Zhang 2010] H. Zhang and S. Kim. *Monitoring Software Quality Evolution for Defects*. IEEE Software, vol. 27, no. 4, pages 58–64, July 2010. 19

[Zhang 2013] He Zhang and Muhammad Ali Babar. *Systematic reviews in software engineering : An empirical investigation*. Information and software technology, vol. 55, no. 7, pages 1341–1354, 2013. 18

[Zhang 2018] Li Zhang, Jia-Hao Tian, Jing Jiang, Yi-Jun Liu, Meng-Yuan Pu and Tao Yue. *Empirical research in software engineering—a literature survey*. Journal of Computer Science and Technology, vol. 33, no. 5, pages 876–899, 2018. 16