Université de Namur
Faculté d'informatique
Année académique 2013–2014

Détection de fichiers pertinents : Expérimentation sur l'analyse de traces

Thomas Drioul
Pierre-Antoine Rappe

Maîtres de stage : Yann-Gaël Guéhéneuc

Promoteur : _____ (Signature pour approbation du dépôt - REE art. 40)
Naji Habra

Mémoire présenté en vue de l'obtention du grade de
Master en Sciences Informatiques.

Abstract

Résumé Depuis de nombreuses années, des recherches ont été effectuées pour mieux appréhender les concepts de maintenance logicielle et de compréhension logicielle. Ce travail s'inscrit dans cette ligne ayant pour finalité, la réduction du temps de maintenance. Cette réduction doit s'effectuer sur le temps de recherche car des études ont prouvé que le temps de recherche est bien plus important que le temps de modification lors d'une maintenance. Pour essayer de réduire ce temps, une expérimentation a été conduite pour déceler les fichiers pertinents. Une distinction a été faite entre les fichiers hautement et significativement pertinents. Cette expérimentation a donné des traces vidéos de résolutions de différentes tâches qui ont été analysées. Des critères de pertinence ont pu être en extraits : le temps total passé dans un fichier, le temps total de recherche passé dans un fichier et le nombre de revisites d'un fichier.

Mots clés Maintenance logicielle, compréhension logicielle, significativement pertinent et hautement pertinent.

Abstract For many years, researchs have been conducted to understand in a better way the concepts of software maintenance and software understanding. This work is axed on these fields in which a reduction of the maintenance time is the ultimate goal. Moreover, the reduction must be made on the search time because a statement is that the search time is more important than the edit time for a task. In order to reduce this time, an expriment has been conducted to find the pertinent files. A distinction has been made between a significantly and a highly relevant file. This experiment gave from video traces of resolved maintenance tasks. Some criteria have been extracted of these traces. These criteria are the total time spent in a file, the total searching time spent in a file and the revisit number of a file.

Keywords Software maintenance, software comprehension, significantly relevant and highly relevant.

Remerciements

Ce mémoire est le résultat de notre stage de 4 mois au sein du laboratoire Ptidej de l'Ecole Polytechnique de Montréal, Canada. Ce laboratoire a pour but de développer des théories, méthodes et outils pour évaluer et augmenter la qualité de programmes orientés objets. Ce stage fut supervisé par Yann-Gaël Guéhéneuc, Foutse Khomh et Zéphyrin Soh. Nous tenons particulièrement à les remercier pour nous avoir permis d'effectuer ce stage. Nous tenons aussi à les remercier pour la découverte qu'ils nous ont offerte du monde scientifique. Nous remercions aussi l'ensemble des chercheurs de l'équipe du laboratoire Ptidej et du laboratoire SoccerLab de l'école polytechnique de Montréal pour leurs conseils et leur sympathie lors de notre stage.

Nous tenions aussi à remercier notre promoteur de notre mémoire, Naji Habra, pour le suivi et les conseils donnés lors de notre stage et pour l'aide apportée lors de la rédaction de ce mémoire.

Notre mémoire se basant sur une expérimentation, nous remercions l'ensemble des sujets y ayant participé. Sans eux, nous ne pourrions avoir écrit ce travail.

Finalement, nous remercions nos parents, familles et amis de nous avoir permis d'accomplir ce stage à l'étranger mais aussi pour leur soutien lors de nos études et de la rédaction de ce mémoire.

Table des matières

1 Introduction 1
- 1.1 Mise en contexte du travail . . . 1
  - 1.1.1 Maintenance logicielle . . . 1
  - 1.1.2 Compréhension du logiciel . . . 2
- 1.2 But du mémoire . . . 3
- 1.3 Plan du mémoire . . . 4

I Etat de l'art 5

2 Compréhension du logiciel 7
- 2.1 Méthodes d'analyse statique . . . 7
  - 2.1.1 Analyse conceptuelle et logique . . . 8
  - 2.1.2 Analyse du code source . . . 10
  - 2.1.3 Analyses des données externes non-structurées . . . 11
- 2.2 Les méthodes d'analyse dynamique . . . 13

3 Maintenance logicielle 15
- 3.1 Vision d'une maintenance . . . 15
- 3.2 But d'une maintenance . . . 16
- 3.3 Types de maintenance . . . 17
- 3.4 Étapes d'une maintenance . . . 19
  - 3.4.1 Le processus selon l'IEEE . . . 20
  - 3.4.2 Le processus selon Aggarwal et Singh . . . 21
- 3.5 Coûts d'une maintenance . . . 22
- 3.6 Modèles de maintenance . . . 24
  - 3.6.1 Modèle quick-fix . . . 24
  - 3.6.2 Modèle d'amélioration itérative . . . 24
  - 3.6.3 Modèle orienté réutilisation . . . 26
  - 3.6.4 Modèle de Boehm . . . 26
  - 3.6.5 Modèle de Taute . . . 28
  - 3.6.6 Modèle d'Osborne . . . 28
  - 3.6.7 Modèle du cycle de vie conscient de la maintenance . 30

II Expérimentation 33

4 Aperçu du processus expérimental 35
4.1 Variables, traitements, objets et sujets 35
4.2 Processus expérimental 37

5 Définition de l'expérimentation 39
5.1 Phase de définition d'une expérimentation : rappels théoriques 39
5.2 Définition concrète de notre expérimentation 41
5.2.1 Idée intuitive de l'expérimentation 41
5.2.2 Définition formelle de l'expérimentation 42

6 Planification de l'expérimentation 43
6.1 Phase de la planification d'une expérimentation : rappels théoriques 43
6.1.1 Sélection du contexte 44
6.1.2 Formulation d'hypothèses 45
6.1.3 Sélection des variables 46
6.1.4 Sélection des sujets 46
6.1.5 Design d'une expérimentation 48
6.1.6 Instrumentation d'une expérimentation 49
6.2 Planification concrète de notre expérimentation 50
6.2.1 Sélection du contexte 50
6.2.2 Formulation d'hypothèses 52
6.2.3 Sélection des variables 52
6.2.4 Sélection des sujets 52
6.2.5 Design de notre expérimentation 53
6.2.6 Instrumentation de notre expérimentation 54

7 Opération de l'expérimentation 55
7.1 Phase d'opération : rappels théoriques 56
7.1.1 Étape de préparation 56
7.1.2 Étape d'exécution 57
7.1.3 Étape de validation des données 58
7.2 Opération concrète de notre expérimentation 58
7.2.1 Étape de préparation 58
7.2.2 Étape d'exécution 59
7.2.3 Étape de validation des données 64

8 Analyses et interprétations 65
8.1 Validité de l'expérimentation : rappels théoriques 65
8.2 Confirmation de l'importance du temps de recherche 68
8.3 Établissement des fichiers de références 70
8.3.1 Fichiers significativement pertinents 70

8.3.2 Fichiers hautement pertinents 70
8.4 Définition des critères de pertinence d'un fichier 70
8.4.1 Le temps comme critère de pertinence 71
8.4.2 Le nombre de revisites comme critère de pertinence . 79
8.4.3 Discussion sur la différentiation des fichiers hautement et significativement pertinents 82
8.5 Menaces sur les résultats de l'expérimentation 82
8.5.1 Le choix des sujets 82
8.5.2 L'expérience en programmation des sujets 83
8.5.3 Pratique des sujets avec l'environnement expérimental 83
8.5.4 Le choix des tâches à effectuer 83
8.5.5 La motivation du sujet lors de l'expérimentation . . . 84
8.5.6 L'interprétation des données 84

9 Conclusion 85
9.1 Résumé et points importants du mémoire 85
9.2 Travaux futurs 87
9.2.1 Point de vue expérimentation 87
9.2.2 Point de vue logiciel et outil 88
9.2.3 Réutilisation de l'expérimentation 88

Bibliographie 89

Annexes 93
A Formulaires 93
A.1 Formulaire projet Eclipse 93
A.2 Formulaire projet Java simple 96
B Résumé des projets 98
B.1 ECF 98
B.2 PDE 103
B.3 jEdit 106
B.4 JHotDraw 109
C Présentation des tâches 112
C.1 ECF 112
C.2 PDE 120
C.3 jEdit 128
C.4 JHotDraw 132
D Checklist 136
E Procédure de l'expérimentation 138
F Configuration d'Eclipse 140
F.1 Eclipse 3.5.2 140
F.2 Eclipse 4.3 141
G Données sur les sujets 141

Table des figures

1.1 Itération d'une maintenance 4

2.1 Acteur en UML 8
2.2 Cas d'utilisation en UML 9

3.1 Pourcentage de répartition des types des tâches de maintenance pour Klint 18
3.2 Pourcentage de répartition des types des tâches de maintenance pour Arrgarwal et Singh 19
3.3 Processus de maintenance IEEE 20
3.4 Processus de maintenance logicielle 22
3.5 Modèle de maintenance Quick-fix 25
3.6 Modèle de maintenance d'amélioration itérative 25
3.7 Modèle de maintenance orienté réutilisation 26
3.8 Modèle de maintenance de Boehm 27
3.9 Modèle de maintenance de Taute 28
3.10 Modèle de maintenance d'Osborne 29
3.11 Modèle du cycle de vie conscient de la maintenance 30
3.12 Effort nécessaire lors des différentes étapes du système 31

4.1 Illustration des variables dépendantes et indépendantes . . . 36
4.2 Illustration d'une expérimentation 36
4.3 Processus d'une expérimentation 38

5.1 Phases de la définition d'une expérimentation 40
5.2 Caractérisation du contexte 41

6.1 Planification d'une expérimentation 44

7.1 Phase opérationnelle 55

8.1 Principes d'une expérimentation 66
8.2 Pourcentage des temps par projet. L'axe des abscisses représentent les temps de recherche, les temps d'édition, les temps d'exécution triés par projets. L'axe des ordonnées correspond au pourcentage de ces temps. 69

8.3 Origine des recherches. L'axe des abscisses représente les différentes origines à partir desquelles une action recherche a été effectuée. L'axe des ordonnées correspond au pourcentage de ces origines. 69
8.4 Moyenne de répartition du temps total par fichier (%) pour ECF 72
8.5 Moyenne de répartition du temps total par fichier (%) pour PDE 72
8.6 Moyenne de répartition du temps total par fichier (%) pour jEdit 73
8.7 Moyenne de répartition du temps total par fichier (%) pour JHotDraw 74
8.8 Moyenne de répartition du temps de recherche par fichier (%) pour ECF 76
8.9 Moyenne de répartition du temps de recherche par fichier (%) pour PDE 76
8.10 Moyenne de répartition du temps de recherche par fichier (%) pour jEdit 77
8.11 Moyenne de répartition du temps de recherche par fichier (%) pour JHotDraw 77
8.12 Moyenne de répartition de revisites par fichier (%) pour ECF 79
8.13 Moyenne de répartition de revisites par fichier (%) pour PDE 80
8.14 Moyenne de répartition de revisites par fichier (%) pour jEdit 80
8.15 Moyenne de répartition de revisites par fichier (%) pour JHotDraw 81

9.1 Itération d'une maintenance 88

Liste des tableaux

3.1 Distribution du travail des développeurs 16

6.1 Résumé des tâches de maintenance 51
6.2 Répartition des sujets de l'expérimentation 53

7.1 Résumé des possibilités d'interprétation des données vidéos . 63

8.1 Ensemble des fichiers à modifier par projet 70

Chapitre 1

Introduction

Depuis de nombreuses années, des recherches ont été effectuées pour mieux comprendre les concepts de maintenance logicielle et de compréhension logicielle.

1.1 Mise en contexte du travail

Nous allons définir brièvement ces 2 concepts qui sont à la base de ce mémoire. La maintenance logicielle est présentée à la section 1.1.1 et la compréhension logicielle à la section 1.1.2.

1.1.1 Maintenance logicielle

De nos jours, la plupart des modèles de cycle vie d'un logiciel montrent que la phase de maintenance devient de plus en plus importante et qu'elle consomme plus de la moitié du temps de vie d'un logiciel. Nous en distinguons 3 types principaux et 2 types secondaires [1][4, 27, 32, 23, 1, 16].

Maintenance adaptative Maintenance effectuée après un changement dans les exigences.

Maintenance corrective Maintenance effectuée après la découverte d'un bug.

Maintenance perfective Maintenance effectuée après un changement dans les besoins des consommateurs.

Maintenance préventive Maintenance effectuée pour pallier la menace potentielle d'une erreur ainsi que pour faciliter les maintenances future et la compréhension du système.

Maintenance d'urgence Maintenance effectuée quand un aléa apparait alors qu'une maintenance corrective est déjà en-cours.

1. Ces types de maintenance sont expliqués plus en détail à la section 3.3

La maintenance a un coût tant financier que temporel. Ce coût de maintenance surpasse le coût du développement sur ces 2 points. Les entreprises veulent donc diminuer le coût de maintenance. De nombreuses recherches sont menées dans ce but. Bien que des recherches aient été conduites, les coûts financiers n'ont pu être réduits de manière significative[1] et restent compris, la plupart du temps, entre 60 et 80% de l'argent dépensé pendant la vie totale d'un système d'informations.

Dans une maintenance logicielle, différents modèles sont utilisés pour donner un caractère automatique à la réalisation d'une tâche de maintenance.

1.1.2 Compréhension du logiciel

Le logiciel existant doit être compris avant de pouvoir effectuer une modification en accord avec un plan de maintenance. Réduire cette phase de compréhension permettrait de réduire le temps global de maintenance.

De nombreux outils tentent de fournir des informations de compréhension logicielle en permettant la visualisation de données statiques ou dynamiques du programme étudié.

Les outils fournissent aux développeurs des informations, des modules, des fichiers, des classes qui leur permettraient de comprendre plus vite le logiciel pour entamer au plus tôt la phase d'implémentation.

Dans ce mémoire, nous utiliserons la notion de pertinence de fichier donnée par Lee et Kang[37]. Les auteurs distinguent deux types de pertinence d'entité. Une entité peut être un élément Java ou une ressource. Un élément Java est un fichier, une classe, un champ, une methode, etc. Une ressource peut être un fichier XML, de propriétés, de *META-INF*, etc. Ces définitions sont tirées de Zimmermann et al[26].

Significativement pertinent Une entité significativement pertinente est une entité qui doit être changée pour accomplir la tâche de maintenance.

Hautement pertinent Une entité hautement pertinente est une entité qui permet de comprendre pourquoi et comment une entité significativement pertinente doit être changée.

Lee et Kang[37] ont, en plus, 2 autres types de pertinence mais nous ne les étudierons pas en détail ici.

1.2 But du mémoire

Un des buts de ce mémoire est d'essayer de voir comment une tâche de maintenance logicielle pourrait aider à donner des informations de compréhension logicielle pour la prochaine étape de maintenance. En effet, nous verrons au chapitre 2 qu'il existe de nombreux artéfacts logiciels qui contiennent des informations sur le système. La question est donc de déterminer comment nous pourrions utiliser une phase de maintenance pour en extraire certaines informations sur les fichiers.

Concrètement, ce mémoire vise à trouver un moyen de capturer le déroulement d'une tâche de maintenance et d'analyser les étapes effectuées par le programmeur pour finalement définir des fichiers significativement et hautement pertinents. De plus, ce mémoire tend à dire que les maintenances passées peuvent servir à la compréhension du logiciel pour la maintenance actuelle.

Pour ce faire, nous avons effectué une expérimentation sur des programmeurs devant effectuer des tâches de maintenance et nous avons capturé leurs interactions. Nous avons analysé leurs interactions pour essayer de déceler des critères permettant de différencier un fichier pertinent d'un fichier non pertinent.

La figure 1.1 représente la mise en contexte de notre mémoire. Elle montre une itération de maintenance. Il pourrait être intéressant de fournir des fichiers pertinents avant l'itération, de faire celle-ci et de donner en sortie d'itération des fichiers pertinents. Les fichiers pertinents obtenus à la fin seront utilisés pour la prochaine itération de maintenance. Nous avons l'idée d'un processus de maintenance itératif. Les fichiers pertinents fournis ne serviront que pour un seul projet. Pour ce faire, on divise l'itération de maintenance en 3 étapes : la capture de traces, la résolution de la tâche et l'analyse des résultats.

Ce mémoire va donc étudier la faisabilité d'une telle approche. La capture de traces sera manuelle ainsi que l'analyse des résultats. La résolution de la tâche sera effectuée par l'ensemble des sujets.

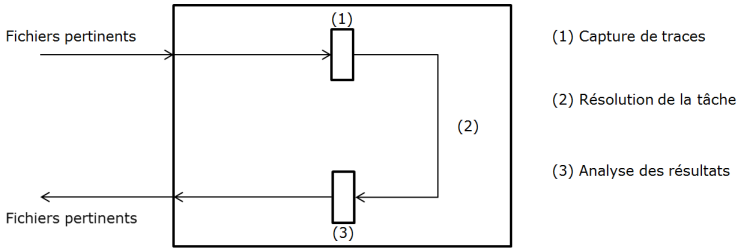


Figure 1.1 – Itération d'une maintenance

1.3 Plan du mémoire

Ce travail est divisé en deux parties.

La première partie présentera l'état de l'art lié à la compréhension logicielle et à la maintenance logicielle aux chapitres 2 et 3, respectivement. Le chapitre 2 dressera les points importants concernant la compréhension logicielle tandis que le chapitre 3 expliquera les différents points théoriques, principes et modèles de la maintenance logicielle.

La deuxième partie de ce travail présentera l'expérimentation. Chaque chapitre donnera de brefs rappels théoriques pour ensuite les mettre en contexte dans notre expérimentation. Le chapitre 4 donnera un bref aperçu du processus expérimental qui structurera la lecture des chapitres 5, 6, 7 et 8. Les chapitres 5 et 6 concerneront la phase préliminaire de l'expérimentation. Le chapitre 7 précisera la phase d'exécution de l'expérimentation. Le chapitre 8 donnera les résultats et leurs menaces.

Enfin, le dernier chapitre résumera les conclusions du travail et proposera des idées pour de futurs travaux dans ce domaine.

Première partie

Etat de l'art

Chapitre 2

Compréhension du logiciel

La compréhension du logiciel est une partie importante de la maintenance logicielle qui est expliquée au chapitre 3. Le logiciel doit être compris avant toute modification. De nombreuses recherches ont été effectuées pour essayer de diminuer le temps de compréhension d'un logiciel inconnu.

Dans ce chapitre, nous présenterons différentes techniques ou artéfacts qui peuvent mener à une meilleure compréhension d'un logiciel. Selon des textes scientifiques relevant ces techniques, il existe deux catégories de techniques se différenciant par la façon dont les propriétés d'un logiciel sont capturées[28]. Le premier groupement concerne l'ensemble des méthodes d'analyse statique, c'est à dire l'ensemble des méthodes extrayant des propriétés d'un logiciel sans devoir exécuter le programme. Le deuxième groupement concerne les méthodes d'analyse dynamique. Ces méthodes requièrent une ou plusieurs exécutions du programme pour obtenir des informations sur celui-ci.

Nous développerons ici certaines méthodes d'analyse statique et dynamique ainsi que les artefacts logiciels sur lesquels elles s'appliquent.

2.1 Méthodes d'analyse statique

Comme expliqué ci-dessus, les méthodes d'analyse statique d'un logiciel consistent à extraire de l'information depuis certains artéfacts d'un logiciel sans même exécuter celui-ci. Ces informations peuvent être extraites depuis plusieurs sources. Nous distinguerons ici 3 types de sources : les documents d'analyse conceptuelle et logique, le code source ainsi que les données externes non-structurées. L'ensemble de ces notions seront expliquées dans leurs sections respectives.

2.1.1 Analyse conceptuelle et logique

Ces documents sont l'ensemble des documents qui représentent les exigences du logiciel. Ils se focalisent uniquement sur ce que fait le logiciel et non sur la manière dont il doit le faire. Il est impératif, au niveau conceptuel, de s'abstraire de toute implémentation. Dans cette section, nous verrons différents moyens de représenter les exigences d'un logiciel par l'analyse des diagrammes de cas d'utilisations, l'analyse des flux de données, l'analyse des diagrammes entités-relations.

Pour l'ensemble des modèles, il existe peu de techniques automatiques d'aide à la compréhension. Il est donc important, pour pouvoir gagner de l'information de ces modèles, de comprendre leur fonctionnement et la façon dont ils sont construits. Ce mémoire ne visant pas ce niveau, nous nous limiterons dans les sections ci dessous à expliquer brièvement ces trois modèles de représentation conceptuelle et/ou logique d'un logiciel.

Diagrammes de cas d'utilisation

Les diagrammes de cas d'utilisation sont composés de trois éléments : les acteurs, les cas d'utilisation et les relations entre les cas d'utilisation.

Acteurs Les acteurs sont ceux qui vont avoir une interaction avec le système. Ils doivent être externes au système. Ceux-ci sont décrits par leurs rôles et capacités à interagir avec le système. Les acteurs sont soit des personnes soit d'autres systèmes qui interagissent dans le système que l'on décrit avec le diagramme de cas d'utilisation. En UML [1], un acteur est représenté par un symbole humanoïde.

Figure 2.1 – Acteur en UML

Les cas d'utilisation Un cas d'utilisation représente une fonctionnalité demandée au système. Celui-ci va représenter l'ensemble des scénarios d'interaction entre le système et le ou les acteurs concernés par la fonctionnalité dans un langage narratif. Les scénarios sont décrits en 2 parties. Ces 2 parties sont le scénario de base et les flux alternatifs. Les flux alternatifs ne sont pas toujours nécessaires mais représentent

1. Universal Modelling Language. Langage développé par l'*OMG*

des interactions qui ne suivent pas le cas normal d'exécution de la fonctionnalité. Par exemple, nous pourrions décrire un flux alternatif si l'interaction est annulée par un acteur. En UML, un cas d'utilisation est représenté par un ovale qui contient le nom du cas d'utilisation.

Figure 2.2 – Cas d'utilisation en UML

Les relations entre cas d'utilisation Les relations entre cas d'utilisation sont de 3 types : les relations simples, les *includes* et les *extends*.En UML, les relations simples sont représentées par des flèches au trait continu et les *inlcudes* et les *extends* par des flèches discontinues surmontées de la notation *includes* et *extends* respectivement.

Les diagrammes de cas d'utilisation permettent de rapidement comprendre l'ensemble des fonctionnalités demandées ainsi que les différents acteurs qui jouent un rôle dans le système[13, 33].

Diagrammes de classes

Les diagrammes de classes spécifient l'ensemble des propriétés statiques de chaque élément d'un domaine d'application.

Chaque classe du diagramme est spécifiée selon des attributs et des opérations. Une classe représente un concept du domaine d'application et non une instance réelle. Les attributs sont définis par leur type. Ces derniers peuvent être un *string*, un *char* ou autre en fonction du langage de programmation utilisé. Les opérations sont définies par un nom, leurs arguments et le type de retour qui peut être *void*. Les opérations ne sont pas définies par leur implémentation. Chaque exigence représente une opération. L'ensemble des classes est mis en relation par des associations. Ces dernières relient et fournissent une structure aux concepts ainsi qu'une hiérarchie dans les concepts.

Le diagramme de classe permet une vue d'ensemble des fonctionnalités et concepts contenus dans le système[33].

Modèle entité-association

Le modèle entité-association représente le domaine d'application au regard des données qu'il manipule. Celui-ci s'abstrait de toute information

d'implémentation technique d'une base de données. Il permet de représenter le domaine d'application d'un système sous formes d'entités et de leurs relations entre elles. Chaque entité sera dotée d'un ensemble de propriétés qui sont appelées attributs[31, 21].

2.1.2 Analyse du code source

Les documents de conception étant très rarement maintenus et accessibles, le code source est l'endroit où les programmeurs cherchent à comprendre le système. Ici, on entend par code source l'ensemble des données structurées par un langage de programmation qui constituent les spécifications exécutables du comportement d'un système. De plus, les systèmes devenant de plus en plus grands, il est difficile de concevoir qu'un nouveau développeur sur le projet puisse comprendre l'ensemble des documents de conception avant d'effectuer une tâche sur le système. Nous allons donc présenter les méthodes d'analyse statique du code source qui permettent une meilleure compréhension du logiciel.

La méthode d'analyse statique la plus couramment utilisée est l'utilisation des métriques. D'ailleurs, un nombre important de celles-ci existe. C'est pourquoi nous nous limiterons à présenter les classes majeures de métriques statiques ainsi qu'à en donner quelques exemples.Cette section est basée sur la thèse de doctorat de Caserta effectuée en 2012[30]. Si celles-ci sont généralement utilisées pour mesurer la qualité d'un logiciel, elles permettent aussi d'extraire de l'information permettant la compréhension d'un logiciel.

Les métriques de taille des composants Ces métriques sont les plus simples à calculer. Elles donnent rapidement une idée de la taille d'un composant par rapport à un autre ou par rapport au système complet. On peut donc rapidement voir où sont les composants plus importants par la taille. Les métriques de taille les plus souvent utilisées sont les *Lines of code(LOC)* et les *Number of classes(NOC)*. Les métriques de LOC calculent le nombre de lignes de code que contient un composant tandis que celles de NOC calculent le nombre de classes que contient un composant.

Les métriques de complexité des composants Ces métriques permettent de juger la complexité d'un composant logiciel. Les métriques de complexité sont utilisées dans la compréhension logicielle ainsi que dans la mesure de qualité d'un logiciel. En effet, on peut considérer qu'un composant complexe sera difficile à maintenir et à comprendre. La métrique la plus utilisée car indépendante du langage de programmation est le nombre *cyclomatique* de Mc Cabe. Cette métrique calcule le nombre de branches ou chemins qu'il est possible de prendre lors de l'exécution d'une méthode.

Les métriques de couplage entre les composants Les métriques de couplage permettent d'identifier ou de calculer les dépendances entre les composants. Plus les composants sont couplés, plus ils sont difficiles à comprendre. Bien qu'un bon nombre de métriques ait été trouvées, la métrique la plus utilisée est le *Couplage entre objets(Coupling Between Objects)* qui va calculer le nombre de classes couplées à la classe mesurée.

Les métriques de cohésion entre les composants Ces métriques calculent le nombre de connexions existantes dans une même classe. Elles vont donc mesurer la proportion des attributs utilisés par les méthodes de la classe par rapport à ceux qui sont instanciés par des méthodes ne venant pas de cette même classe. Ceci calcule la capacité d'une classe à instancier ses attributs. La métrique de complexité la plus utilisée est celle du *Manque de cohésion sur les méthodes(Lack of Cohesion On Methods)*.

Les métriques d'héritage Ces métriques calculent l'importance de l'héritage dans un système. Deux métriques sont couramment utilisées. Celles-ci sont la *profondeur de l'arbre de l'héritage(Depth in Inheritance Tree)* et le *nombre d'enfants d'une classe(Number of Children of a Class)*. Les métriques d'héritage sont utilisées pour juger de la complexité et de l'intensité d'utilisation de classes héritées dans un système.

L'ensemble des métriques de base présentées ci-dessus permettent de comprendre une partie de la conception et de l'organisation d'implémentation du système. Cependant, les nombreux outils utilisent des combinaisons de celles-ci ou des conditions logiques entre elles pour permettre une meilleure visualisation. Par exemple, on peut dire qu'une classe dont la complexité est forte et la cohésion est basse sera plus difficilement compréhensible et maintenable qu'une classe dont la complexité est faible et la cohésion forte.

2.1.3 Analyses des données externes non-structurées

Les recherches se portent de plus en plus sur les éléments extérieurs du logiciel pour aider à la compréhension et à la maintenance. Nous résumerons dans cette section les artéfacts qui contiennent des données non-structurées et ensuite les techniques qui permettent d'obtenir des informations sur celles-ci[38]. Nous reprendrons pour cela la définition de données non-structurées formulée par Manning : "Les données non-structurées sont les données qui n'ont pas une structure claire, sémantiquement déclarée, facile à comprendre par un ordinateur"[8]. Les données non-structurées correspondent donc aux données écrites dans un langage naturel.

Données non-structurées

Ces données peuvent être comprises dans différents endroits du logiciel ou dans l'environnement de développement : le code source, les bases de données de bugs, les chaines d'email ainsi que les systèmes de contrôles de versions.

Code source Le code source comprend l'ensemble des commentaires, noms de variables, méthodes ainsi que tous les littéraux compris dans les commandes d'impression vers une source externe comme la console ou des fichiers de logs.

Base de données de bugs Dans les bases de données de bugs telle que Bugzilla [2], nous pouvons obtenir de l'information à partir des *bug reports* [3].

Chaines d'email Dans les chaines d'email ainsi que dans les enregistrements des messageries instantanées fournies par certains logiciels de développement, les programmeurs laissent des informations pertinentes sur un système.

Système de contrôle de version Les systèmes de contrôle de version permettent d'inscrire un message lors de la publication d'une nouvelle version.

Techniques d'extraction des données non-structurées

Nous présenterons ici les étapes générales de pré-traitement de l'information et ensuite les différents systèmes de récupération d'information[38].

Pré-découpage La première phase d'extraction de données est le prétraitement de celles-ci. Elle consiste à enlever l'ensemble des bruits et données parasites pour obtenir un document contenant un maximum de données pertinentes. Cette phase comprend la *tokenization*, l'*identifier splitting*, le *stop word removal*, le *stemming* et le *pruning*.

- La *tokenization* consiste à casser le texte en mots, appelés tokens, tout en enlevant l'ensemble de la ponctuation ainsi que les caractères numériques.
- L'*indentifier splitting* consiste à diviser les noms de variables ou méthodes en utilisant certaines techniques telles que le *camel case* [4].

2. `http://www.bugzilla.org`

3. Un bug report est un message qui est posté dans une base de données de bugs et il permet d'avertir les autres développeurs d'un bug présent, avec une description, dans le système. De plus, des commentaires et patchs sont attachés aux bug reports

4. Le camel case est une technique écrivant les noms des méthodes et variables en montrant la séparation entre plusieurs mots par une majuscule. Un exemple de camel case est la fonction "ajout fonctionnalité" qui devient, avec cette technique, "ajoutFonctionnalité".

– Le *stop word removal* consiste à retirer les mots de liaisons ne transportant aucune information [5].
– Le *stemming* consiste à rechercher la racine des mots avec pour effet de supprimer les bruits liés à la conjugaison d'un verbe ou au pluriel d'un mot.
– Le *pruning* consiste à enlever l'ensemble des mots qui apparaissent trop souvent ou qui n'apparaissent pas assez souvent. En effet, ces mots pourraient apporter du bruit[35].

Systèmes de récupération de l'information D'après la définition de Manning[8], la récupération d'information permet de trouver parmi une large collection de documents, un document précis selon notre besoin. Il existe de nombreux systèmes qui permettent de récupérer de l'information. Nous allons présenter ici les 3 principaux évoqués que sont le *Vector Space Model*, le *Latent Semantic Indexing* et le *Latent Dirichlet Allocation*[38].

– le *Vector Space Model* est un modèle algébrique qui se base sur une matrice M termes sur N documents. Chaque entrée de la matrice représente le poids de ce terme dans le document. Pour calculer la similarité entre deux documents, il faut se baser sur les vecteurs colonnes. Pour éviter l'influence des termes trop communs, un système de pondération permet de minimiser ceux-ci[11].
– le *Latent Semantic Indexing* est une extension du *Vector Space Model*. En effet, il réduit la dimension de la matrice Termes/Documents d'un certain facteur K en regroupant des termes similaires en K sujet. Ceci permet de supprimer le bruit causé par la synonymie et la polysémie très courantes en langage naturel[36].
– Le *Latent Dirichlet Allocation* est un modèle probabiliste se basant sur une approche différente des deux autres modèles présentés ci-dessus. Il fait l'hypothèse que le corpus [6] contient un certain nombre de sujets et que dans chaque sujet, un terme a une certaine probabilité d'être généré. Il a l'avantage de privilégier la sémantique plutôt que la syntaxe[9].

2.2 Les méthodes d'analyse dynamique

L'analyse dynamique d'un logiciel, signifie en opposition à l'analyse statique, l'analyse du programme en exécution. Cela permet d'obtenir un ensemble d'informations générées lors de l'exécution d'un programme. Cette analyse permet aussi de se focaliser sur des scénarios spécifiques d'utilisation.

5. En français, les mots comme "et" et "ils" n'ont aucune information.
6. Le corpus est l'ensemble des documents. Voir [36].

Ce type d'analyse a été largement documenté depuis un certain nombre d'années. C'est pourquoi une étude fut menée pour tirer une vue d'ensemble du domaine[3].Le but du mémoire ne visant pas cette matière nous nous limiterons à donner ici les conclusions de cette étude.

Pour caractériser les différents articles qu'ils ont regroupés entre 1999 et 2008 dans cette étude, les auteurs[3] ont défini quatre facettes : l'activité, la cible, la méthode et l'évaluation.

Activité L'activité décrit ce qui est fait dans l'article

Cible La cible représente le type de logiciel ciblé par l'approche de l'article.

Méthode La méthode est la méthode d'analyse dynamique utilisée pour effectuer l'activité

Évaluation L'évaluation est la manière dont l'approche est validée.

Les auteurs ont ensuite défini pour chaque facette un ensemble d'attributs et affecté chaque article représentatif pour l'étude [7] d'un attribut par facette. Ceci a permis de connaitre les sujets les plus visés par la littérature.

Après regroupement de certains attributs, les auteurs ont calculé le nombre d'articles par attribut et par facette. Ci-dessous, se trouve l'attribut trié par facette dont le nombre d'articles en relation avec celui-ci est le plus grand.

Activité L'activité principale des documents scientifiques est de produire des vues d'ensemble de propriétés dynamiques.

Cible La plupart des systèmes ciblés sont des systèmes orientés objets.

Méthode La plupart des articles utilisent comme méthode des techniques de visualisation.

Évaluation En ce qui concerne l'évaluation, la plupart des articles étaient validés par l'utilisation de cas d'études open-source.

7. L'ensemble de l'étude a été réduite à l'étude de 172 articles

Chapitre 3

Maintenance logicielle

La maintenance logicielle [1] est l'une des étapes fondamentales dans le cycle de vie d'un système informatique.

Nous discuterons d'abord de la vision que nous avons de la maintenance et de ce que nous cherchons à faire avec celle-ci. Ensuite, nous évoquerons les différents types possibles de maintenance ainsi que les processus qui peuvent être mis en place pour les effectuer. Puis, nous parlerons des coûts engendrés par une maintenance. Enfin, nous verrons les modèles qui peuvent s'appliquer aux maintenances.

3.1 Vision d'une maintenance

Chaque maintenance est vue comme une entreprise temporaire initiée dans le but de créer un produit, un service ou un résultat unique[15]. Une maintenance est aussi un processus unique, qui consiste en un ensemble d'activités coordonnées et maitrisées comportant des dates de début et de fin, entrepris dans le but d'atteindre un objectif conforme à des exigences spécifiques telles que des contraintes de délais, de coûts et de ressources[17]. Les différentes maintenances d'un système sont donc vues comme des projets à part entière[24]. Cela nous conduit à dire que chaque maintenance est un projet unique où nous prenons en compte les caractéristiques propres et où nous adaptons la conduite du projet à chaque cas. De plus, une maintenance n'est ni une activité répétitive ni une mission permanente car elle a une date de début et de fin déterminée. Ceci la rend temporaire mais avec un résultat durable[24].

En outre, les maintenances incluent un ensemble très large d'activités comme la correction d'erreurs, l'ajout et la suppression de fonctionnalités et

1. La maintenance logicielle est appelée, dans la suite de ce travail, maintenance

l'optimisation du système tant au niveau de la vitesse d'exécution que de l'architecture du système en passant par l'amélioration de la maintenabilité du logiciel[23].

De plus, le pourcentage de développeurs qui doivent effectuer des opérations de maintenance est en constante augmentation. En effet, comme nous pouvons le voir ci-dessous, en 1950, 1 développeur sur 10 effectuait des maintenances tandis qu'en 2010, 65% des développeurs travaillaient sur la maintenance et l'évolution de système. De plus, les analystes prévoient qu'en 2020, 70% des développeurs travailleront sur des projets de maintenance[32] comme le montre la table 3.1.

Year	New projects	Enhancements	Repairs	Total
1950	90	3	7	100
1960	8,500	500	1,000	10,000
1970	65,000	15,000	20,000	100,000
1980	1,200,000	600,000	200,000	2,000,000
1990	3,000,000	3,000,000	1,000,000	7,000,000
2000	4,000,000	4,500,000	1,500,000	10,000,000
2010	5,000,000	7,000,000	2,000,000	14,000,000
2020	7,000,000	11,000,000	3,000,000	21,000,000

Table 3.1 – Distribution du travail des développeurs

3.2 But d'une maintenance

Une maintenance est effectuée pour suivre les besoins, en constante évolution, des systèmes d'information. Ces besoins d'évolution peuvent être tout autant des changements dans les exigences du système que dans l'environnement d'exécution dudit système[29].

Une maintenance est due à 3 éléments que sont la correction d'erreurs, le *business pull* et l'*IT push*[32].

Correction d'erreur La correction d'erreurs dans une maintenance consiste en la découverte d'un bug et en la résolution de celui-ci.

Business pull Le *business pull* réside dans tous les changements de haut-niveau qui peuvent modifier le comportement d'un système. Ces changements peuvent être une modification dans une loi ou dans les exigences de l'entreprise, ce qui fait que le logiciel doit évoluer pour respecter les nouvelles lois ou exigences lui permettant ainsi de continuer à survivre. Mais le

business pull ne se résume pas qu'à ces changements. En effet, une maintenance doit être effectuée lors d'une réorganisation, d'une fusion ou d'une acquisition d'entreprises pour que l'organisation résultante de ces changements puisse travailler avec le système. Une dernière partie du *business pull* est la création et suppression d'un produit. Les besoins des utilisateurs font également partie du *business pull*.

IT push L'*IT push* consiste en l'avènement de nouvelles technologies informatiques qui vont changer la relation avec le système. Nous pouvons citer l'apparition d'Internet, des appareils mobiles mais aussi de nouveaux systèmes d'exploitation et de nouveaux matériels.

3.3 Types de maintenance

Il existe 3 types principaux de maintenance pour un système informatique. Ce sont les maintenances adaptatives, correctives et perfectives[4, 27, 32]. Une évolution logicielle est perçue comme une maintenance adaptative et perfective. Il existe deux autres types de maintenance que sont la maintenance préventive[23, 1] et la maintenance d'urgence[16]. Ces types sont catégorisés en fonction du but de la maintenance.

Adaptative La maintenance adaptative est effectuée suite à un changement dans l'environnement ou dans les exigences.

Corrective La maintenance corrective est effectuée suite à la découverte d'une erreur qui n'a pas été détectée à la livraison du produit. Les erreurs peuvent survenir à cause d'une mauvaise conception du design, une logique métier incorrecte ou des erreurs de codage[1].

Perfective La maintenance perfective est effectuée suite à un changement des besoins des consommateurs. Cette maintenance est vue comme une amélioration fonctionnelle et non-fonctionnelle du système pour que celui-ci tende vers la perfection[1]. En effet, le succès d'un logiciel n'est pas fait dès sa première itération mais bien par les itérations suivantes où le logiciel s'est adapté aux besoins des consommateurs et que celui-ci soit devenu parfait pour eux.

Préventive La maintenance préventive est effectuée pour pallier la menace potentielle d'une erreur mais aussi pour faciliter la maintenance et la compréhension du système[23] car les 3 premiers types de maintenance ajoutent de la complexité. Cette maintenance est vue comme du *refactoring* de code quand le système fonctionne correctement. Ce type de maintenance

s'effectue aussi pour avoir une documentation du système qui reste cohérente avec le code qu'elle explique.

D'urgence La maintenance d'urgence est une opération non prévue qui permet de garder le système opérationnel alors qu'une maintenance corrective est déjà en cours. Ce type de maintenance fait partie des maintenances correctives[16].

Après avoir détaillé ces 5 types de maintenance, nous pouvons observer qu'une maintenance *traditionnelle* ne reprend que la maintenance corrective car les autres sont vues comme une évolution du logiciel[32, 27]. La maintenance d'urgence est bien entendu une maintenance *traditionnelle* mais faisant partie des maintenances correctives, elle n'est pas différente de celles-ci.

Ces types de maintenance n'ont pas le même pourcentage de répartition durant le cycle de vie d'un système. Leurs différents pourcentages restent sensiblement identiques selon les auteurs.

Comme nous pouvons le voir sur la figure 3.1, Klint[32], qui ne distingue que 3 types de maintenance, estime qu'un peu plus de la moitié des maintenances est de type perfective et que le reste de celles-ci se partage de manière relativement équivalente entre les maintenances correctives et adaptatives. Cette répartition induit que 80% des maintenances sont vues comme des évolutions du logiciel.

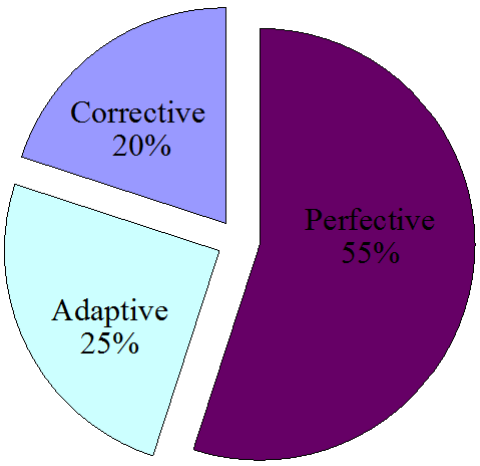


Figure 3.1 – Pourcentage de répartition des types des tâches de maintenance pour Klint

Pour Aggarwal et Senghi[23], malgré les maintenances préventives, les pourcentages de répartition sont quasiment équivalents comme nous pouvons le voir à la figure 3.2.

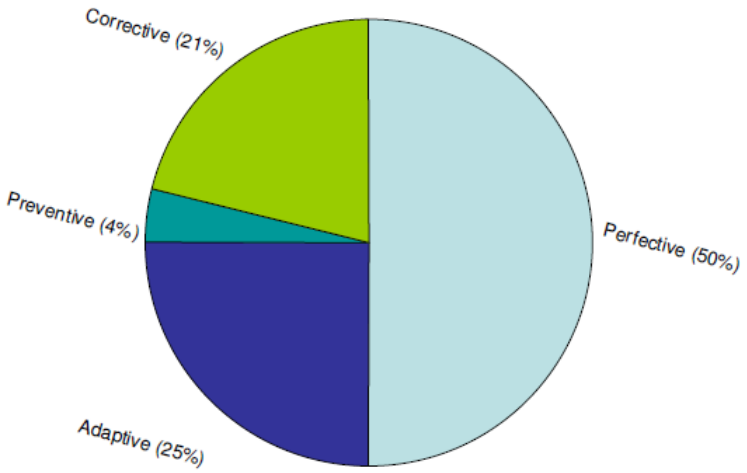


Figure 3.2 – Pourcentage de répartition des types des tâches de maintenance pour Arrgarwal et Singh

Il ne faut pas oublier que certains auteurs donnent des noms différents pour ces types de maintenance et que ces derniers peuvent avoir un sens différent d'un auteur à l'autre. Par exemple, Sommerville[14] cite 3 types de maintenance que sont la réparation d'erreurs, l'adaptation d'environnement et l'ajout de fonctionnalités. La réparation d'erreurs correspond à une maintenance corrective et nous sommes dans le cas d'un simple changement de noms. L'adaptation d'environnement est une adaptation du système pour un nouvel environnement alors qu'une maintenance adaptative est une adaptation à un nouvel environnement mais aussi à de nouvelles exigences. Nous sommes alors dans un changement de sens. De même qu'une maintenance perfective signifie l'ajout de fonctionnalités ou l'amélioration de la structure et des performances, une adaptation du système n'ajoute que des fonctionnalités à celui-ci et nous nous retrouvons dans le cas d'un changement de sens.

3.4 Étapes d'une maintenance

Lors d'une maintenance, différentes étapes sont réalisées pour que le système atteigne le résultat attendu. Ces étapes forment un processus de maintenance. Ce processus commence toujours par une requête de changement. Différents processus existent et ont un nombre d'étapes variables. Les pro-

cessus sont importants car ils donnent des guides pour dire ce qui doit être fait et comment cela doit être fait[7] [2].

3.4.1 Le processus selon l'IEEE

Le processus de maintenance proposé conjointement par l'*IEEE* et l'*ISO* explique les différentes étapes qui peuvent arriver entre la fin du développement initial du système et la mise à la retraite de ce dernier[16]. Il comporte 6 étapes montrées par le figure 3.3.

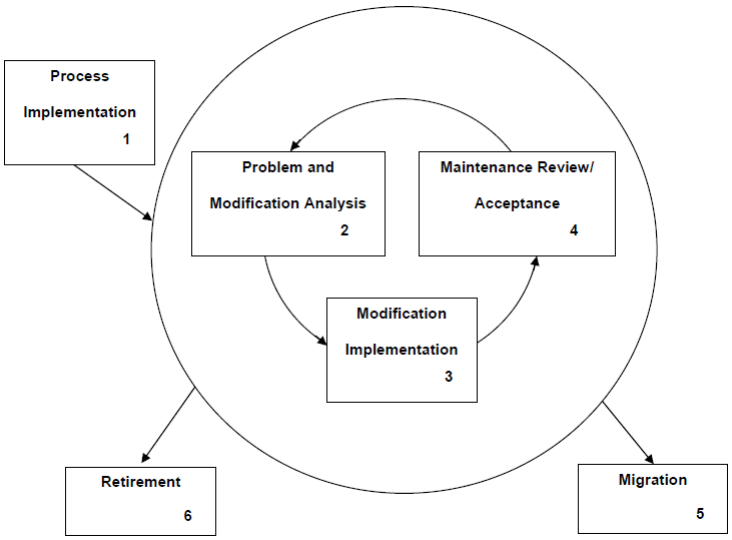


Figure 3.3 – Processus de maintenance IEEE

Les étapes 2, 3 et 4 sont celles qui seront exécutées plusieurs fois durant la vie du logiciel tandis que les étapes 1, 5 et 6 ne sont effectuées qu'une seule et unique fois sur le système.

Implémentation du processus Cette étape consiste à établir les plans et les procédures qui seront exécutés durant le processus de maintenance.

Analyse du problème et de la modification Cette étape est effectuée quand survient une requête pour modifier le système. Ce dernier est alors analysé pour détecter les modifications qui devront être réalisées. Ces modifications sont analysées à leur tour pour évaluer les conséquences qu'elles auront sur l'organisation et le système. Quand les conséquences sont jugées acceptables, les modifications sont approuvées pour être implémentées.

2. Les modèles définis au point 3.6 sont aussi importants et donnent les mêmes avantages.

Implémentation de la modification Cette étape consiste à implémenter les modifications et à les tester.

Acceptation de la maintenance Cette étape consiste à s'assurer que les modifications sont correctes et qu'elles ont été effectuées avec les standards appropriés ainsi qu'avec la bonne méthodologie.

Migration Cette étape consiste à documenter et à développer les étapes requises pour effectuer une migration vers un nouvel environnement. Cette migration doit être conforme avec la norme ISO 12207. Cela signifie qu'un plan de migration doit être créé. De plus, les utilisateurs doivent être formés en plus d'être avertis du début de la migration et de la fin de celle-ci. En outre, les impacts cette migration ainsi que toutes les données doivent être archivés.

Retraite Cette étape consiste en la fin de vie du système. Une analyse doit être effectuée pour aider à la décision de mettre, le cas échéant, le produit à la retraite. Cette analyse est la plupart du temps basée sur l'économie. De plus, des aspects comme l'utilité de garder de vieilles technologies, de passer à une nouvelle en changeant de projet ou de développer un nouveau système pour en avoir un plus modulaire, plus facile à maintenir, respectant un standard ou pour faciliter l'indépendance par rapport à un vendeur sont à prendre en compte avant de mettre un système à la retraite.

3.4.2 Le processus selon Aggarwal et Singh

Ce processus consiste en 5 étapes chronologiques : l'identification du problème, la compréhension, la proposition, la prise en compte des répercussions et les tests[23].

Identification du problème Cette étape consiste à identifier la source de la requête de changement et à savoir si c'est une maintenance pour l'ajout de fonctionnalités, pour la correction d'erreurs, pour la suppression de fonctionnalités obsolètes ou pour une optimisation.

Compréhension Cette étape consiste à analyser le système pour en saisir le fonctionnement et appréhender comment et où effectuer la maintenance. La compréhension logicielle passe par l'analyse des différents documents définis au chapitre 2 et par l'utilisation des techniques expliquées aussi au chapitre 2.

Proposition Cette étape consiste en l'implémentation d'une solution pour résoudre la requête qui a été définie à l'étape d'identification du problème.

Prise en compte des répercussions Cette étape consiste en la prise en compte de tous les effets secondaires que peut avoir, pour le système, l'implémentation de la solution. De plus, lors de cette étape, nous essayons de nous prémunir contre ces effets secondaires.

Tests Cette étape consiste à tester le système après l'implémentation de la solution et à véri er que ce dernier se comporte correctement et garde au moins le même niveau de abilité qu'avant les changements. Si les tests sont négatifs, nous revenons à la première étape.

Ce processus est représenté par la gure 3.4.

Figure 3.4 Processus de maintenance logicielle

3.5 Coûts d'une maintenance

Quand nous parlons de maintenance, nous voyons les types de maintenance à faire, les changements à produire, les étapes à faire mais aussi les coûts associés aux modi cations à e ectuer. Ces coûts sont tant humains et nanciers que temporels. De plus, le temps et l'argent sont corrélés de manière proportionnelle, ce qui fait que, lorsque le temps augmente, l'argent dépensé augmente également.

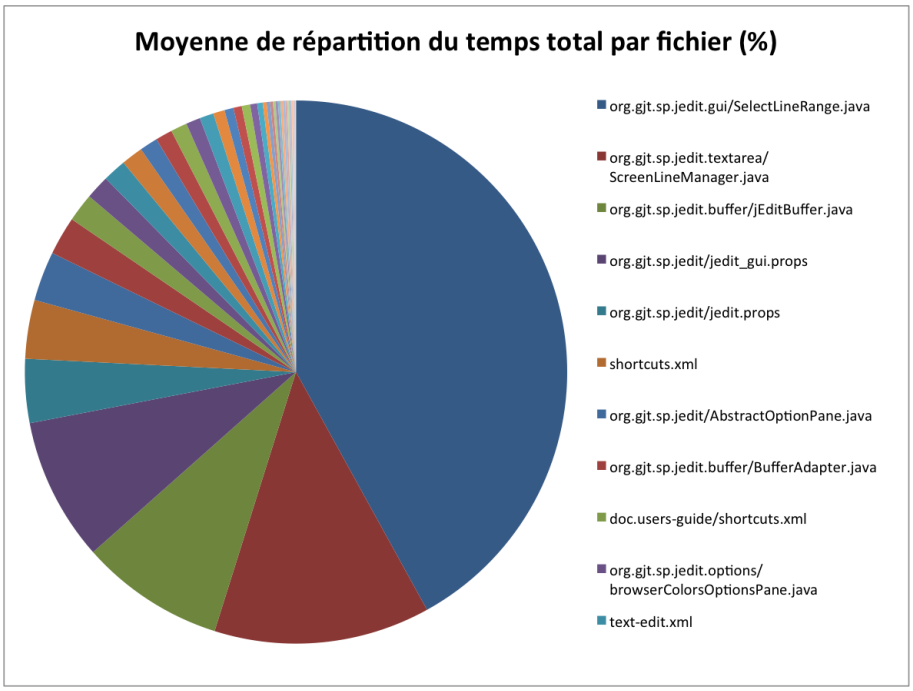


Figure 8.6 – Moyenne de répartition du temps total par fichier (%) pour jEdit

Figure 8.7 Moyenne de répartition du temps total par chier (%) pour JHotDraw

Have you already developed an Eclipse plugin with the Eclipse IDE?

- ☐ Yes
- ☐ No

What do you know about the architecture of an Eclipse plugin?

_____

_____

What is your knowledge of plugin development?

(1 = very bad and 5 = very good)

☐ 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5

What is your knowledge of framework development?

(1 = very bad and 5 = very good)

☐ 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5

A.2 Formulaire projet Java simple

Subject: _____

Pre-experiment questionnaire

What is your gender?

- ☐ Male
- ☐ Female

What is your study level?

- ☐ Bachelor
- ☐ Master
- ☐ Ph. D
- ☐ Post-Doc
- ☐ Professor
- ☐ Other : _____

What is your proficiency in English?

(1 = very bad and 5 = very good)

☐ 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5

How many year(s) have you been programming in Java?

What is your experience/knowledge of Java?

(1 = very bad and 5 = very good)

☐ 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5

Do you use the Eclipse IDE to develop Java program?

- ☐ Yes
- ☐ No

How many years have you been using the Eclipse IDE?

What is your experience/knowledge of the Eclipse IDE?

(1 = very bad and 5 = very good)

☐ 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5

B Résumé des projets

B.1 ECF

ECF

Description:

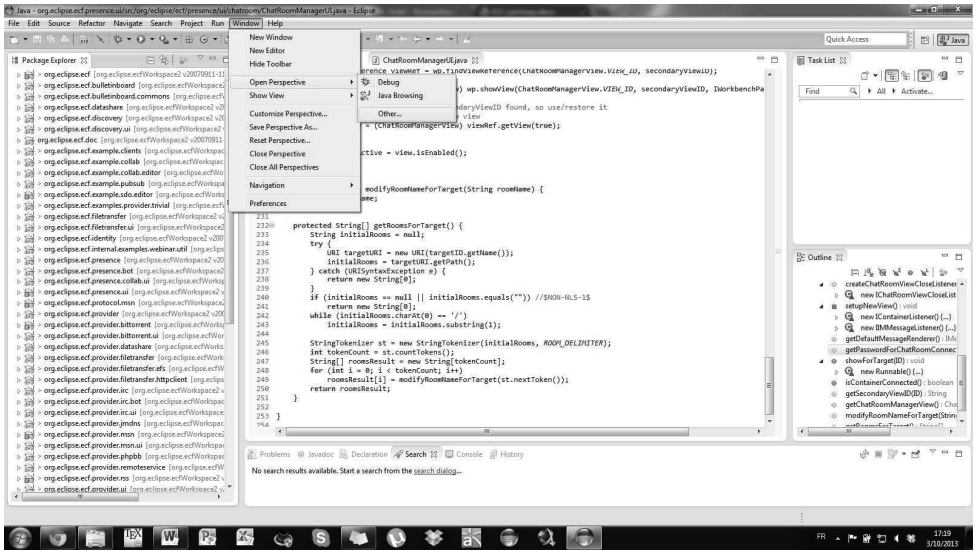
“Eclipse Communication Framework”, aka ECF, is a framework for building distributed servers, applications, and tools. It can also be viewed as an Eclipse plugin to make communications between people easy.

For example, when developers work, they need to talk to other developers without switching from their Eclipse IDE to chat or talk. ECF allows developers to communicate through their Eclipse IDE.

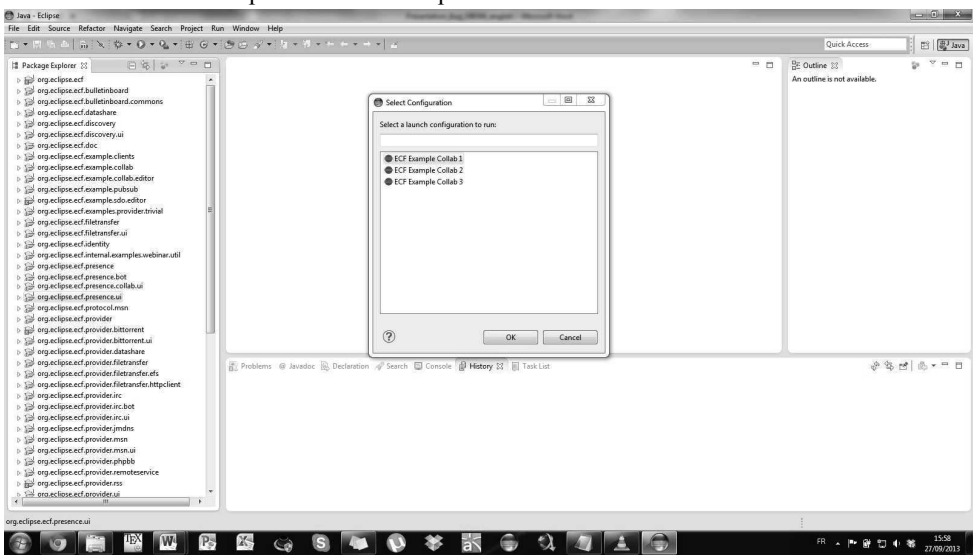
Example:

For example, if you want to connect to IRC (Internet Relay Chat) with ECF, then you can follow those steps:

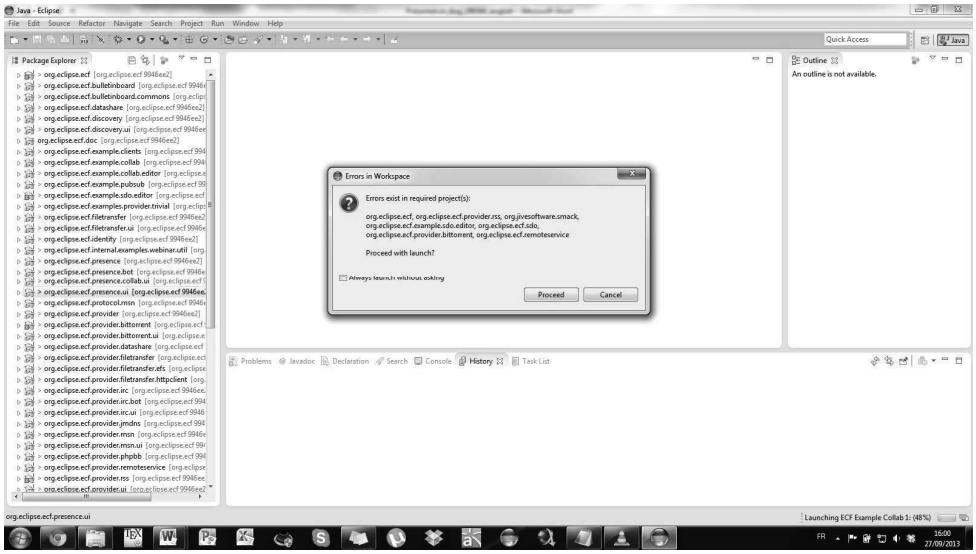
1. Open communication perspective. “Window ->Open Perspective ->Other”



3. Choose the “ECF Example Collab 2*”* and press “OK”.



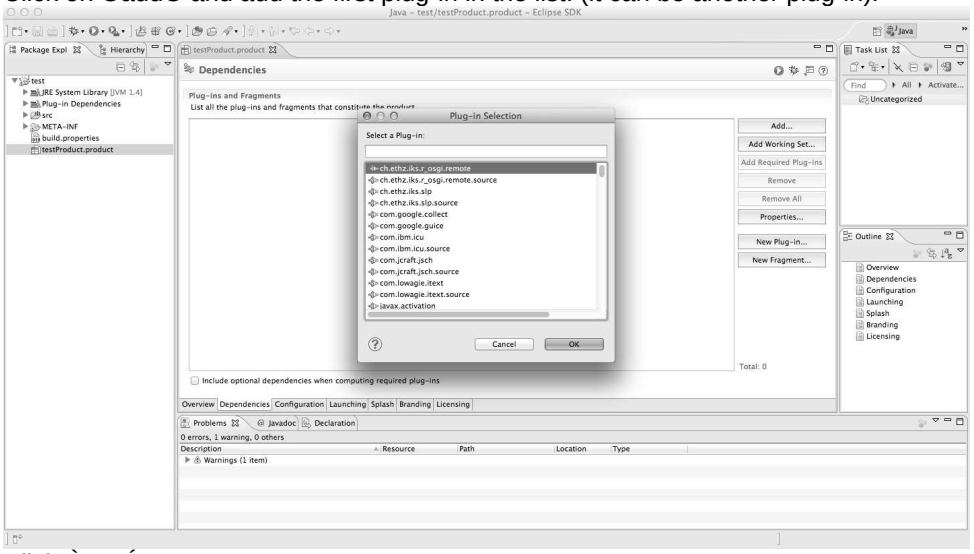
4. Ignore the error message by clicking on “Proceed*”*.



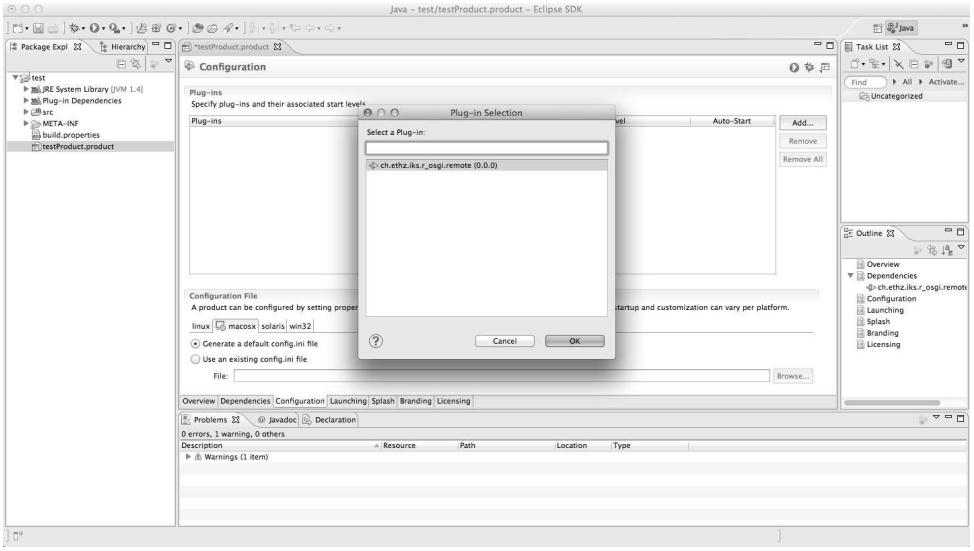
5. Wait some time to see an Eclipse window appear.

!

8. Click on ÒaddÓ and add the first plug-in in the list. (it can be another plug-in).



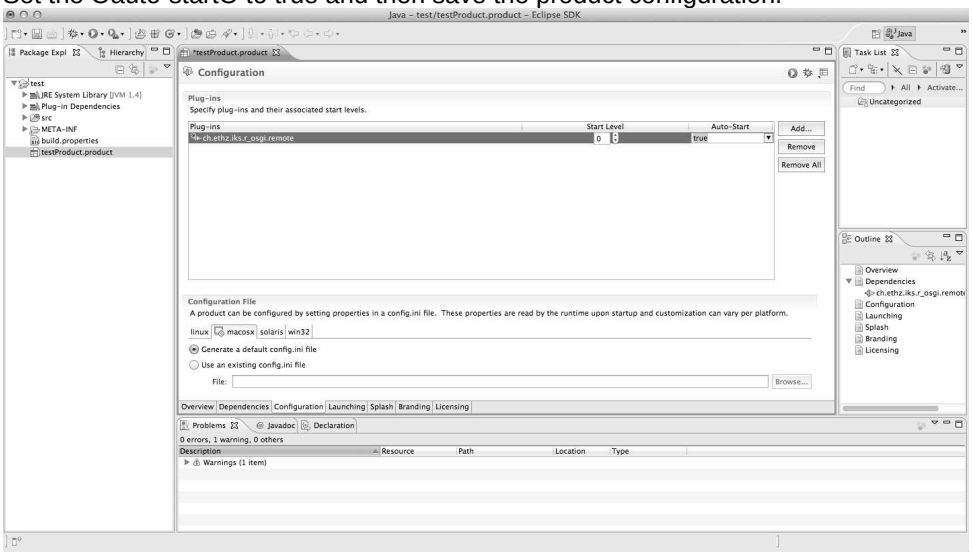
9. Click ÒOKÓ.
10. Go to ÒConfigurationÓ tab and click on Òadd Ò. Select the plug-in in the list and click on ÒOKÓ.



! (#$!

!

11. Set the ÒauteStartÓ to true and then save the product configuration.



12. Go by the file explorer of the computer (not in eclipse) at the same level of the workspace of Eclipse (Desktop -> runtimeEclipseApplication) and search the folder ÒruntimeEclipseApplicationÓ. Open it and you will find your ÒtestÓ project. Open it and open the file Ò.productÓ with a text editor. Look at the value of the variable ÒautoStartÓ and here is the
Bug !

```
<?xml version="1.0" encoding="UTF-8"?>
<?pde version="3.5"?>

<product useFeatures="false">

   <configIni use="default">
   </configIni>

   <launcherArgs>
      <vmArgsMac>-XstartOnFirstThread -Dorg.eclipse.swt.internal.carbon.smallFonts</vmArgsMac>
   </launcherArgs>

   <plugins>
      <plugin id="ch.ethz.iks.r_osgi.remote"/>
   </plugins>

   <configurations>
      <plugin id="ch.ethz.iks.r_osgi.remote" autoStart="false" startLevel="0" />
   </configurations>

</product>
```

!

)#$!

!

13. The right version has to be.

```
<?xml version="1.0" encoding="UTF-8"?>
<?pde version="3.5"?>

<product useFeatures="false">

   <configIni use="default">
   </configIni>

   <launcherArgs>
      <vmArgsMac>-XstartOnFirstThread -Dorg.eclipse.swt.internal.carbon.smallFonts</vmArgsMac>
   </launcherArgs>

   <plugins>
      <plugin id="ch.ethz.iks.r_osgi.remote"/>
   </plugins>

   <configurations>
      <plugin id="ch.ethz.iks.r_osgi.remote" autoStart="true" startLevel="0" />
   </configurations>

</product>
```

14. Close the launched Eclipse and go back to source code.

C.3 jEdit